Write-Up R252

Learning Code Suggestion with a Sparse Pointer Network (SPN)

Samuel Müller (sgm48)

March 13, 2019

1 Introduction

This term I reimplemented Sparse Pointer Networks (SPN) (Bhoopchand et al. 2016) on learning a language model (LM) with copy functionality for Python code, adapted it to work with Java and improved upon it by making better usage of its internal representations and by feeding the Java type of variables.

2 Reimplementation

2.1 SPN

In this section I will describe how to implement SPNs and in later sections I will describe how to add my extensions. An SPN can be thought of as having three parts. (i) An LSTM with a affine transformation (a standard fully connected layer without non-linearity) with a softmax that maps to the output vocabulary. (ii) A copy mechanism that is based on an attention distribution over the previous identifiers. (iii) Lastly, it has a mechanism to combine the prediction from (i) the LSTM and (ii) the copy mechanism. The copy mechanism has a memory of hidden states $M_t = [h_{ide_t(1)} \dots h_{ide_t(k)}] \in \mathbb{R}^{d \times k}$ of the LSTM at the positions of the previous k identifiers, here $ide_t(i)$ returns the ith previous identifier, that is used to find the right identifiers to copy. We compute a distribution α over the previous identifiers with

$$G_t = \tanh(W^M M_t + (W^h h_t) 1_k^T) \qquad \in \mathbb{R}^{d \times k} \qquad (1)$$

$$\alpha_t = \text{softmax}(w^T G_t) \qquad \in \mathbb{R}^{1 \times k} \qquad (2)$$

where $h_i \in \mathbb{R}^d$ for any i are the hidden states from the LSTM and t is the current time step, $W^M, W^h \in \mathbb{R}^{k \times k}$ as well as $w \in \mathbb{R}^k$ are trainable parameters and 1_k is the k-dimensional vector of only ones. This then yields a pseudo-sparse distribution i_t over the vocabulary V that is defined as

$$i_t[i] = \begin{cases} \alpha_t[j] & \text{if } m_t[j] = i, j \in \{1 \dots k\} \\ 0 & \text{otherwise} \end{cases} \qquad \in \mathbb{R} \tag{3}$$

where $m_t[j]$ is the index of the token fed in step $ide_t(j)$ to yield the hidden state $M_t[j]$ and $i \in \{1 \dots |V|\}$. The distribution i_t of the copy mechanism is now combined with the distribution y_t from the LSTM, that is defined as

$$y_t = \text{softmax}(W^V h_t + b^V) \qquad \in \mathbb{R}^{|V|} \tag{4}$$

where $W^V \in \mathbb{R}^{|V| \times d}$ and $b^V \in \mathbb{R}^{|V|}$ are trainable parameters, using a a weight λ_t. λ_t depends on the context c_t that is defined as $c_t = M_t \alpha_t^T$, the average of the hidden states in the memory weighted by α_t, and the representation of the last token fed to the LSTM x_t. Using these λ_t can be defined as

$$h_t^\lambda = [h_t; x_t; c_t] \qquad \in \mathbb{R}^{3d} \tag{5}$$

$$\lambda_t = \text{softmax}(W^\lambda h_t^\lambda + b^\lambda) \qquad \in \mathbb{R}^2 \tag{6}$$

where $W^\lambda \in \mathbb{R}^{2 \times 3d}$ and $b^\lambda \in \mathbb{R}^2$ are trainable parameters. With λ_t as weight at hand we can now define our final next-token distribution as the mixture of the copy distribution i_t and the LSTM prediction y_t defined as

$$y_t^* = [y_t, i_t] \lambda_t \qquad \in \mathbb{R}^{|V|}. \tag{7}$$

2.2 Baselines

2.2.1 LSTM with Attention (LMAtt)

This baseline uses most of the components defined for the SPN, but instead of using α_t as copy distribution, it only uses c_t to predict the next token as

$$n_t = \tanh\left(W^A \begin{bmatrix} h_t \\ c_t \end{bmatrix}\right) \qquad \in \mathbb{R}^d \tag{8}$$

$$y_t' = \text{softmax}(W^V n_t + b^V) \qquad \in \mathbb{R}^{|V|} \tag{9}$$

where $W^A \in \mathbb{R}^{d \times 2d}$, $W^V \in \mathbb{R}^{|V| \times d}$ and $b^V \in \mathbb{R}^{|V|}$ are trainable parameters. Another change to the model above for LMAtt is that M_t does not include the hidden states of the LSTM at the k previous identifiers, but instead holds the previous k hidden states of any tokens that is $M_t = [h_{t-k} \dots h_{t-1}]$.

2.2.2 LSTM

The LSTM baseline is to just use y_t as defined above to predict the next tokens. This is a very simple LM that does not use any previous state, but predicts the next character solely based on its current hidden state.

2.3 Implementation and Training Details

All tokens from code were presented to the LSTM as learned embeddings of size d and the initial hidden state of the LSTM is set to be all zeros. d was chosen to be 200. k was set to 30 for all models. Whole files were always fed to the models, so that we initialize the LSTM only at the beginning of each file. During training the gradient of every parameter element was clipped to be maximally 5 times the norm of the flattened vector of all parameter's elements as described by Pascanu et al. (2012). Also, dropout was applied on the inputs of the LSTM with a keep rate of 0.9 during training. Lastly, an SGD with an initial learning rate of 0.7 and a learning decay factor of 0.9 per epoch was used. All parameters were uniformly between -0.05 and 0.05 initialized. During training all copy probabilities were increased by 1×10^{-10}, so that the cross entropy, the log of the probability, is defined even for situations in which it is not possible to copy.

2.4 Dataset Anonymization

The dataset was normalized by replacing every identifier with a string indicating what kind of identifier it is (class, function, parameter, argument or local variable) and a random number between 1 and 3000 that is unique in the file.

3 Comparison

In this section I will speak about the differences between the implementation by Bhoopchand et al. (2016) and mine, as well as attention sharing, a model extension, and a type-specific dataset anonymization.

3.1 Implementation Differences

My implementation differs from the model described in Bhoopchand et al. (2016) in two main ways. (i) They removed comments and replaced numericals with a single *NUM* token, both of which I did not. Also, they replaced words with an out-of-vocabulary token if they appeared less than three times, while I used a threshold of 5000 on the total vocabulary size. Lastly, they normalized identifiers to be unique in their scope, while my normalization targeted file-level uniqueness, which for Java in most cases means class-level uniqueness. (ii) While Bhoopchand et al. (2016) describe that they use a pseudo softmax distribution for their copy distribution i_t, I used a similar distribution during training by feeding by adding a constant to the sum in the logarithm to compute the cross entropy over a batch, but I did not use such a constant at inference time, and thus allowed a copy distribution of all zeros. The only hyper parameter that differs between the models, is that I was a able to use a k of 30 for all my experiments, while Bhoopchand et al. (2016) used a k of 20.

3.2 Extensions

3.2.1 Incorporating Types

For Python identifiers it is non-trivial to determine their type statically, since Python has a dynamic typing system and so variables might change type over time and it is hard to find out what the initial type is, since it can be multiple layers deep hidden. In Java on the other hand we have a static typing system and so we can use types to make better predictions of the next word. In incorporated types not into my model directly, but left the model as general as possible, and instead changed the normalization approach to represent types. The normalization approach, as presented by Bhoopchand et al. (2016), normalizes variables without taking into account of what type these variables are by replacing an argument of type *String* for example with the token *Argument123*. My typed approach to normalization on the other hand is to replace the argument in the example with the token *Argument-String123*. In general the typed normalization approach replaces all identifiers, besides class names and function names, with an anonymous identifier that is chosen randomly but restricted to their type. I treated all built-in types, as a single type, and type parameters were ignored for generic types. Both of these design decisions were made so that the number of different types does not explode, since a larger number of types also means that the normalized data has a larger vocabulary. To decrease the vocabulary size even further the range from which the appended number is drawn is chosen to be the minimal range for each type that allows all normalized identifiers to be unique in their files.

3.2.2 Attention Sharing

As described in section 2 the SPN already has a distribution over previous hidden states of the LSTM and also already computes the weighted average c_t, so the logical next step would be to use c_t for the next-token prediction of the LSTM, as See et al. (2017) did in their model for abstractive summarization. This can easily be done by changing the definition of y_t^* in the SPN to not be a mixture of y_t and i_t, but of y_t', defined in section 2.2.1, and i_t. That is equivalent to replacing equation 7 with

$$y_t^* = [y_t', i_t]\lambda_t \qquad \in \mathbb{R}^{|V|}. \tag{10}$$

For a more visual explanation of the models described, including the attention sharing, can be found in figure 1. This model can be used with the attention being restricted to identifiers or without this restriction. While further theoretically should be helpful for copying, which is the major idea for the work by Bhoopchand et al. (2016), latter is likely to be more helpful to the attention mechanism. Overall, as shown in section 4.2 TODO performs better.

4 Evaluation

4.1 SPN Performance in Comparison to Baselines

Bhoopchand et al. (2016) compare the SPN with the LMAtt and the LSTM in terms of its perplexity and accuracy. In their results the LMAtt with an attention window of 50 steps, ours has 30, has the highest accuracy, the SPN has the lowest perplexity and the LSTM performs worst in both metrics. These relative results could be reproduced on the test set in this project. The exact results can be found in table 4.1, though, show that in our experiments the SPN did not perform as well as for Bhoopchand et al. (2016) compared to the baselines: While I observed a 4.34 % higher accuracy of LMAtt relative to the SPN's accuracy, they only see an improvement of 0.38% and the improvement upon the LSTM's perplexity they observe is of 31.00%, I could only observe an improvement of 2.85%. This might either be due to the fact that the data worked with is not very comparable, they work in with Python code and I work with Java code, but it could also be due to the fact that they choose their data to support their thesis that the SPN is a good model. The second claim, as unlikely as it might be, is supported by the fact that Bhoopchand et al. (2016) did not use a standard

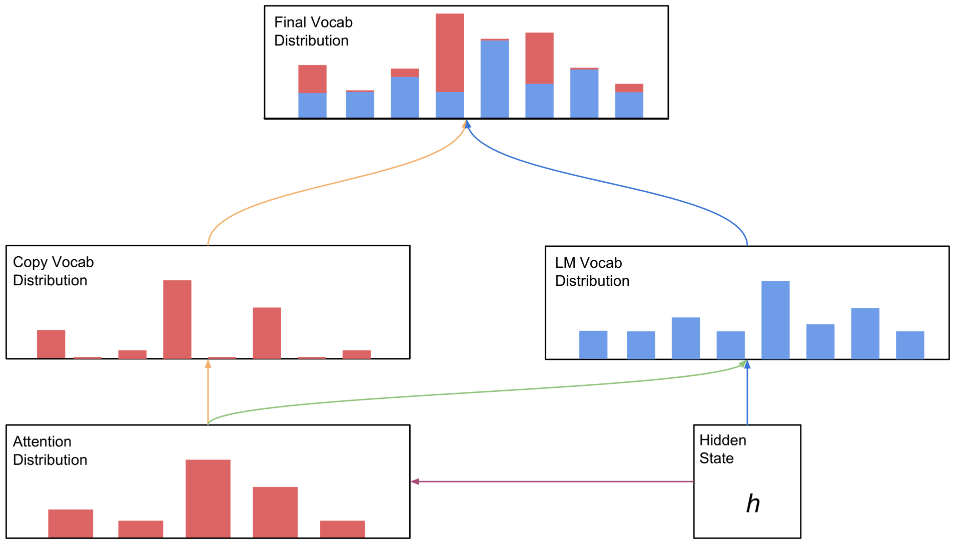

Figure 1: A schematic drawing of the dependencies between different components of the models described. Dependencies of the LSTM: blue, LMAtt: blue + magenta + green, SPN: blue + magenta + orange, and for the SPN with shared attention: all colors.

	LSTM	LMAtt	SPN
Test Perplexity	39.58	116.44	38.45
Test Accuracy	60.43 %	63.64 %	60.99 %

Table 1: This table shows the performance of the SPN and the baselines used by Bhoopchand et al. (2016), i.e. a LMAtt and a LSTM.

Metric \ Model	SPN	Attention Sharing
Test Perplexity	38.45	28.73
Test Accuracy	60.99 %	62.63 %

Table 2: This table shows the performance of the SPN with attention sharing to the SPN.

dataset but instead build one themselves. A third possibility might be that my implementation of the SPN differs from theirs.

4.2 Can the SPN be improved by sharing the attention weights between the copy mechanism and the LSTM?

While the SPN is a powerful model, it does not make use of the attention weights computed to generate better next-token predictions using the LSTM, therefore the thesis for this experiment is that the SPN model and the LMAtt model can be improved by merging them, as described in section 3.2.2. While this definitely makes the model more powerful theoretically, due to a new connection, it also increases the number of parameters. The size increasement on the other hand is not to big, since we only introduce one new layer to the SPN which has 80'000 weights, equivalent to 2.4 % of the parameters of the SPN.
Sharing attention improves the SPN clearly, as can be seen in 4.2, and it does so without restricting its attention to identifiers, the main novelty in Bhoopchand et al. (2016). It does, on the other hand, only outperform the LMAtt in perplexity.

4.3 The Effects of a Typed Corpus

In section 3.2.1 a method is introduced that does not change the model but still lets it depend more directly on types of identifiers. This is done by changing the approach to identifier normalization.
Even though this is a very simple method to incorporate types into the model, it is still expected that the accuracy increases when training the same

Metric \ Corpus	Non-Typed	Typed
Test Accuracy	60.99 %	62.63 %

Table 3: This table compares the performance of the SPN trained on the non-typed corpus, described in section 2.4, and trained on the typed corpus, described in section 3.2.1.

model on the typed dataset. It is only fair to compare accuracy here, since this metric, unlike perplexity, cannot be artificially be increased by increasing the number of unknown tokens in the dataset. To compare I trained the SPN until convergence on the typed corpus and compared its performance with the performance of the SPN trained on the non-typed corpus. You can find the results in table 3. The accuracy could be improved by adding types and the frequency of out-of-vocabulary tokens the model sees could be decreased from 12.4% on the non-typed corpus to 10.3%, even though the number of tokens in the corpus was slightly increased from 1.10×10^5 to 1.31×10^5.

4.4 Identifier Accuracy

4.5 Qualitative Analysis of Example Predictions

As an example to analyze I chose a sequence a two line sequence, that refers to the same identifier twice, randomly from the test corpus. Below the example to be analyzed can be seen:

```
localvariable91 = new hashmap<>();
localvariable91.put(<unk>, ...
```

In this code snippet a previously defined variable is reassigned to a newly initialized hashmap and in the next line a map is added to that hashmap, where the maps key is an out-of-vocabulary token and the value is omitted. We are going to analyze teacher-forced predictions of an LSTM, an SPN, an SPN with attention sharing and an SPN on the typed corpus. Gray tokens were predicted with less than 75% confidence. The LSTM predicts the following:

```
assertthat = new <unk><>();
for.<unk>(<unk>, ...
```

While the SPN predicts:

```
assertthat = <unk> <unk><>();
<unk> = put(<unk>, ...
```

Both predict the last tokens of the initialization of the hashmap well, which are a common pattern in Java. While the LSTM correctly predicts that we want to call a member function of the above defined variable, the SPN goes with the very unintuitive prediction of a reassignment of the just previously defined variable. The SPN seems to have some understanding of the type of the variable, since it correctly predicts what member function is called.
The SPN can be improved with attention sharing, as can be seen in the prediction of an SPN with attention sharing below:

```
assertthat = <unk> <unk><>();
localvariable91.put(<unk>, ...
```

It does not only have an idea of the type of the variable, since it correctly predicts the member function called in the second line, but it also predicts that we want to call a member-function of the previously defined variable. This can be even further improved by making the types of variables more accessible to the SPN. The same example from above has the following form in the typed corpus:

```
localvariable-map41 = new hashmap<>();
localvariable-map41.put(<unk>, ...
```

The SPN trained on this dataset predicts the following:

```
assertthat = new hashmap<>();
localvariable-map41.put(<unk>, ...
```

It predicts the whole sequence, besides the first token, but it has a lot of tokens with low confidence.
Generally these predictions exemplify the strengths of the extensions proposed in this paper, to incorporate types and to share the attention weights in the SPN.

5 Reflection

5.1 Getting Started

To get to the first implementation of the SPN and the LMAtt was quite quick, since Bhoopchand et al. (2016) published their code online and I

could reuse it after adapting it to fit the python and tensorflow version that I was using.
After that though there was a long progress plateau, due to slight differences in my implementation compared to theirs.

Batching First, it did make a difference that I did not feed the model whole files during training but only extracts of 100 tokens. It was not easy to feed longer sequences because of the memory limitations on GPUs, that is why I need to change the batching mechanism to compose batches of files with similar lengths and then feed batches of 100 tokens from each file to the model at each step. This way I was able to train on very long consecutive sequences without a too big an efficiency loss.

Settings Second, my settings of the hyper-parameters and my architecture were slightly different. On the one hand, I computed the attention weights α_t not over the hidden state of the LSTM, but instead over the token embeddings. In hindsight it makes a lot of sense that this makes a significant difference, since the embeddings of the normalized identifiers should not give away much more than the obvious information that this is an identifier of some type, while the hidden state also entails the usage and possibly the circumstances of its definition, e.g. if it is a loop variable. On the other hand, I used a too small memory size k.

These problems were intensified by the fact, that training the model on the full dataset does take very long. While the improvement becomes very small, the model does not fully converge even after two days of training.

5.2 Extensions

After having a version of the SPN and the LMAtt that clearly outperform the LSTM, I could begin working on the extensions detailed in section 3.2. While the attention detailed above were working in only a few iterations, since the model and the normalization script were easily extensible, there was another extension I put time and thought into that did not produce the results I hoped for. This extension should teach the model to copy more, by masking the LSTM prediction during training whenever it was possible to copy the correct next token.

6 Future Work

The typed normalization is well-defined for Java versions prior to Java 10, but Java 10 introduces type inference, which yields situations in which a variable token appears before it's type definition and thus the language model on the typed corpus would suffer an information leakage. This should be addressed so that the language model's objective is as close as possible to next token predictions in editors.

References

Bhoopchand, A., Rocktäschel, T., Barr, E. & Riedel, S. (2016), `Learning python code suggestion with a sparse pointer network', *arXiv preprint arXiv:1611.08307* .

Pascanu, R., Mikolov, T. & Bengio, Y. (2012), `On the difficulty of training recurrent neural networks'.

See, A., Liu, P. J. & Manning, C. D. (2017), `Get to the point: Summarization with pointer-generator networks', *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* .
URL: *http://dx.doi.org/10.18653/v1/P17-1099*