Code Completion with Neural Attention and Pointer

Anonymous Authors

Abstract

Intelligent code completion has become an essential tool to accelerate modern software development. To facilitate effective code completion for dynamically-typed programming languages, we apply neural language models by learning from large codebases, and investigate the effectiveness of attention mechanism on the code completion task which has not been explored in prior work. However, standard neural language models even with attention mechanism cannot correctly predict out-of-vocabulary (OoV) words thus restrict the code completion performance. In this paper, inspired by the prevalence of locally repeated terms in program source code, and the recently proposed pointer network which uses attention to reproduce a word from local context, we propose a pointer mixture network for better predicting OoV words in code completion. Based on the context, the pointer mixture network learns to either generate a within-vocabulary word through an RNN component, or copy an OoV word from local context through a pointer component. Experiments on two benchmarked datasets demonstrate the effectiveness of our attention mechanism and pointer mixture network on the code completion task.

Introduction

Integrated development environments (IDEs) have become essential tools for modern software engineers, as IDEs provide a set of helpful services to accelerate the software development process. Intelligent code completion is one of the most useful features in IDEs, which suggests next probable code tokens such as method names or object fields, based on existing code in the context. Traditionally, code completion relies on type checking and static analysis of libraries (Han, Wallace, and Miller 2009). Thus, it works well for statically-typed languages such as Java. Yet code completion is harder and less supported for dynamically-typed languages like JavaScript and Python, due to the lack of type annotations.

To render effective code completion for dynamically-typed languages, recently, researchers turn to learning-based language models (Hindle et al. 2012; White et al. 2015; Bielik, Raychev, and Vechev 2016). They treat programming languages as natural languages, and train code completion systems by learning from large codebases (e.g., GitHub). In particular, neural language models such as Recurrent Neural Networks (RNNs) can capture sequential distributions and deep semantics, hence are very popular. However, these standard neural language models are limited by the so-called *hidden state bottleneck*: all the information about current sequence is compressed into a fixed-size vector. The limitation makes it hard for RNNs to deal with long-range dependencies, which are common in program source code such as a class identifier declared many lines before it is used.

Attention mechanism (Bahdanau, Cho, and Bengio 2014) provides one solution to this challenge. With attention, neural language models learn to retrieve and make use of relevant previous hidden states, thereby increasing the model's memorization capability and providing more paths for gradients to back propagate. However, attention mechanism on the code completion task has not been explored in prior work. To investigate its effectiveness, in this paper, we develop an attention mechanism for neural network based code completion.

But even with attention, there is another issue called *unknown word problem*. In general, the last component of neural language models is a softmax classifier, with each output dimension corresponding to a unique word in the predefined vocabulary. As computing high-dimensional softmax is computational expensive, a common practice is to build the vocabulary with only *K* most frequent words in the corpus and replace other out-of-vocabulary (OoV) words with a special word, i.e. *UNK*. Intuitively, standard softmax based neural language models cannot correctly predict OoV words. In code completion, simply recommending an *UNK* offers no help to developers. The *unknown word problem* restricts the performance of neural language models especially when there are a large number of unique words in the corpus like program source code.

Back to our code completion task, we observe that when writing programs, developers tend to repeat locally. For example, the variable name `my_salary` in Figure 1 may be rare and marked as *UNK* with respect to the whole corpus. But within that specific code block, it repeats several times and has a relatively high frequency. Intuitively, when predicting such unknown words, our model can *learn to choose* one location in local context and then copy the word at that location to be our prediction. Actually, the recently proposed Pointer Networks (Vinyals, Fortunato, and Jaitly 2015) are

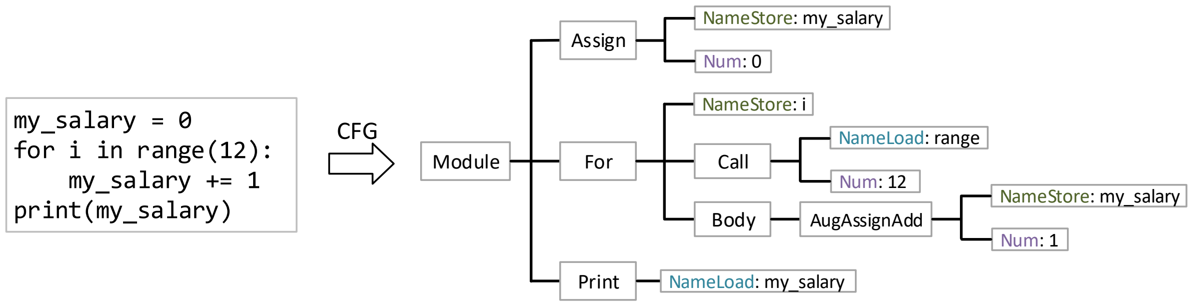
```
my_salary = 0
for i in range(12):
    my_salary += 1
print(my_salary)
```

Figure 1: A Python program and its corresponding abstract syntax tree (AST).

able to do so, which use attention scores to select a word from the input sequence as output. Although pointer networks can make better predictions on unknown words or rare words, they are unable to predict words beyond current input sequence, i.e., lacking the global view. Therefore they may not work well in our code completion task.

In this paper, to facilitate effective code completion, we propose a *pointer mixture network*, which can predict next word by either generating one from the global vocabulary or copying a word from local context. For the former, we apply a standard RNN with attention, which we call the *global RNN component*. For the latter, we employ a pointer network with a fixed-size attention window over previous inputs, which we call the *local pointer component*. Actually the two components share the same RNN architecture and attention scores. Our pointer mixture network is a weighted combination of the two components. At each prediction, a controller is learned based on the context information, which can guide our pointer mixture network to choose one component for generating the next word. In this way, our model learns *when and where* to copy an OoV word from local context as the final prediction.

The main contributions of this work are as follows:

- We propose a pointer mixture network for better predicting OoV words in code completion, which generates new words from either the global vocabulary or local context.
- We demonstrate the effectiveness of attention on code completion, which has not been explored in prior work.
- We evaluate our methods on two benchmarked datasets (JavaScript and Python) for code completion. The experimental results show significant improvements upon the state-of-the-arts.

Approach

Program Representation

In our corpus, each program is represented in the form of *abstract syntax tree* (AST). Regardless of whether it is dynamically-typed or statically-typed, any programming language has an unambiguous context-free grammar (CFG), which can be used to parse source code into an AST. Further, the AST can be converted back into source code in a one-to-one correspondence. Processing programs in the form of ASTs is a typical practice in *Software Engineering* (SE) (Mou et al. 2016; Li et al. 2017).

Figure 1 shows an example Python program and its corresponding AST. We can see that each AST node contains two attributes: the type of the node and an optional value. For each leaf node, ":" is used as a separator between type and value. For each non-leaf node, we append a special `EMPTY` token as its value. As an example consider the AST node `NameLoad:my_salary` in Figure 1 where `NameLoad` denotes the type and `my_salary` is the value. The number of unique types is relative small (hundreds in our corpus), with types encoding the program structure, e.g., `Identifier`, `IfStatement`, `SwitchStatement`, etc. Whereas there are infinite possibilities for values, which encode the program text. A value may be any program identifier (e.g. `jQuery`), literal (e.g. `66`), program operator (e.g., `+`, `-`, `*`), etc.

Representing programs as ASTs rather than plain text enables us to predict the structure of the program, i.e. type of each AST node. See the example in Figure 1 again, when the next token is a keyword `for`, the corresponding next AST node is `For(:EMPTY)`, which corresponds to the following code block:

```
for ___ in ___:
  ## for loop body
```

In this way, successfully predicting next AST node completes not only the next token `for`, but also the whole code block including `in` and "`:`". Such structure completion enables more intelligent code completion.

To apply statistical sequence models, we serialize each AST as a sequence of nodes in the in-order depth-first traversal. To make sure the sequence can be converted back to the original AST thus converted back to the source code, we allow each node type to encode two additional bits of information about whether the AST node has a child and/or a right sibling. If we define a word as $w_i = (T_i, V_i)$ to represent an AST node, with T_i being the type and V_i being the value, then each program can be denoted as a sequence of words $w_{i=1}^n$. Thus our code completion problem is defined as: given a sequence of words $w_1, ..., w_{t-1}$, our task is to predict the next word w_t. Obviously, we have two kinds of tasks: predicting the next node type T_t and predicting the next node value V_t. We build one model for each task and train them separately. We call this AST-based code completion.

Neural Language Model

The task of code completion can be approached by a language model that estimates a probability distribution over sequences of words. Based on the estimated probability, we can make a prediction on the next word. In a formal definition, we parameterize the language model by θ and formulate our code completion task as:

$$p(w_t|w_1, w_2, ..., w_{t-1}; \theta). \quad (1)$$

Recently recurrent neural networks (RNNs) have achieved appealing success in language modeling, however, they suffer from the gradients vanishing/exploding problem (Bengio, Simard, and Frasconi 1994). LSTM (a variant of RNNs) (Hochreiter and Schmidhuber 1997) is proposed to mitigate this problem, by keeping an internal memory s_t for each cell and utilizing gating mechanisms to update it. LSTM has been proven to outperform vanilla RNNs on various tasks, including language modeling (Sundermeyer, Schlüter, and Ney 2012). So we choose LSTM as our base model for code completion.

A standard LSTM cell is defined as $(h_t, s_t) = f(x_t, h_{t-1}, s_{t-1})$. At each timestep t, an LSTM cell takes current input vector x_t and previous hidden state h_{t-1} and internal memory s_{t-1} as inputs, then produces the current hidden state h_t and internal memory s_t via the following calculations:

$$\begin{bmatrix} i_t \\ f_t \\ o_t \\ \tilde{s}_t \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} W_{4k,2k} \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix} \quad (2)$$

$$s_t = f_t \odot s_{t-1} + i_t \odot \tilde{s}_t \quad (3)$$

$$h_t = o_t \odot \tanh(s_t) \quad (4)$$

Here, $W_{4k,2k}$ denotes a $4k * 2k$ trainable weight matrix, and k is the size of the hidden state, i.e. dimension of h_t. Furthermore, $i_t, f_t, o_t, \tilde{s}_t, x_t, h_t, s_t \in \mathbb{R}^k$, among which i_t, f_t and o_t operate as three gate functions to control the information flow inside LSTM cells. σ and $\odot$ represent sigmoid activation function and element-wise multiplication respectively.

Attention Mechanism

Although LSTM is better at capturing long-range dependencies than vanilla RNNs, it still suffers from *hidden state bottleneck* that attempts to carry the whole sequence information in a fixed-size vector. To alleviate this problem, attention mechanism is proposed and incorporated into standard neural language models like LSTM, which we call attention-enhanced LSTM in this work. With attention, neural language models learn to retrieve and make use of relevant previous hidden states, thereby increasing the model's memorization capability and providing more paths for gradients to back propagate. However, attention mechanism has not been applied in our AST-based code completion task.

An attention-enhanced LSTM is illustrated in Figure 2. Formally, to use attention mechanism, we keep an external memory of L previous hidden states, which is denoted as $M_t = [h_{t-L}, ..., h_{t-1}] \in \mathbb{R}^{k*L}$. At timestep t, the model uses an attention layer to compute the relation between h_t

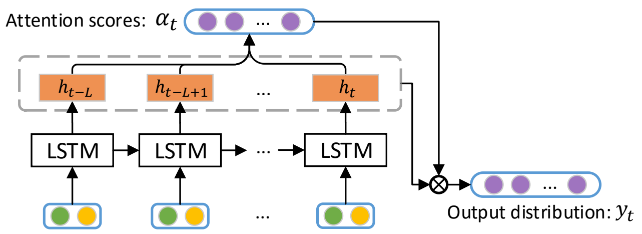

Figure 2: The attention-enhanced LSTM. The inputs fed to each LSTM cell is composed of two kinds of embeddings (green for Type and yellow for Value). $\otimes$ represents the element-wise multiplication. Here we neglect the concatenation of c_t and h_t for simplicity.

and hidden states in M_t, represented as attention scores α_t, and then produces a summary context vector c_t. As the original attention mechanism (Bahdanau, Cho, and Bengio 2014) is used in sequence-to-sequence models, we adapt it into our sequence model and design an attention mechanism for code completion as follows:

$$A_t = v^T \tanh(W^m M_t + (W^h h_t)1_L^T) \quad (5)$$

$$\alpha_t = softmax(A_t) \quad (6)$$

$$c_t = M_t \alpha_t^T \quad (7)$$

where $W^m, W^h \in \mathbb{R}^{k*k}$ and $v \in \mathbb{R}^k$ are trainable parameters. 1_L represents a L-dimensional vector of ones.

When predicting next word at timestep t, we condition the decision on not only the current hidden state h_t but also the context vector c_t. The output vector G_t encodes the information about next token and is then projected into the vocabulary space, followed by a softmax function to produce the final probability distribution $y_t \in \mathbb{R}^V$:

$$G_t = \tanh(W^g[h_t; c_t]) \quad (8)$$

$$y_t = softmax(W^v G_t + b^v) \quad (9)$$

where $W^g \in \mathbb{R}^{k*2k}$ and $W^v \in \mathbb{R}^{V*k}$ are two trainable projection matrix and $b^v \in \mathbb{R}^V$ is a trainable bias vector. Note that V represents the size of vocabulary and ";" denotes the concatenation operation.

Pointer Mixture Network

Rare tokens are prevalent in program source code as programming habits vary from person to person, resulting in a large number of unique tokens in a code corpus. To render efficient code completion, we build the token vocabulary with only *K* most frequent tokens, and replace all OoV tokens with a special token *UNK*. OoV tokens cannot be correctly predicted out since they are not eligible outputs. Considering that locally repeated tokens are also prevalent in program source code, we can leverage the pointer networks to predict OoV tokens, by copying a token from previous input sequence. However, pointer networks fail to predict tokens that are not present in the input. We propose to resolve this by using a *pointer mixture network* that combines a standard RNN and a pointer network, as shown in Figure 3.

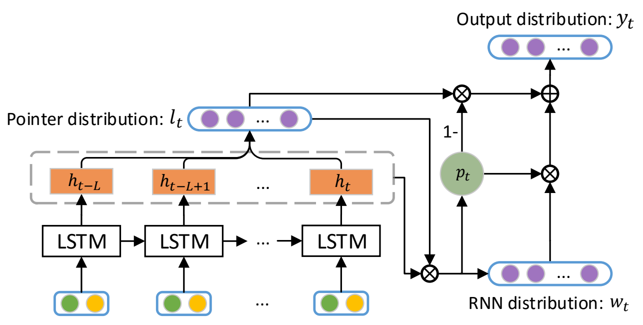

Figure 3: The pointer mixture network. We reuse the attention scores α_t (see Figure 2) as the pointer distribution l_t. The controller produces $p_t \in [0, 1]$ to balance two distributions, i.e. l_t and w_t, based on the context. The final distribution is generated by concatenating two scaled distributions and applying softmax function. Here $\oplus$ indicates the concatenation operation.

Our pointer mixture network consists of two major components (global RNN component and local pointer component) and one controller to strike a balance between them. For global RNN component, it is an attention-enhanced L-STM that predicts the next token from a predefined global vocabulary. For local pointer component, it points to local locations according to the location weights, which can be obtained by reusing the attention scores from global RN-N component. Our pointer mixture network combines the two components by concatenating the two components' output vectors. Before concatenation, the two individual outputs are scaled by a learned controller based on the context, thus our model learns how to choose certain component at each prediction. Specifically, the controller produces a scalar $p_t \in [0, 1]$ which indicates the probability to use the global RNN component, and then $1 - p_t$ is the probability to use the local pointer network.

Then we calculate the concatenated softmax and pick one output dimension with the highest probability. If this dimension belongs to the RNN component, then the next token is generated from the global vocabulary. Otherwise, the next token is copied from local context.

Formally, at timestep t, the global RNN component produces probability distribution $w_t \in \mathbb{R}^V$ for next token x_t within the vocabulary according to formula 9. The local pointer component points to the locations inside a memory according to the distribution $l_t \in \mathbb{R}^L$, where L is the length of the memory. In order to reduce the parameters and accelerate the training, we reuse the attention scores (see formula 6) as l_t in practice.

To arrive at the final prediction from w_t and l_t, we propose a controller to indicate which component to be chosen based on the context, which is a sigmoid function conditioned on the current hidden state h_t and context vector c_t:

$$p_t = \sigma(W^p[h_t; c_t] + b^p) \tag{10}$$

where $W^p \in \mathbb{R}^{2k*1}$ and $b^p \in \mathbb{R}^1$ are trainable weights. $p_t \in [0, 1]$ is a scalar to balance w_t and l_t. Finally, the model completes by concatenating the two scaled distributions and applying softmax to produce the final prediction:

$$y_t = softmax([p_t w_t; (1 - p_t) l_t]) \tag{11}$$

Table 1: Dataset Statistics

	JS	PY
Training Queries	$10.7 * 10^7$	$6.2 * 10^7$
Test Queries	$5.3 * 10^7$	$3.0 * 10^7$
Type Vocabulary	95	329
Value Vocabulary	$2.6 * 10^6$	$3.4 * 10^6$

Evaluation

Dataset

We evaluate different approaches on two benchmarked datasets: one is JavaScript language (JS) and another is Python language (PY), which are summarized in Table 1. Collected from GitHub, both two datasets are publicly available[1] and used in previous work (Bielik, Raychev, and Vechev 2016; Raychev, Bielik, and Vechev 2016; Liu et al. 2016). Both datasets contain 150,000 program files which are stored in their corresponding AST formats, with the first 100,000 used for training and the last 50,000 used for testing. After serializing each AST in the in-order depth-first traversal, we generate multiple *queries* used for training and evaluation, one per AST node, by removing the node (plus all the nodes to the right) from the sequence and then attempting to predict back the node.

The numbers of unique node types in JS and PY are 44 and 181 originally, by adding information about children and siblings as discussed before, we increase the numbers to 95 and 329. As shown in Table 1, the number of unique node values in both datasets are too large to apply neural language models, thus we only choose K most frequent values in each training set to build the global vocabulary, where K is a free parameter. We further add three special values: `UNK` for out-of-vocabulary values, `EOF` indicating the end of each program, and `EMPTY` being the value of non-leaf AST nodes.

Experimental Setup

Our base model is a single layer LSTM network with unrolling length of 50, hidden unit size of 1500, and forget gate biases initialized to 1. To train the model, we use the cross entropy loss function and mini-batch SGD with the Adam optimizer (Kingma and Ba 2014). We set the initial learning rate as 0.001 and decay it by multiplying 0.6 after every epoch. We clip the gradients' norm to 5 to prevent gradients exploding. The batch size is 128 and we train our model for 8 epochs.

We divide each program into segments consisting of 50 consecutive tokens, with the last segment being padded with `EOF` tokens if it is not full. The LSTM hidden state and memory state are initialized with h_0, c_0, which are two trainable vectors. The last hidden and memory states from the previous LSTM segment are fed into the next one as initial states

[1]http://plml.ethz.ch

Table 2: Accuracies on next value prediction with different vocabulary sizes. The out-of-vocabulary (OoV) rate denotes the percentage of AST nodes whose value is beyond the global vocabulary.

	JS			PY		
Vocabulary Size (OoV Rate)	1k (20%)	10k (11%)	50k (7%)	1k (24%)	10k (16%)	50k (11%)
Vanilla LSTM	69.9%	75.8%	78.6%	63.6%	66.3%	67.3%
Attention-enhanced LSTM (ours)	71.7%	78.1%	80.6%	64.9%	68.4%	69.8%
Pointer Mixture Network (ours)	**73.2%**	**78.9%**	**81.0%**	**66.4%**	**68.9%**	**70.1%**

if both segments belong to the same program. Otherwise, the hidden and memory states are reset to h_0, c_0. We initialize h_0, c_0 to be all-zero vectors while all other variables are randomly initialized using a uniform distribution over the interval [-0.05, 0.05].

As each AST node consists of a type and a value, to encode the node and input it to the LSTM, we train an embedding vector for each type (300 dimensions) and value (1200 dimensions) respectively, then concatenate the two embeddings into one vector. Since the number of unique types is relatively small in both datasets, there is no *unknown word problem* when predicting next AST node type. Therefore, we only apply our *pointer mixture network* on predicting next AST node value. When using the attention-enhanced LSTM and pointer mixture network, we set the attention window length as 50, so the model learns to make use of last 50 hidden states at each timestep.

For each dataset, we build the global vocabulary for AST node values with K most frequent values in the training set, and mark all out-of-vocabulary node values in training set and test set as unknown values. Before training, if an unknown value appears exactly the same as another previous value within the attention window, then we label that unknown value as the corresponding position in the attention window. Otherwise, the unknown value is labeled as `UNK`. If there are multiple matches in the attention window, we choose the position label as the last occurrence of the matching value in the attention window, which is the closest one. For within-vocabulary values, we label them as the corresponding IDs in the global vocabulary. During training, whenever the ground truth of a training query is `UNK` we set the loss function to zero for that query such that our model does not learn to predict `UNK` tokens. In both training and evaluation, all predictions where the target word is `UNK` are treated as wrong predictions. We employ *accuracy* as our evaluation metric, i.e. the proportion of correctly predicted next node types/values.

We implement our models using Tensorflow (Abadi et al. 2016) and run our experiments on a Linux server with one NVIDIA GTX TITAN GPU. Unless otherwise stated, each experiment is run for three times and the average result is reported.

Experimental Results

For each experiment, we run the following models for comparison, which have been introduced in the APPROACH section:

- **Vanilla LSTM**: A standard LSTM network without any attention or pointer mechanisms.
- **Attention-enhanced LSTM**: An LSTM network equipped with our attention mechanism which attends to last 50 hidden states at each timestep. We are the first to use attention mechanism for AST-based code completion.
- **Pointer Mixture Network**: Our proposed mixture network which combines the above attention-enhanced LSTM and the pointer network.

OoV Prediction We first evaluate our pointer mixture network's ability to ease the *unknown word problem* when predicting next AST node value. For each of the two datasets, we create three specific datasets by varying the global vocabulary size K for node values to be 1k, 10k, and 50k, resulting in different out-of-vocabulary (OoV) rates. We run the aforementioned models on each specific dataset. Table 2 lists the corresponding statistics and experimental results.

As Table 2 shows in each column, on each specific dataset, the vanilla LSTM achieves the lowest accuracy, while the attention-enhanced LSTM improves the performance upon vanilla LSTM, and our pointer mixture network achieves the highest accuracy. Besides, we can see that by increasing the vocabulary size in JS or PY dataset, the OoV rate decreases, and the general accuracies on different models increase due to more available information. We also notice a performance gain which pointer mixture network achieves upon attention-enhanced LSTM, and the gain is largest with 1k vocabulary size. We attribute this performance gain to correctly predicting some OoV values through the local pointer component. Therefore, the results demonstrate the effectiveness of our pointer mixture network to predict OoV words, especially when the vocabulary is small and the OoV rate is large.

State-of-the-Art Comparison As there are already several prior work conducting code completion task on the two benchmarked datasets, to validate the effectiveness of our proposed approaches, we need to compare against the state-of-the-arts. Particularly, Liu et al. (2016) employ a standard LSTM on the JS dataset, without using any attention or pointer mechanisms. Raychev et al. (2016) build a probabilistic model for code based on probabilistic grammars and decision tree learning, and achieve the state-of-the-art accuracies for code completion on the two datasets.

Specifically, we conduct experiments on next AST node type prediction and next AST node value prediction respectively. For the former, there is no *unknown word problem* due to the small type vocabulary, so we only use the vanilla LSTM and the attention-enhanced LSTM. For the latter,

Table 3: Comparisons against the state-of-the-arts. The upper part is the results from our experiments while the lower part is the results from prior work. TYPE means next node type prediction and VALUE means next node value prediction.

	JS		**PY**	
	TYPE	VALUE	TYPE	VALUE
Vanilla LSTM	87.1%	78.6%	79.3%	67.3%
Attention-enhanced LSTM (ours)	**88.6%**	80.6%	**80.6%**	69.8%
Pointer Mixture Network (ours)	-	81.0%	-	**70.1%**
LSTM (Liu et al. 2016)	84.8%	76.6%	-	-
Probabilistic Model (Raychev et al. 2016)	83.9%	**82.9%**	76.3%	69.2%

we set the value vocabulary size to 50k to make the results comparable with (Liu et al. 2016), and employ all the three models. The results are listed in Table 3.

The upper part of Table 3 shows the results of experiments we have done in this work, while the lower part lists the results from prior work's papers. Note that Liu et al. (2016) only apply LSTM on the JS dataset, so they do not have results on the PY dataset. For next type prediction, our attention-enhanced LSTM constantly achieves the highest accuracy on both datasets, significantly improving the best records of the two datasets. For next value prediction on JS dataset, our pointer mixture network achieves comparable performance with Raychev et al.'s (2016), which is a probabilistic model based on domain-specific grammars. However, we do outperform Liu et al. (2016) a lot who also try neural network based methods (we explain it later). On PY dataset, our pointer mixture network for next value prediction outperforms the previous best record. Therefore, we conclude that our attention-enhanced LSTM and pointer mixture network are effective for code completion, achieving three state-of-the-art performances out of the four tasks.

Discussion

More explanations about experimental results. From Table 2 and Table 3, we observe that the accuracies produced in JS dataset are consistently higher than the accuracies in PY dataset, whether in type prediction or value prediction. We attribute this difference to the fact that JS dataset contains more data to train the model and fewer categories to predict than the PY dataset, as shown in Table 1. The work by Raychev et al. (2016) also confirms this accuracy gap between the two datasets. Besides, from Table 3 we notice that our vanilla LSTM outperforms the work by Liu et al. (2016) a lot who also apply a simple LSTM. We think the main reason lies in the different formulation of loss function, where Liu et al. define a loss function for both type and value prediction and train a model for them together. On the contrary, we define one loss function for each task and train them separately, which is much easier to train.

Why attention mechanism works? Like vanilla RNNs, LSTMs still suffer from *hidden state bottleneck* that carrying the whole sequence information in a fixed-size vector. The problem becomes even worse when the sequence has a longer length. In this work, the mean sequence length (i.e. the number of AST nodes of each program) is over 1000 in JS dataset and over 600 in PY dataset. Moreover, when writing programs, it is quite common to refer to a variable identifier declared many lines before. Therefore in our code completion task, the dependencies are too long to be precisely captured by a vanilla LSTM. Attention mechanism relieves this problem by attending previous hidden states and provides more paths for gradients to back propagate.

Table 4: Showing why pointer mixture network works.

	JS_1k	PY_1k
Pointer Random Network	71.4%	64.8%
Attention-enhanced LSTM	71.7%	64.9%
Pointer Mixture Network	**73.2%**	**66.4%**

Why pointer mixture network works? For vanilla LSTM and attention-enhanced LSTM, all OoV AST node values are marked as `UNK`, and all predictions where the target value is `UNK` are treated as wrong predictions. After incorporating the pointer network, we predict OoV values by copying a value from local context and that copied value may be the correct prediction. Thus we observe a performance gain in our pointer mixture network. However, one may argue that no matter how capable the pointer component is, the accuracy will definitely increase as long as we get chances to predict OoV values.

To verify the copy ability of our pointer component, we develop a *pointer random network* where the pointer distribution l_t (see Figure 3) is a random distribution instead of reusing the learned weights from attention mechanism. We conduct comparisons on value prediction in JS and PY datasets with 1k vocabulary size. The results are listed in Table 4, where the pointer random network achieves lower accuracies than the pointer mixture network. Thus we demonstrate that our pointer mixture network indeed learns how to copy some OoV values. However, the pointer random network performs even worse than the attention-enhanced LSTM. We think the reason lies in the controller of pointer random network, which is disturbed by the random noise and cannot always choose the correct component i.e. the RNN component, thus influences the overall performance.

Case Study

We depict a code completion example in Figure 4. In this example, the target prediction `employee_id` is an OoV value with respect to the whole training corpus. We show the

```
class Operator(Employee):
  def __init__(self, name, employee_id):
    super(Operator, self).__init__(name, Rank.OPERATOR)
    self.employee_id = employee_id

  def _dispatch_call(self, call, employees):
    for employee in employees:
      employee.take_call(call)

  def record_path(self, base_name):
    return os.path.join(base_name, str(self.___?___))
```

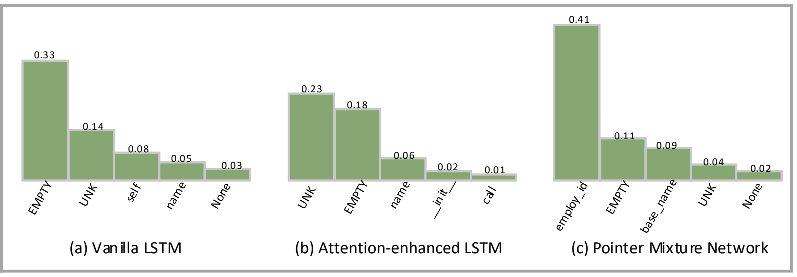

Figure 4: A code completion example showing predicting an OoV value.

top five predictions of each model. For vanilla LSTM, it just produces `EMPTY` which is the most frequent token in our corpus. For attention-enhanced LSTM, it learns from the context that the target has a large probability to be `UNK`, but fails to produce the real value. Pointer mixture network successfully points out the OoV value from the context, as it observes the value appears in the previous code.

Related Work

Our work synthesizes two strands of research, namely statistical code completion and neural language modeling.

Statistical code completion. There is a body of recent work that explores the application of statistical learning based sequence models on the code completion task. Early attempts simply use n-gram models or their extensions (Hindle et al. 2012; Tu, Su, and Devanbu 2014). Probabilistic grammars are also employed for code completion (Allamanis and Sutton 2014; Bielik, Raychev, and Vechev 2016; Raychev, Bielik, and Vechev 2016). Bielik et al. (2016) generalized the traditional probabilistic context-free grammars (PCFGs) to condition on more grammar rules thus captured a richer context relevant to the current prediction. Further, Raychev et al. (2016) improved upon Bielik et al.'s work by introducing decision tree approaches within their infrastructure, and achieved the state-of-the-art code completion accuracy on two datasets which we also use in our work.

Recently, neural sequence models like RNNs for code completion become increasingly popular, as they can capture longer dependencies and deeper semantics than n-grams (Raychev, Vechev, and Yahav 2014; White et al. 2015). In particular, Liu et al. (2016) attempted to solve the same problem as in our work. However, instead of trying attention or pointer mechanisms as we do, they only employed vanilla LSTMs. Bhoopchand et al. (2016) further proposed a special mechanism called sparse pointer, for better predicting identifiers in Python source code. Nevertheless, their pointer component targets at identifiers in Python source code, rather than out-of-vocabulary (OoV) words in our work. Besides, their corpus is quite different from ours, where they directly serialize each program as a sequence of code tokens, while in our corpus each program is represented as a sequence of nodes on corresponding AST to facilitate more intelligent structure prediction.

Neural language modeling. In terms of language modeling, deep learning techniques such as RNNs have been shown to achieve the state-of-the-art results (Mikolov et al. 2010). Other work also explored various RNN regularization methods to prevent overfitting (Zaremba, Sutskever, and Vinyals 2014). As standard RNNs suffer from the well-known gradient vanishing, to ease the problem, some language and translation models have added the soft attention or memory mechanisms (Bahdanau, Cho, and Bengio 2014; Cheng, Dong, and Lapata 2016; Tran, Bisazza, and Monz 2016). Pointer is another mechanism proposed recently (Vinyals, Fortunato, and Jaitly 2015), which can select elements from the input as output thus give RNNs the ability to "copy". The pointer mechanism is shown to be helpful in tasks like summarization (Gu et al. 2016), neural machine translation (Gulcehre et al. 2016), code generation (Ling et al. 2016), as well as language modeling (Merity et al. 2016).

Specially, the work of Gulcehre et al. (2016) and Merity et al. (2016) shares a similar idea as ours, i.e., generating new words at each timestep based on a standard RNN component and a local pointer component. Our work differs from the first work in that, their scenario is sequence-to-sequence tasks such as neural machine translation, their attention window is fixed to cover the whole source sentence. While our scenario is neural language modeling, when generating new words each time we need to slide forward our attention window. Our work differs from the second work in that, they employ the pointer component to effectively reproduce rare words, whose outputs are still IDs in the global vocabulary rather than locations in local context, whereas our pointer component is specialized to tackle out-of-vocabulary words. Further, the two work's corpora are natural language while our corpus is program source code.

Conclusion and Future Work

In this paper, we apply neural language models on the code completion task by learning from large codebases, and demonstrate the effectiveness of attention mechanism on code completion. To deal with the OoV values in code completion, we propose a pointer mixture network which learns to either generate a new value through an RNN component, or copy an OoV value from local context through a pointer component. Experiments on two benchmarked datasets show the superiority of our pointer mixture network on the code completion task. Future work includes incorporating more static type information of programs and extending our models towards automated code generation.

References

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. 2016. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, 265–283.

Allamanis, M., and Sutton, C. 2014. Mining idioms from source code. In *FSE'14: Proc. of the International Symposium on Foundations of Software Engineering*, 472–483.

Bahdanau, D.; Cho, K.; and Bengio, Y. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bengio, Y.; Simard, P.; and Frasconi, P. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks* 5(2):157–166.

Bhoopchand, A.; Rocktäschel, T.; Barr, E.; and Riedel, S. 2016. Learning python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307*.

Bielik, P.; Raychev, V.; and Vechev, M. 2016. Phog: probabilistic model for code. In *ICML'16: Proc. of the International Conference on Machine Learning*, 2933–2942.

Cheng, J.; Dong, L.; and Lapata, M. 2016. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*.

Gu, J.; Lu, Z.; Li, H.; and Li, V. O. 2016. Incorporating copying mechanism in sequence-to-sequence learning. *arXiv preprint arXiv:1603.06393*.

Gulcehre, C.; Ahn, S.; Nallapati, R.; Zhou, B.; and Bengio, Y. 2016. Pointing the unknown words. *arXiv preprint arXiv:1603.08148*.

Han, S.; Wallace, D. R.; and Miller, R. C. 2009. Code completion from abbreviated input. In *ASE'09: Proc. of the International Conference on Automated Software Engineering*, 332–343.

Hindle, A.; Barr, E. T.; Su, Z.; Gabel, M.; and Devanbu, P. 2012. On the naturalness of software. In *ICSE'12: Proc. of the International Conference on Software Engineering*, 837–847.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural computation* 9(8):1735–1780.

Kingma, D., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Li, J.; He, P.; Zhu, J.; and Lyu, M. R. 2017. Software defect prediction via convolutional neural network. In *QRS'17: Proc. of the International Conference on Software Quality, Reliability and Security*, 318–328.

Ling, W.; Grefenstette, E.; Hermann, K. M.; Kočiskỳ, T.; Senior, A.; Wang, F.; and Blunsom, P. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*.

Liu, C.; Wang, X.; Shin, R.; Gonzalez, J. E.; and Song, D. 2016. Neural code completion.

Merity, S.; Xiong, C.; Bradbury, J.; and Socher, R. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*.

Mikolov, T.; Karafiát, M.; Burget, L.; Cernockỳ, J.; and Khudanpur, S. 2010. Recurrent neural network based language model. In *INTERSPEECH*.

Mou, L.; Li, G.; Zhang, L.; Wang, T.; and Jin, Z. 2016. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, 1287–1293.

Raychev, V.; Bielik, P.; and Vechev, M. 2016. Probabilistic model for code with decision trees. In *OOPSLA'16: Proc. of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 731–747.

Raychev, V.; Vechev, M.; and Yahav, E. 2014. Code completion with statistical language models. In *ACM SIGPLAN Notices*, volume 49, 419–428.

Sundermeyer, M.; Schlüter, R.; and Ney, H. 2012. Lstm neural networks for language modeling. In *Annual Conference of the International Speech Communication Association*.

Tran, K.; Bisazza, A.; and Monz, C. 2016. Recurrent memory networks for language modeling. *arXiv preprint arXiv:1601.01272*.

Tu, Z.; Su, Z.; and Devanbu, P. 2014. On the localness of software. In *FSE'14: Proc. of the International Symposium on Foundations of Software Engineering*, 269–280.

Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015. Pointer networks. In *NIPS'15: Advances in Neural Information Processing Systems*, 2692–2700.

White, M.; Vendome, C.; Linares-Vásquez, M.; and Poshyvanyk, D. 2015. Toward deep learning software repositories. In *MSR'15: Proc. of the Working Conference on Mining Software Repositories*, 334–345.

Zaremba, W.; Sutskever, I.; and Vinyals, O. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.