Neural Net Consituency Parser on Penn Treebank Dataset

Ananya Appan (IMT2017004), Seelam Lalitha (IMT2017027), Swasti Shreya Mishra (IMT2017043)

April 2020

1 Introduction

The Penn Treebank is a dataset built on over a million words of 1989 Wall Street Journal material. Annotated manually according to the Treebank bracketing style, it is designed to allow the extraction of simple predicate/argument structure.

Constituency parsing aims to extract a constituency-based parse tree from a sentence that represents its syntactic structure according to a phrase structure grammar. Recent approaches convert the parse tree into a sequence following a depth-first traversal in order to be able to apply sequence-to-sequence models to it.

The purpose of this assignment is to build a constituency parser for the Penn Treebank dataset using the power of Neural Networks.

2 Approach Followed

Our model is a span based parsing model that uses various architectures of neural network architectures for constituency parsing. It consists of a single scoring function s(i, j, l) which assigns a score to the label for span(i, j). The score of a tree T is defined as sum of all internal nodes of the labeled span scores i.e s(T) = $\sum_{(i,j,l)\in T}$ s(i, j, l). So, the main solution to our parsing problem is to find the tree with highesh score i.e $\tilde{T}$ = argmax(s(T)) $\forall$ T. Implementation of s(i, j, l) is mainly done using the following approaches

2.1 Word Representation

We use word embeddings to represent words. So, we have different embeddings for different works used in the training vocabulary and we map every unknown word in the testing vocabulary to an $< UNK >$ token

2.2 Span Representation

Given an input sentence, to compute span(i, j), we first run a bi-directional LSTM over the word representations of the input and compute f_i, f_j, b_i, b_j which signifies forward and backward representations of fenceposts i and j. Span(i, j) represented by r_{ij} is computed by the concatenation of the respective forward and backward span differences

$$r_{\text{ij}} = [f_i - f_j, b_i - b_j]$$

2.3 Label Scoring

Finally, we implement the label scoring function by feeding the span representation through a onelayer feedforward network whose output dimensionality equals the number of possible labels. The score of a specific label l is the corresponding component of the output vector:

$$\text{s(i, j, l)} = [W_2 g(W_1 r_{\text{ij}} + z_1) + z_2]_{\text{l}}$$

where
W_1, W_2 are weights corresponding to the feed forward network
r_{ij} the representation of span encoding span(i, j),
z_1, z_2 are biases respectively and
g is an element-wise ReLU non linearity

2.4 Inference

We employed a CKY-style algorithm for efficient globally optimal interface. We use the following optimised algorithm to compute our scores

- Let $s_{\text{best}}(i, j)$ denote the score of the best subtree spanning (i, j). We compute for spans of length one i.e

$$s_{\text{best}}(i, j) = max_l s(i, i + 1, l)$$

- To compute general spans, we use the following recursive equation

$$s_{\text{best}}(i, j) = max_l s(i, j, l) + max_k [s_{\text{best}}(i, k) + s_{\text{best}}(k, j)]$$

- We can independently select the best label for the current span and the best split point, where the score of a split is the sum of the best scores for the corresponding subtrees.

2.5 Training

Our training objective is to maximize the hinge loss, where the hinge loss is defined as follows:

$$\text{Hinge loss} = \max_T [s(T) + \Delta(T, T^*)] - s(T^*)$$

Here, T is our predicted tree and T^* is the gold tree. Also, $\Delta(T, T^*)$ is the hamming loss corresponding to the predicted tree T and the gold tree T^*. This is equal to 0 when all constraints are satisfied, or the magnitude of the largest margin violation otherwise. Also, since Δ decomposes over spans, we replace $s(i, j, l)$ with $s(i, j, l) + 1[l \neq l^*_{ij}]$, where l^*_{ij} is the label of span (i, j) in the gold tree T^*.

3 Practical Implementation

In order to implement our code, we used *this code* as a reference. We implemented our neural network using pytorch. We created the following files for the mentioned purposes.

3.1 parse.py

Sentence Formatting : We append (START, START) to the start and (STOP, STOP) to the end of each sentence. In addition to this, in order to prevent overfitting, we replace a small fraction of words with the UNK token.

Obtaining our Embeddings : We create embeddings for each tag and word in our vocabulary. Our final embedding is taken as the concatenation of the tag and word embedding. This is fed into a bidirectional lstm to get forward and backward embeddings.

```
embeddings = torch.cat([self.tag_embeddings(indices[0]), self.
    word_embeddings(indices[1])],-1)

lstm_outputs, _ = self.lstm(embeddings.unsqueeze(1))
```

Listing 1: Obtaining embedding as lstm output

Span Encoding Generation : The aforementioned forward and backward embeddings are concatenated to get the span encoding. This is done as described in our approach.

```
def get_span_encoding(left, right):
        forward = (
            lstm_outputs[right][0][:self.lstm_dim] -
            lstm_outputs[left][0][:self.lstm_dim])
        backward = (
            lstm_outputs[left + 1][0][self.lstm_dim:] -
            lstm_outputs[right + 1][0][self.lstm_dim:])
        return torch.cat([forward, backward])
```

Listing 2: Getting the span encoding

Defining our Neural Networks : We define two networks f_label and f_split as follows :

1. **f_label :** Takes as input the span encoding and gives an output which has the same size as the number of labels in our corpus. This gives scores for

each label, in order to determine which one can be assigned to the given span.

2. : **f_split :** Takes as input the span encoding and gives as output the score for the given span.

```
self.f_label = Feedforward(2 * lstm_dim, [label_hidden_dim],
    label_vocab.size)
self.f_split = Feedforward(2 * lstm_dim, [split_hidden_dim], 1)
```

Listing 3: Defining the networks

The helper function : This function is used as a means of getting the parse tree and loss for the entire sentence in a recursive, bottom up manner. This is done as follows :

1. **obtaining the label :** We pass the span encoding for the sentence to f_label. We take an argmax over this to get the most probable label.

```
    label_scores = self.f_label(get_span_encoding(left, right)
    )

```

Listing 4: Assigning Label Scores

2. **obtaining the label loss:** We then compute the label loss. This is assigned as the label score of the label with the maximum score.

```
    label_loss = label_scores[argmax_label_index]

```

Listing 5: Obtaining Label Loss

3. **obtaining the split scores :** In order to determine the split, we get left and right span encodings for each index in our given span. we then get the corresponding left and right scores and compute the split score as an addition of these.

```
    left_scores = self.f_split(torch.stack(left_encodings))
    right_scores = self.f_split(torch.stack(right_encodings))
    split_scores = left_scores + right_scores

```

Listing 6: Obtaining split scores

4. **obtaining the split :** An argmax is taken over all the split scores and the index at which it is maximum is assigned as our split.

```
    argmax_split_index = int(split_scores_np.argmax())

```

Listing 7: Getting split index

5. **obtaining the left and right subtrees :** We then recursively call the helper function for the left and right split to get the parse trees of the left and right subtrees, as well as their corresponding losses.

```
    left_trees, left_loss = helper(left, split)
    right_trees, right_loss = helper(split, right)

```

Listing 8: Obtaining left and right subtrees

6. **obtaining the parse tree :** We assign to the root of our parse tree the label learnt, and the left and right subtrees as it's children.

```
    children = left_trees + right_trees
    if label:
        children = [trees.InternalParseNode(label, children)]

```

Listing 9: Defining the parse tree

7. **return :** We finally return the above constructed tree and the loss as label_loss + split_loss + left_loss + right_loss

```
    return children, label_loss + split_loss + left_loss +
    right_loss

```

Listing 10: Return statement

3.2 evaluate.py

In order to score the predicted parse tree, we use a bracket scoring program called **Evalb**. We used PYEVALB, the python version of Evalb which is used to score the bracket tree banks. It reports precision, recall, F-measure, non crossing and tagging accuracy for given data. We save the predicted outputs as well as the gold trees for each tree in our batch. These are stored in two different files and then the output is written to another file. We finally return the **Bracketing Fscore**.

```
s = scorer.Scorer()
s.evalb(gold_path, predicted_path, output_path)
```

Listing 11: Getting bracketing Fscore

3.3 main.py

The functions for training and testing the model are called here. The run_train and run_test functions perform the following.

run_train

1. We first load the trees for training and development

2. Each tree is stored as a treeBankNode object, defined by us. The leaves are stored separately as leafTreeBankNode object. This is useful for defining base cases during our resursion
3. We then define a vocabulary for tags, words and labels.
4. We initialize our TopDownParser model using the arguments given, as well as the vocabularies.
5. The model is then trained in epochs. In each epoch, we evaluate the fscore 4 times (determined by the checks_per_epoch argument). In this process, we save the model with the best fscore.

run_test

1. We load the test trees from the path specified
2. We load the model from the path at which we had saved earlier.
3. We predict the parse tree for the sentence corresponding to every tree in the test bank as the gold tree.
4. We then compute the fscore for the predicted trees.

4 Experiments and Results

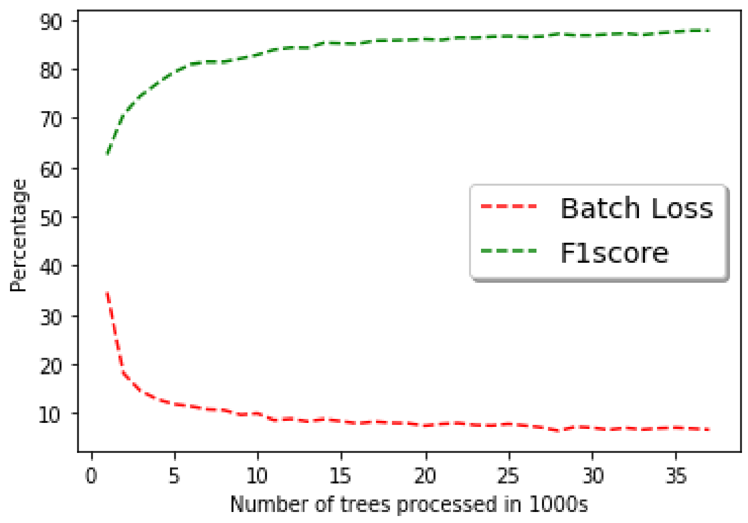
4.1 Model Parameters

The following are the dimensions of the parameters used for training the model:
Embedding dimensions:
Tag embedding dimension: 50
Word embedding dimension: 100
LSTM dimensions:
Number of layers in the LSTM: 2
Dimension of hidden layer of the LSTM: 250
Dropout: 0.4
Optimization: We have used the Adam optimizer for training with the above hyperparameters for 1 epoch with a total of about 39000 bracketed sentences.

4.2 Results

Precision - Precision is the ratio of correctly predicted positive observations to the total predicted positive observations.
Recall (Sensitivity) - Recall is the ratio of correctly predicted positive observations to the all observations in actual class.
F1 score - F1 Score is the weighted average of Precision and Recall.
F1 score is usually more useful than accuracy, especially if we have an uneven class distribution. Therefore, we have used F1 score as a measure to monitor the training procedure.

The figure below summarizes our training:



Number of sentence: 2416.00
Number of Error sentence: 0.00
Number of Skip sentence: 0.00
Number of Valid sentence: 2416.00
Bracketing Recall: 86.95
Bracketing Precision: 88.29
Bracketing FMeasure: 87.61
Complete match: 27.15
Average crossing: 1.06
No crossing: 59.69
Tagging accuracy: 100.00

5 References

- What's Going On in Neural Constituency Parsers? An Analysis (David Gaddy, Mitchell Stern, and Dan Klein)
- PyTorch Tutorials:
  `https://pytorch.org/tutorials/`
- Minimal Span Parser:
  `https://github.com/liucong3/minimal-span-parser/blob/master/src/main.py`