Notes on the different papers

Code Completion with Neural Attention and Pointer Networks

Introduction

Problem of the *hidden state bottleneck* making it impossible to deal with long-range dependencies.

We can use a form of attention to save previous state in a useful way. Even considering that we have the *unknown word problem*. This problem stems from the fact that when using the softmax models usually doesn't display the **unknown** words but only display a special word (*I.e:* UNK) which is useless to a programmer.

In example a variable name would usually be an unknown to token if it had to be suggested by classic RNN.

Pointer Networks are a way to solve this problem. They are a way to use the attention mechanism to point to a specific word in the input sequence. This way we can use the attention mechanism to point to the correct word in the input sequence. Those have the opposite problem in the fact that they cannot suggest word which aren't in the current sequence (*I.e:* they lack the global view that we need)

The paper proposes the **pointer mixture network** which is a mix of a **standard RNN** and a **pointer network**. They share the same RNN architecture.

A **switcher** is also learned to decide when to use a word which is **OoV** (out of view).

Program representation

The **corpus** is represented as an AST. Any programming language has a non-ambiguous context-free grammar which can be used to create the AST and also to do the inverse process.

Using this approach the number of possible **unique type** is relatively small

The AST is flattened to a sequence of nodes in the **depth-first** traversal. To make sure that the sequence can be converted back to the original tree structure to then go back to the source code we need to add additional information to the nodes (bits to understand if a node has a left or right **child**). We define a word $w_i = (T_i, V_i)$ to represent a node in which T_i is the type and V_i is the value. Starting from these the program is a sequence of words $w_{i=1}^{n}$.

We have **two tasks:**

- predicting next value
- predicting next type

Attention mechanism

Regarding the tokenization, for the attention mechanism in this case we consider that we want to use the **Parent attention**, which means that in an AST-based code we want the parent to be of great relevance for the child but we lose that information in our flattening. What we do is recording the parent location of each AST node.

This way a decision can be made using the information on the current hidden state as well as the context vector and the parent node.

Big code != Big Vocabulary: Open-Vocabulary Models for Source Code

Modeling vocabulary

A series of modeling choices were analyzed (Java language was used but similar results on python and c were indicated). The criteria used were:

- **Scalability**: influenced by vocabulary size (number of unique words) and corpus size (number of tokens)
- **Information loss**: models should be able to represent the original input as much as possible. OOV tokens are undesiderable. We build a vocabulary on the training set and compare it with the test set one, reporting the percentage of new vocabulary words seen in the test set.
- **Word frequency**: rare words usually have worse representation than frequent ones. This means that increasing word frequency is desirable and different modelling choices can increase and decrease the number of rare words.

Baseline: the baseline model is (11.6M 100%, 2.43B 100%, 42% *79%*, 83%). These are unsplit tokens, except for strings and comments.

Basic filtering

- **Non-english** words are removed ("<non-en>")
- **whitespaces** are filtered
- **Comments** are removed ("<comment>")
- **Strings** are removed ("<string>")

New baseline: (10.9M 94%, 1.15B 47%, 39% *80%*, 84%).

Word splitting

The words will be split based on **camelCase** and **snake_case** notations in order to reduce the vocabulary size effectively.

Disadvantages

This increases the complexity of the model as source code is no longer only a sequence of tokens. Words need to be represented by a sequence of subtokens (*Note:* n-grams are therefore unviable.)

Advantages

This has also many advantages, decreasing the number of OOV and allowing relationships between subtokens. This could also allow to generate neologism.

Splitting: (1,27M 12%, 1.81B 157%, 8% *20%*, 81%)

Note: Removing case was tried but proved to be ineffective.

Subword splitting

First of all we split the number into a **sequence of digits**.
Numbers: (795K 63%, 1.85B 102%, 6% *18%*, 72%)

Then we apply **Spiral token splitter**.
Spiral: (476K 37%, 1.89B 104%, 3% *9%*, 70%)

Note: stemming and character models were explored but didn't produce great result.

BPE

Byte Pair Encoding is an algorithm originally designed for data compression. In this case we use the approach to build NMT vocabularies: the most frequently occuring sequences of characters are merged to form a new vocabulary words.

The special subword "</t>" is added to the vocabulary so that we are able to convert the sequence of words back to the original tokens. So the vocabulary now is made out all the characters and the end-of-token character and the corpus is split into characters and end-of-token character.

From here, all the symbols pair in the vocabulary are counted and all the appearances of the most frequent pair are replaced with the new symbol. This is done iteratively until the vocabulary reaches a certain size (*Note:* we don't remove the single characters in the other places in which they appear). This operation is called **merge**

Advantages

No OOV tokens, as the vocabulary contains all the characters. At the same time the most common words are still represented by a single token (*like* exception). We can also manage how big we want the vocabulary by choosing how many times we want to perform the merge operation.

We apply this method for 10k times. The corpus size is big but not too big in respect to the other option explored. More than that, given the process that has been followed, the number of words that appear less than 10 times is only 1%.
BPE Subwords: (10K 1%, 1.57B 137%, 0% *0%*, 1%)

Note: we could try to generate other vocabularies and corpus using different iterations of the merge operation.

A thing that we should note is that the token generate **are not human readable** but looking at them we can recognize some simple pattern that we can understand. (*I.e:* some words are common words in code, stem of words, part of acronyms etc).

Predicting the tokens starting from the subwords

The standard model used, would normally predict subwords instead of tokens. What we need to do is merge multiple subwords in order to get the token. Once we have that we are left with the task of **ranking** the tokens.

To handle this we do the **beam search**. This kind of search operates on tokens made out of lists of subwords that all **must end** with the end-of-token character. The beam search is used to find the most likely token.

The algorithm starts with a token size k and a beam size b. It uses two priority queues:

- **candidates**: the tokens (sequence of subwords) that still have to be analyzed. The candidates have the **Text** (concatenation of subwords) and **prob** (product of the probabilities) fields.
- **bestToken**: which are the k highest probability tokens among the ones analyzed

Both the probability queue are sorted by the probability of the candidate.

At each iteration the algorithm pops the b best candidates from the candidates queue and expands them with one more subword. If this operation creates a new token (which means that the end-of-token was picked) then it's pushed inside the bestToken queue, pushing out the worst token of the queue, otherwise it's pushed again inside the candidates queue.