A Convolutional Attention Network for Extreme Summarization of Source Code

Citation for published version:
Allamanis, M, Peng, H & Sutton, C 2016, A Convolutional Attention Network for Extreme Summarization of Source Code. in *Proceedings of The 33rd International Conference on Machine Learning, PMLR.* vol. 48, Proceedings of Machine Learning Research, vol. 48, PMLR, New York, United States, pp. 2091-2100, 33rd International Conference on Machine Learning, New York, United States, 19/06/16. <http://proceedings.mlr.press/v48/allamanis16.html>

A Convolutional Attention Network for Extreme Summarization of Source Code

Miltiadis Allamanis M.ALLAMANIS@ED.AC.UK
School of Informatics, University of Edinburgh, Edinburgh, EH8 9AB, United Kingdom

Hao Peng† PENGHAO.PKU@GMAIL.COM
School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China

Charles Sutton CSUTTON@INF.ED.AC.UK
School of Informatics, University of Edinburgh, Edinburgh, EH8 9AB, United Kingdom

Abstract

Attention mechanisms in neural networks have proved useful for problems in which the input and output do not have fixed dimension. Often there exist features that are locally translation invariant and would be valuable for directing the model's attention, but previous attentional architectures are not constructed to learn such features specifically. We introduce an attentional neural network that employs convolution on the input tokens to detect local time-invariant and long-range topical attention features in a context-dependent way. We apply this architecture to the problem of extreme summarization of source code snippets into short, descriptive function name-like summaries. Using those features, the model sequentially generates a summary by marginalizing over two attention mechanisms: one that predicts the next summary token based on the attention weights of the input tokens and another that is able to copy a code token as-is directly into the summary. We demonstrate our convolutional attention neural network's performance on 10 popular Java projects showing that it achieves better performance compared to previous attentional mechanisms.

1. Introduction

Deep learning for structured prediction problems, in which a sequence (or more complex structure) of predictions need to be made given an input sequence, presents special difficulties, because not only are the input and output high-dimensional, but the dimensionality is not fixed in advance. Recent research has tackled these problems using neural models of attention (Mnih et al., 2014), which have had great recent successes in machine translation (Bahdanau et al., 2015) and image captioning (Xu et al., 2015). Attentional models have been successful because they separate two different concerns: predicting which input locations are most relevant to each location of the output; and actually predicting an output location given the most relevant inputs.

In this paper, we suggest that many domains contain translation-invariant features that can help to determine the most useful locations for attention. For example, in a research paper, the sequence of words "in this paper, we suggest" often indicates that the next few words will be important to the topic of the paper. As another example, suppose a neural network is trying to predict the name of a method in the Java programming language from its body. If we know that this method name begins with `get` and the method body contains a statement `return ____ ;` , then whatever token fills in the blank is likely to be useful for predicting the rest of the method name. Previous architectures for neural attention are not constructed to learn translation-invariant features specifically.

We introduce a neural convolutional attentional model, that includes a convolutional network within the attention mechanism itself. Convolutional models are a natural choice for learning translation-invariant features while using only a small number of parameters and for this reason have been highly successful in non-attentional models for images (LeCun et al., 1998; Krizhevsky et al., 2012) and text classification (Blunsom et al., 2014). But to our knowledge they have not been applied within an attentional mechanism. Our model uses a set of convolutional layers — without any pooling — to detect patterns in the input and identify "interesting" locations where attention should be focused.

We apply this network to an "extreme" summarization prob-

† Work partially done while author was as an intern at the University of Edinburgh.

lem: We ask the network to predict a short and descriptive name of a source code snippet (*e.g.* a method body) given solely its tokens. Source code has two distinct roles: it not only is a means of instructing a CPU to perform a computation but also acts as an essential means of communication among developers who need to understand, maintain and evolve software systems. For these reasons, software engineering research has found that good names are important to developers (Liblit et al., 2006; Takang et al., 1996; Binkley et al., 2013). Additionally, learning to summarize source code has important applications in software engineering, such as in code understanding and in code search. The highly structured form of source code makes convolution naturally suited for the purpose of extreme summarization. Our choice of problem is inspired by previous work (Allamanis et al., 2015a) that tries to name existing methods (functions) using a large set of hard-coded features, such as features from the containing class and the method signature. But these hard-coded features may not be available for arbitrary code snippets and in dynamically typed languages. In contrast, in this paper we consider a more general problem: given an arbitrary snippet of code — without any hard-coded features — provide a summary, in the form of a descriptive method name.

This problem resembles a summarization task, where the method name is viewed as the summary of the code. However, extreme source code summarization is drastically different from natural language summarization, because unlike natural language, source code is unambiguous and highly structured. Furthermore, a good summary needs to explain *how* the code instructions compose into a higher-level meaning and not naïvely explain what the code does. This necessitates learning higher-level patterns in source code that uses both the structure of the code and the identifiers to detect and explain complex code constructs. Our extreme summarization problem may also be viewed as a translation task, in the same way that any summarization problem can be viewed as translation. But a significant difference from translation is that the input source code sequence tends to be very large (72 on average in our data) and the output summary very small (3 on average in our data). The length of the input sequence necessitates the extraction of both temporally invariant attention features and topical sentence-wide features and — as we show in this paper — existing neural machine translation techniques yield sub-optimal results.

Furthermore, source code presents the challenge of out-of-vocabulary words. Each new software project and each new source file introduces new vocabulary about aspects of the software's domain, data structures, and so on. This vocabulary often does not appear in the training set. To address this problem, we introduce a *copy mechanism*, which uses the convolutional attentional mechanism to identify important tokens in the input even if they are out-of-vocabulary tokens that do not appear in the training set. The decoder, using a meta-attention mechanism, may choose to copy tokens directly from the input to the output sequence, resembling the functionality of Vinyals et al. (2015).

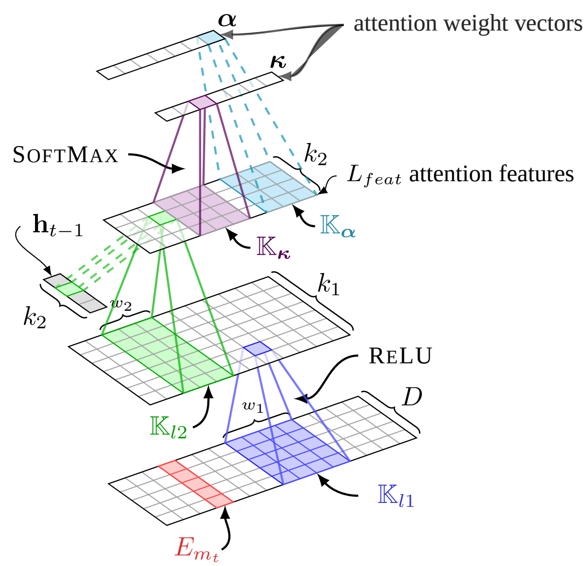

Figure 1. The architecture of the convolutional attentional network. **attention_features** learns location-specific attention features given an input sequence $\{m_i\}$ and a context vector $\mathbf{h}_{t-1}$. Given these features **attention_weights** —using a convolutional layer and a SoftMax— computes the final attention weight vectors such as $\boldsymbol{\alpha}$ and $\boldsymbol{\kappa}$ in this figure.

The key contributions of our paper are: (a) a novel convolutional attentional network that successfully performs extreme summarization of source code; (b) a comprehensive approach to the extreme code summarization problem, with interest both in the machine learning and software engineering community; and (c) a comprehensive evaluation of four competing algorithms on real-world data that demonstrates the advantage of our method compared to standard attentional mechanisms.

2. Convolutional Attention Model

Our convolutional attentional model receives as input a sequence of code subtokens[1] $\mathbf{c} = [c_{\texttt{<s>}}, c_1, \dots c_N, c_{\texttt{</s>}}]$ and outputs an extreme summary in the form of a concise method name. The summary is a sequence of subtokens $\mathbf{m} = [m_{\texttt{<s>}}, m_1 \dots m_M, m_{\texttt{</s>}}]$, where `<s>` and `</s>` are the special start and end symbols of every subtoken

[1]Subtokens refer to the parts of a source code token *e.g.* `getInputStream` has the `get`, `Input` and `Stream` subtokens.

sequence. For example, in the `shouldRender` method (top left of Table 3) the input code subtokens are $\mathbf{c}$ = [`<S>`, `try`, `{`, `return`, `render`, `requested`, ...] while the target output is $\mathbf{m}$ = [`<s>`, `should`, `render`, `</s>`]. The neural network predicts each summary subtoken sequentially and models $P(m_t | m_{\text{<s>}}, \ldots, m_{t-1}, \mathbf{c})$. Information about the previously produced subtokens $m_{\text{<s>}}, \ldots, m_{t-1}$ is passed into a recurrent neural network that represents the input state with a vector $\mathbf{h}_{t-1}$. Our convolutional attentional neural network (Figure 1) uses the input state $\mathbf{h}_{t-1}$ and a series of convolutions over the embeddings of the tokens $\mathbf{c}$ to compute a matrix of attention features L_{feat}, (Figure 1) that contains one vector of attention features for each sequence position. The resulting features are used to compute one or more normalized attention vectors (*e.g.* $\boldsymbol{\alpha}$ in Figure 1) which are distributions over input token locations containing a weight (in $\mathbb{R}^{(0,1)}$) for each subtoken in $\mathbf{c}$. Finally, given the weights, a context representation is computed and is used to predict the probability distribution over the targets m_i. This model is a generative bimodal model of summary text given a code snippet.

2.1. Learning Attention Features

We describe our model from the bottom-up (Figure 1). First we discuss how to compute the attention features L_{feat} from the input $\mathbf{c}$ and the previous hidden state $\mathbf{h}_{t-1}$. The basic building block of our model is a convolutional network (LeCun et al., 1990; Collobert & Weston, 2008) for extracting position and context-dependent features. The input to **attention_features** is a sequence of code subtokens $\mathbf{c}$ of length $\text{LEN}(\mathbf{c})$ and each location is mapped to a matrix of attention features L_{feat}, with size $(\text{LEN}(\mathbf{c}) + const) \times k_2$ where the $const$ is a fixed amount of padding. The intuition behind **attention_features** is that given the input $\mathbf{c}$, it uses convolution to compute k_2 features for each location. By then using $\mathbf{h}_{t-1}$ as a multiplicative gating-like mechanism, only the currently relevant features are kept in L_2. In the final stage, we normalize L_2. **attention_features** is described with the following pseudocode:

attention_features (code tokens $\mathbf{c}$, context $\mathbf{h}_{t-1}$)
$C \leftarrow \text{LOOKUPANDPAD}(\mathbf{c}, E)$
$L_1 \leftarrow \text{RELU}(\text{CONV1D}(C, \mathbb{K}_{l1}))$
$L_2 \leftarrow \text{CONV1D}(L_1, \mathbb{K}_{l2}) \odot \mathbf{h}_{t-1}$
$L_{feat} \leftarrow L_2 / \left\|L_2\right\|_2$
return L_{feat}

Here $E \in \mathbb{R}^{|V| \times D}$ contains the D dimensional embedding of each subtoken in names and code (*i.e.* all possible c_is and m_is). The two convolution kernels are $\mathbb{K}_{l1} \in \mathbb{R}^{D \times w_1 \times k_1}$ and $\mathbb{K}_{l2} \in \mathbb{R}^{k_1 \times w_2 \times k_2}$, where w_1, w_2 are the window sizes of the convolutions and RELU refers to a rectified linear unit (Nair & Hinton, 2010). The vector $\mathbf{h}_{t-1} \in \mathbb{R}^{k_2}$ represents information from the previous subtokens $m_0 \ldots m_{t-1}$. CONV1D performs a one-dimensional (throughout the length of sentence $\mathbf{c}$) narrow convolution. Note that the input sequence $\mathbf{c}$ is padded by LOOKUPANDPAD. The size of the padding is such that with the narrow convolutions, the attention vector (returned by **attention_weights**) has exactly $\text{LEN}(\mathbf{c})$ components. The $\odot$ operator is the elementwise multiplication of a vector and a matrix, *i.e.* $B = A \odot \mathbf{v}$ for $\mathbf{v} \in \mathbb{R}^M$ and A a $M \times N$ matrix, $B_{ij} = A_{ij} v_i$. We found the normalization of L_2 into L_{feat} to be useful during training. We believe it helps because of the widely varying lengths of inputs $\mathbf{c}$. Note that no pooling happens in this model; the input sequence $\mathbf{c}$ is of the same length as the output sequence (modulo the padding).

To compute the final attention weight vector — a vector with non-negative elements and unit norm — we define **attention_weights** as a function that accepts L_{feat} from **attention_features** and a convolution kernel $\mathbb{K}$ of size $k_2 \times w_3 \times 1$. **attention_weights** returns the normalized attention weights vector with length $\text{LEN}(\mathbf{c})$ and is described by the following pseudocode:

attention_weights (attention features L_{feat}, kernel $\mathbb{K}$)
return $\text{SOFTMAX}(\text{CONV1D}(L_{feat}, \mathbb{K}))$

Computing the State $\mathbf{h}_t$. Predicting the full summary $\mathbf{m}$ is a sequential prediction problem, where each subtoken m_t is sequentially predicted given the previous state containing information about the previous subtokens $m_0 \ldots m_{t-1}$. The state is passed through $\mathbf{h}_t \in \mathbb{R}^{k_2}$ computed by a Gated Recurrent Unit (Cho et al., 2014) *i.e.*

GRU(current input $\mathbf{x}_t$, previous state $\mathbf{h}_{t-1}$)
$\mathbf{r}_t \leftarrow \sigma(\mathbf{x}_t W_{xr} + \mathbf{h}_{t-1} W_{hr} + \mathbf{b}_r)$
$\mathbf{u}_t \leftarrow \sigma(\mathbf{x}_t W_{xu} + \mathbf{h}_{t-1} W_{hu} + \mathbf{b}_u)$
$\mathbf{c}_t \leftarrow \tanh(\mathbf{x}_t W_{xc} + r_t \odot (\mathbf{h}_{t-1} W_{hc}) + \mathbf{b}_c)$
$\mathbf{h}_t \leftarrow (1 - \mathbf{u}_t) \odot \mathbf{h}_{t-1} + \mathbf{u}_t \odot \mathbf{c}_t$
return $\mathbf{h}_t$

During testing the next state is computed by $\mathbf{h}_t = \text{GRU}(E_{m_t}, \mathbf{h}_{t-1})$ *i.e.* using the embedding of the current output subtoken m_t. For regularization during training, we use a trick similar to Bengio et al. (2015) and with probability equal to the dropout rate we compute the next state as $\mathbf{h}_t = \text{GRU}(\hat{\mathbf{n}}, \mathbf{h}_{t-1})$, where $\hat{\mathbf{n}}$ is the predicted embedding.

2.2. Simple Convolutional Attentional Model

We now use the components described above as building blocks for our extreme summarization model. We first build **conv_attention**, a convolutional attentional model that uses an attention vector $\boldsymbol{\alpha}$ computed from **attention_weights** to weight the embeddings of the tokens in $\mathbf{c}$ and compute the predicted target embedding $\hat{\mathbf{n}} \in \mathbb{R}^D$. It returns a distribution over all subtokens in

V.

conv_attention (code $\mathbf{c}$, previous state $\mathbf{h}_{t-1}$)
  $L_{feat} \leftarrow$ **attention_features**$(\mathbf{c}, \mathbf{h}_{t-1})$
  $\boldsymbol{\alpha} \leftarrow$**attention_weights** $(L_{feat}, \mathbb{K}_{att})$
  $\hat{\mathbf{n}} \leftarrow \sum_i \alpha_i E_{c_i}$
  $\mathbf{n} \leftarrow \textsc{SoftMax}(E\,\hat{\mathbf{n}}^\top + \mathbf{b})$
  return $\textsc{ToMap}(\mathbf{n}, V)$

where $\mathbf{b} \in \mathbb{R}^{|V|}$ is a bias vector and ToMap returns a map of each subtoken in $v_i \in V$ associated with its probability n_i. We train this model using maximum likelihood. Generating from the model works as follows: starting with the special $m_0 =$ `<s>` subtoken and $\mathbf{h}_0$, at each timestep t the next subtoken m_t is generated using the probability distribution $\mathbf{n}$ returned by **conv_attention** $(\mathbf{c}, \mathbf{h}_{t-1})$. Given the new subtoken m_t, we compute the next state $\mathbf{h}_t = \text{GRU}(E_{m_t}, \mathbf{h}_{t-1})$. The process stops when the special `</s>` subtoken is generated.

2.3. Copy Convolutional Attentional Model

We extend **conv_attention** by using an additional attention vector $\boldsymbol{\kappa}$ as a copying mechanism that can suggest out-of-vocabulary subtokens. In our data a significant proportion of the output subtokens (about 35%) appear in $\mathbf{c}$. Motivated by this, we extend **conv_attention** and allow a direct copy from the input sequence $\mathbf{c}$ into the summary. Now the network when predicting m_t, with probability λ copies a token from $\mathbf{c}$ into m_t and with probability $1-\lambda$ predicts the target subtoken as in **conv_attention**. Essentially, λ acts as a *meta-attention*. When copying, a token c_i is copied into m_t with probability equal to the attention weight κ_i. The process is the following:

copy_attention (code $\mathbf{c}$, previous state $\mathbf{h}_{t-1}$)
  $L_{feat} \leftarrow$ **attention_features** $(\mathbf{c}, \mathbf{h}_{t-1})$
  $\boldsymbol{\alpha} \leftarrow$**attention_weights** $(L_{feat}, \mathbb{K}_{att})$
  $\boldsymbol{\kappa} \leftarrow$**attention_weights** $(L_{feat}, \mathbb{K}_{copy})$
  $\lambda \leftarrow \max(\sigma(\textsc{Conv1d}(L_{feat}, \mathbb{K}_{\lambda})))$
  $\hat{\mathbf{n}} \leftarrow \sum_i \alpha_i E_{c_i}$
  $\mathbf{n} \leftarrow \textsc{SoftMax}(E\,\hat{\mathbf{n}}^\top + \mathbf{b})$
  return $\lambda\textsc{Pos2Voc}(\boldsymbol{\kappa}, \mathbf{c}) + (1-\lambda)\textsc{ToMap}(\mathbf{n}, V)$

where σ is the sigmoid function, $\mathbb{K}_{att}$, $\mathbb{K}_{copy}$ and $\mathbb{K}_{\lambda}$ are different convolutional kernels, $\mathbf{n} \in \mathbb{R}^{|V|}$, $\boldsymbol{\alpha}, \boldsymbol{\kappa} \in \mathbb{R}^{\textsc{Len}(\mathbf{c})}$, Pos2Voc returns a map of each subtoken in $\mathbf{c}$ (which may include out-of-vocabulary tokens) to the probabilities κ_i assigned by the copy mechanism. Finally, the predictions of the two attention mechanisms are merged, returning a map that contains all potential target subtokens in $V \cup \mathbf{c}$ and interpolating over the two attention mechanisms, using the meta-attention weight λ. Note that $\boldsymbol{\alpha}$ and $\boldsymbol{\kappa}$ are analogous attention weights but are computed from different kernels, and that $\mathbf{n}$ is computed exactly as in **conv_attention**.

Objective. To obtain signal for the copying mechanism and λ, we input to **copy_attention** a binary vector $\mathbb{I}_{\mathbf{c}=m_t}$ of size $\textsc{Len}(\mathbf{c})$ where each component is one if the code subtoken is identical to the current target subtoken m_t. We can then compute the probability of a correct copy over the marginalization of the two mechanisms, *i.e.*

$$P(m_t|\mathbf{h}_{t-1}, \mathbf{c}) = \lambda \sum_i \kappa_i \mathbb{I}_{c_i=m_t} + (1-\lambda)\mu r_{m_t}$$

where the first term is the probability of a correct copy (weighted by λ) and the second term is the probability of the target token m_t (weighted by $1-\lambda$). We use $\mu \in (0,1]$ to penalize the model when the simple attention predicts an Unk but the subtoken can be predicted exactly by the copy mechanism, otherwise $\mu = 1$. We arbitrarily used $\mu = e^{-10}$, although variations did not affect performance.

2.4. Predicting Names

To predict a full method name, we use a hybrid breath-first search and beam search. We start from the special $m_0 =$ `<s>` subtoken and maintain a (max-)heap of the highest probability partial predictions so far. At each step, we pick the highest probability prediction and predict its next subtokens, pushing them back to the heap. When the `</s>` subtoken is generated the suggestion is moved onto the list of suggestions. Since we are interested in the top k suggestions, at each point, we prune partial suggestions that have a probability less than the current best kth full suggestion. To make the process tractable, we limit the partial suggestion heap size and stop iterating after 100 steps.

3. Evaluation

Dataset Collection. We are interested in the extreme summarization problem where we summarize a source code snippet into a short and concise method-like name. Although such a dataset does not exist for arbitrary snippets of source code, it is natural to consider existing method (function) bodies as our snippets and the method names picked by the developers as our target extreme summaries.

To collect a good dataset of good quality, we cloned 11 open source Java projects from GitHub. We obtained the most popular projects by taking the sum of the z-scores of the number of watchers and forks of each project, using GHTorrent (Gousios & Spinellis, 2012). We selected the top 11 projects that contained more than 10MB of source code files each and use libgdx as a development set. These projects have thousands of forks and stars, being widely known among software developers. The projects along with short descriptions are shown in Table 1. We used this procedure to select a mature, large, and diverse corpus of real source code. For each file, we extract the Java methods, removing methods that are overridden, are abstract or are the constructors of a class. We find the overridden methods by an approximate static analysis that checks for inheritance relationships and the `@Override` annotation. Overridden methods are removed, since they are highly repetitive and their names are easy to predict. Any full tokens that are identical to the method name (*e.g.* in recursion) are replaced with a special SELF token. We split and lowercase each method name and code token into subtokens $\{m_i\}$ and $\{c_i\}$ on `camelCase` and `snake_case`. The dataset and code can be found at groups.inf.ed.ac.uk/cup/codeattention.

Experimental Setup. To measure the quality of our suggestions we compute two scores. *Exact match* is the percentage of the method names predicted exactly, while the F1 score is computed in a per-subtoken basis. When suggesting summaries, each model returns a ranked list. We compute exact match and F1 at rank 1 and 5, as the best score achieved by any one of the top suggestions (*i.e.* if the fifth suggestion achieves the best F1 score, we use this one for computing F1 at rank 5). Using BLEU (Papineni et al., 2002) would have been possible, but it would not be different from F1 given the short lengths of our output sequences (3 on average). We use each project separately, training one network for each project and testing on the respective test set. This is because each project's domain varies widely and little information can be transferred among them, due to the principle of code reusability of software engineering. We note that we attempted to train a single model using all project training sets but this yielded significantly worse results for all algorithms. For each project, we split the *files* (top-level Java classes) uniformly at random into training (65%), validation (5%) and test (30%) sets. We optimize hyperparameters using Bayesian optimization with Spearmint (Snoek et al., 2012) maximizing F1 at rank 5.

For comparison, we use two algorithms: a tf-idf algorithm that computes a tf-idf vector from the code snippet subtokens and suggests the names of the nearest neighbors using cosine similarity. We also use the standard attention model of Bahdanau et al. (2015) that uses a biRNN and fully connected components, that has been successfully used in machine translation. We perform hyperparameter optimizations following the same protocol on libgdx.

Training. To train **conv_attention** and **copy_attention** we optimize the objective using stochastic gradient descent with RMSProp and Nesterov momentum (Sutskever et al., 2013; Hinton et al., 2012). We use dropout (Srivastava et al., 2014) on all parameters, parametric leaky RELUs (Maas et al., 2013; He et al., 2015) and gradient clipping. Each of the parameters of the model is initialized with normal random noise around zero, except for $\mathbf{b}$ that is initialized to the log of the empirical frequency of each target token in the training set. For **conv_attention** the optimized hyperparameters are $k_1 = k_2 = 8$, $w_1 = 24$, $w_2 = 29$, $w_3 = 10$, dropout rate 50% and $D = 128$. For **copy_attention** the optimized hyperparameters are $k_1 = 32$, $k_2 = 16$, $w_1 = 18$, $w_2 = 19$, $w_3 = 2$, dropout rate 40% and $D = 128$.

3.1. Quantitative Evaluation

Table 1 shows the F1 scores achieved by the different methods for each project while Table 2 shows a quantitative evaluation, averaged across all projects. "Standard Attention" refers to the machine translation model of Bahdanau et al. (2015). The tf-idf algorithm seems to be performing very well, showing that the bag-of-words representation of the input code is a strong indicator of its name. Interestingly, the standard attention model performs worse than tf-idf in this domain, while **conv_attention** and **copy_attention** perform the best. The copy mechanism gives a good F1 improvement at rank 1 and a larger improvement at rank 5. Although our convolutional attentional models have an exact match similar to tf-idf, they achieve a much higher precision compared to all other algorithms.

These differences in the data characteristics could be the cause of the low performance achieved by the model of Bahdanau et al. (2015). Although source code snippets resemble natural language sentences, they are more structured, much longer and vary greatly in length. In our training sets, each method has on average 72 tokens (median 25 tokens, standard deviation 156) and the output method names are made up from 3 subtokens on average ($\sigma = 1.7$).

OOV Accuracy. We measure the out-of-vocabulary (OOV) word accuracy as the percentage of the out-of-vocabulary subtokens that are correctly predicted by **copy_attention**. On average, across our dataset, 4.4% of the test method name subtokens are OOV. Naturally, the standard attention model and tf-idf have an OOV accuracy of zero, since they are unable to predict those tokens. On average we get a 10.5% OOV accuracy at rank 1 and 19.4% at rank 5. This shows that the copying mechanism is useful in this domain and especially in smaller projects that tend to have more OOV tokens. We also note that OOV accuracy varies across projects, presumably due to different coding styles.

Topical *vs.* Time-Invariant Feature Detection. The difference of the performance between the **copy_attention** and the standard attention model of Bahdanau et al. (2015) raises an interesting question. What does **copy_attention** learn that cannot be learned by the standard attention model? One hypothesis is that the biRNN of the standard attention model fails to capture long-range features, especially in very long inputs. To test our hypothesis, we shuffle the subtokens in libgdx, essentially removing all features that depend on the sequential information. Without any lo-

Project Name	Git SHA	Description	F1 tf-idf Rank 1	tf-idf Rank 5	Standard Attention Rank 1	Standard Attention Rank 5	**conv_attention** Rank 1	**conv_attention** Rank 5	**copy_attention** Rank 1	**copy_attention** Rank 5
cassandra	`53e370f`	Distributed Database	40.9	52.0	35.1	45.0	46.5	60.0	**48.1**	**63.1**
elasticsearch	`485915b`	REST Search Engine	27.8	39.5	20.3	29.0	30.8	45.0	**31.7**	**47.2**
gradle	`8263603`	Build System	30.7	45.4	23.1	37.0	35.3	52.5	**36.3**	**54.0**
hadoop-common	`42a61a4`	Map-Reduce Framework	34.7	48.4	27.0	45.7	38.0	54.0	**38.4**	**55.8**
hibernate-orm	`e65a883`	Object/Relational Mapping	53.9	63.6	49.3	55.8	57.5	67.3	**58.7**	**69.3**
intellij-community	`d36c0c1`	IDE	28.5	42.1	23.8	41.1	33.1	49.6	**33.8**	**51.5**
liferay-portal	`39037ca`	Portal Framework	59.6	70.8	55.4	70.6	63.4	75.5	**65.9**	**78.0**
presto	`4311896`	Distributed SQL query engine	41.8	53.2	33.4	41.4	46.3	59.0	**46.7**	**60.2**
spring-framework	`826a00a`	Application Framework	35.7	47.6	29.7	41.3	35.9	49.7	**36.8**	**51.9**
wildfly	`c324eaa`	Application Server	45.2	57.7	32.6	44.4	**45.5**	61.0	44.7	**61.7**

Table 1. Open source Java projects used and F1 scores achieved. Standard attention refers to the model of Bahdanau et al. (2015).

	F1 (%)		Exact Match (%)		Precision (%)		Recall (%)	
	Rank 1	Rank 5	Rank 1	Rank 5	Rank 1	Rank 5	Rank 1	Rank 5
tf-idf	40.0	52.1	**24.3**	29.3	41.6	55.2	**41.8**	51.9
Standard Attention	33.6	45.2	17.4	24.9	35.2	47.1	35.1	42.1
conv_attention	43.6	57.7	20.6	29.8	57.4	73.7	39.4	51.9
copy_attention	**44.7**	**59.6**	23.5	**33.7**	**58.9**	**74.9**	40.1	**54.2**

Table 2. Evaluation metrics averaged across projects. Standard Attention refers to the work of Bahdanau et al. (2015).

cal features all models should reduce to achieving performance similar to tf-idf. Indeed, **copy_attention** now has an F1 at rank 1 that is +1% compared to tf-idf (presumably thanks to the language model-like structure of the output), while the standard attention model worsens its performance getting an F1 score (rank 1) of 26.2%, compared to the original 41.8%. This suggests that the biRNN fails to capture long-range topical attention features.

A simpler $\mathbf{h}_{t-1}$. Since the target summaries are quite short, we tested a simpler alternative to the GRU, assigning $\mathbf{h}_{t-1} = W \times [G_{m_{t-1}}, G_{m_{t-2}}]$, where $G \in \mathbb{R}^{D\times|V|}$ is a new embedding matrix (different from the embeddings in E) and W is a $k_2 \times D \times 2$ tensor. This model is simpler and slightly faster to train and achieves similar performance to **copy_attention**, reducing F1 by less than 1%.

3.2. Qualitative Evaluation

Figure 2 shows a visualization of a small method that illustrates how **copy_attention** typically works. At the first step, it focuses its attention at the whole method and decides upon the first subtoken. In a large number of cases this includes subtokens such as `get`, `set`, `is`, `create` *etc.* In the next steps the meta-attention mechanism is highly confident about the copying mechanism ($\lambda = 0.97$ in Figure 2) and sequentially copies the correct subtokens from the code snippet into the name. We note that across many examples the copying mechanism tends to have a significantly more focused attention vector $\boldsymbol{\kappa}$, compared to the attention vector $\boldsymbol{\alpha}$. Presumably, this happens because of the different training signals of the attention mechanisms.

A second example of **copy_attention** is seen in Figure 3. Although due to space limitations this is a relatively short method, it illustrates how the model has learned both time-invariant features and topical features. It correctly detects the `==` operator and predicts that the method has a high probability of starting with `is`. Furthermore, in the next step (prediction of the m_2 `bullets` subtoken) it successfully learns to ignore the `e` prefix (prepended on all enumeration variables in that project) and the `flag` subtoken that does not provide useful information for the summary.

Table 3 presents a set of hand-picked examples from libgdx that show interesting challenges of the domain and how our **copy_attention** handles them. Understandably, the model does *not* distinguish between `should` and `is` — both implying a `boolean` return value — and instead of `shouldRender`, `isRender` is suggested. The `getAspectRatio`, `surfaceArea` and `minRunLength` examples show the challenges of describing a previously unseen abstraction. Interestingly, the model correctly recognizes that a novel (UNK) token should be predicted after `get` in `getAspectRatio`. Most surprisingly, `reverseRange` is predicted correctly, because of the structure of the code, even though no code tokens contain the summary subtokens.

4. Related Work

Convolutional neural networks have been used for image classification with great success (Krizhevsky et al., 2012; Szegedy et al., 2015; LeCun et al., 1990; 1998). More related to this work is the use of convolutional neural

```java
boolean shouldRender()
```

```java
try {
  return renderRequested||isContinuous;
} finally {
  renderRequested = false;
}
```

Suggestions: ▶is,render (27.3%) ▶is,continuous (10.6%) ▶is,requested (8.2%) ▶render,continuous (6.9%) ▶get,render (5.7%)

```java
void reverseRange(Object[] a, int lo, int hi)
```

```java
hi--;
while (lo < hi) {
  Object t = a[lo];
  a[lo++] = a[hi];
  a[hi--] = t;
}
```

Suggestions: ▶reverse,range (22.2%) ▶reverse (13.0%) ▶reverse,lo (4.1%) ▶reverse,hi (3.2%) ▶merge,range (2.0%)

```java
int createProgram()
```

```java
GL20 gl = Gdx.gl20;
int program = gl.glCreateProgram();
return program != 0 ? program : -1;
```

Suggestions: ▶create (18.36%) ▶init (7.9%) ▶render (5.0%) ▶initiate (5.0%) ▶load (3.4%)

```java
VerticalGroup right()
```

```java
align |= Align.right;
align &= ~Align.left;
return this;
```

Suggestions: ▶left (21.8%) ▶top (21.1%) ▶right (19.5%) ▶bottom (18.5%) ▶align (3.7%)

```java
boolean isBullet()
```

```java
return (m_flags & e_bulletFlag)
       == e_bulletFlag;
```

Suggestions: ▶is (13.5%) ▶is,bullet (5.5%) ▶is,enable (5.1%) ▶enable (2.8%) ▶mouse (2.7%)

```java
float getAspectRatio()
```

```java
return (height == 0) ?
    Float.NaN : width / height;
```

Suggestions: ▶get,UNK (9.0%) ▶get,height (8.7%) ▶get,width (6.5%) ▶get (5.7%) ▶get,size (4.2%)

```java
int minRunLength(int n)
```

```java
if (DEBUG) assert n >= 0;
int r = 0;
while (n >= MIN_MERGE) {
  r |= (n & 1);
  n >>= 1;
}
return n + r;
```

Suggestions: ▶min (43.7%) ▶merge (13.0%) ▶pref (1.9%) ▶space (1.0%) ▶min,all (0.8%)

```java
JsonWriter pop()
```

```java
if (named) throw
   new IllegalStateException(UNKSTRING);
stack.pop().close();
current = stack.size == 0 ?
          null : stack.peek();
return this;
```

Suggestions: ▶close (21.4%) ▶pop (10.2%) ▶first (6.5%) ▶state (3.8%) ▶remove (2.2%)

```java
Rectangle setPosition(float x, float y)
```

```java
this.x = x;
this.y = y;
return this;
```

Suggestions: ▶set (54.0%) ▶set,y (12.8%) ▶set,x (9.0%) ▶set,position (8.6%) ▶set,bounds (1.68%)

```java
float surfaceArea()
```

```java
return 4 * MathUtils.PI *
     this.radius * this.radius;
```

Suggestions: ▶dot,radius (26.5%) ▶dot (13.1%) ▶crs,radius (9.0%) ▶dot,circle (6.5%) ▶crs (4.1%)

Table 3. A sample of handpicked snippets (the sample is necessarily limited to short methods because of space limitations) and the respective suggestions that illustrate some interesting challenges of the domain and how the **copy_attention** model handles them or fails. Note that the algorithms do *not* have access to the signature of the method but only to the body. Examples taken from the libgdx Android/Java graphics library test set.

Target			Attention Vectors	λ
m_1	set	$\boldsymbol{\alpha} =$	<s> { this . use Browser Cache = use Browser Cache ; } </s>	0.012
		$\boldsymbol{\kappa} =$	<s> { this . use Browser Cache = use Browser Cache ; } </s>	
m_2	use	$\boldsymbol{\alpha} =$	<s> { this . use Browser Cache = use Browser Cache ; } </s>	0.974
		$\boldsymbol{\kappa} =$	<s> { this . use Browser Cache = use Browser Cache ; } </s>	
m_3	browser	$\boldsymbol{\alpha} =$	<s> { this . use Browser Cache = use Browser Cache ; } </s>	0.969
		$\boldsymbol{\kappa} =$	<s> { this . use Browser Cache = use Browser Cache ; } </s>	
m_4	cache	$\boldsymbol{\alpha} =$	<s> { this . use Browser Cache = use Browser Cache ; } </s>	0.583
		$\boldsymbol{\kappa} =$	<s> { this . use Browser Cache = use Browser Cache ; } </s>	
m_5	END	$\boldsymbol{\alpha} =$	<s> { this . use Browser Cache = use Browser Cache ; } </s>	0.066
		$\boldsymbol{\kappa} =$	<s> { this . use Browser Cache = use Browser Cache ; } </s>	

Figure 2. Visualization of **copy_attention** used to compute $P(m_t|m_0 \ldots m_{t-1}, \mathbf{c})$ for `setUseBrowserCache` in libgdx. The darker the color of a subtoken, they higher its attention weight. This relationship is linear. Yellow indicates the convolutional attention weight of the **conv_attention** component, while purple the attention of the copy mechanism. Since the values of $\boldsymbol{\alpha}$ are usually spread across the tokens the colors show a normalized $\boldsymbol{\alpha}$, *i.e.* $\boldsymbol{\alpha}/\left\|\boldsymbol{\alpha}\right\|_\infty$. In contrast, the copy attention weights $\boldsymbol{\kappa}$ are usually very peaky and we plot them as-is. Underlined subtokens are out-of-vocabulary. λ shows the meta-attention probability of using the copy attention $\boldsymbol{\kappa}$ *vs.* the convolutional attention $\boldsymbol{\alpha}$. More visualizations of libgdx methods can be found at http://groups.inf.ed.ac.uk/cup/codeattention/.

Target			Attention Vectors	λ
m_1	is	$\boldsymbol{\alpha} =$	<s> { return (m Flags & e Bullet Flag) == e Bullet Flag ; } </s>	0.012
		$\boldsymbol{\kappa} =$	<s> { return (m Flags & e Bullet Flag) == e Bullet Flag ; } </s>	
m_2	bullet	$\boldsymbol{\alpha} =$	<s> { return (m Flags & e Bullet Flag) == e Bullet Flag ; } </s>	0.436
		$\boldsymbol{\kappa} =$	<s> { return (m Flags & e Bullet Flag) == e Bullet Flag ; } </s>	
m_3	END	$\boldsymbol{\alpha} =$	<s> { return (m Flags & e Bullet Flag) == e Bullet Flag ; } </s>	0.174
		$\boldsymbol{\kappa} =$	<s> { return (m Flags & e Bullet Flag) == e Bullet Flag ; } </s>	

Figure 3. Visualization of **copy_attention** modeling $P(m_t|m_0 \ldots m_{t-1}, \mathbf{c})$ for `isBullet` in libgdx. The **copy_attention** captures location-invariant features and the topicality of the input code sequence. For information about the visualization see Figure 2.

networks for text classification (Blunsom et al., 2014). Closely related is the work of Denil et al. (2014) that learns representations of documents using convolution but uses the network activations to summarize a document rather than an attentional model. Rush et al. (2015) use an attention-based encoder to summarize sentences, but do not use convolution for their attention mechanism. Our work is also related to other work in attention mechanisms for text (Hermann et al., 2015) and images (Xu et al., 2015; Mnih et al., 2014) that does not use convolution to provide the attention values. Pointer networks (Vinyals et al., 2015) are similar to our copy mechanism but use an RNN for providing attention. Finally, distantly related to this work is research on neural architectures that learn code-like behaviors (Graves et al., 2014; Zaremba & Sutskever, 2014; Joulin & Mikolov, 2015; Grefenstette et al., 2015; Dyer et al., 2015; Reed & de Freitas, 2015; Neelakantan et al., 2015).

In recent years, thanks to the insight of Hindle et al. (2012) the use of probabilistic models for software engineering applications has grown. Research has mostly focused on token-level (Nguyen et al., 2013; Tu et al., 2014) and syntax-level (Maddison & Tarlow, 2014) language models of code and translation between programming languages (Karaivanov et al., 2014; Nguyen et al., 2014). Movshovitz-Attias et al. (2013) learns to predict code comments using a source code topic model. Allamanis et al. (2015b) create a generative model of source code given a natural language query and Oda et al. (2015) use machine translation to convert source code into pseudocode. Closer to our work, Raychev et al. (2015) aim to predict names and types of variables, whereas Allamanis et al. (2014) and Allamanis et al. (2015a) suggest names for variables, methods and classes. Similar to Allamanis et al. (2015a), we predict method names but using only the tokens within a method and no other features (*e.g.* method signature). Mou et al. (2016) use syntax-level convolutional neural networks to learn vector representations for code and classify student submissions into tasks without considering naming. Piech et al. (2015) also learn program embeddings from student submissions using the program state, to assist MOOC students debug their submissions but do not consider naming. Additionally, compared to Piech et al. (2015) and Mou et al. (2016) our work looks into highly diverse, non-student submission code that performs a wide range of real-world tasks.

5. Discussion & Conclusions

Modeling and understanding source code artifacts through machine learning can have a direct impact in software engineering research. The problem of extreme code summarization is a first step towards the more general goal of developing machine learning representations of source code that will allow machine learning methods to reason probabilistically about code resulting in useful software engineering tools that will help code construction and maintenance.

Additionally, source code — and its derivative artifacts — represent a new modality for machine learning with very different characteristics compared to images and natural language. Therefore, models of source code necessitate research into new methods that could have interesting parallels to images and natural language. This work is a step towards this direction: our neural convolutional attentional model attempts to “understand” the highly-structured source code text by learning both long-range features and localized patterns, achieving the best performance among other competing methods on real-world source code.

Acknowledgements

This work was supported by Microsoft Research through its PhD Scholarship Programme and the Engineering and Physical Sciences Research Council [grant number EP/K024043/1]. We would like to thank Daniel Tarlow and Krzysztof Geras for their insightful comments and suggestions.

References

Allamanis, Miltiadis, Barr, Earl T, Bird, Christian, and Sutton, Charles. Learning natural coding conventions. In *Symposium on the Foundations of Software Engineering (FSE)*, 2014.

Allamanis, Miltiadis, Barr, Earl T, Bird, Christian, and Sutton, Charles. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015a.

Allamanis, Miltiadis, Tarlow, Daniel, Gordon, Andrew, and Wei, Yi. Bimodal modelling of source code and natural language. In *ICML*, 2015b.

Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

Bengio, Samy, Vinyals, Oriol, Jaitly, Navdeep, and Shazeer, Noam. Scheduled sampling for sequence prediction with recurrent neural networks. *arXiv preprint arXiv:1506.03099*, 2015.

Binkley, Dave, Davis, Marcia, Lawrie, Dawn, Maletic, Jonathan I, Morrell, Christopher, and Sharif, Bonita. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 2013.

Blunsom, Phil, Grefenstette, Edward, Kalchbrenner, Nal, et al. A convolutional neural network for modelling sentences. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, 2014.

Cho, Kyunghyun, van Merriënboer, Bart, Bahdanau, Dzmitry, and Bengio, Yoshua. On the properties of neural machine translation: Encoder–decoder approaches. *Syntax, Semantics and Structure in Statistical Translation*, pp. 103, 2014.

Collobert, Ronan and Weston, Jason. A unified architecture for natural language processing: deep neural networks with multitask learning. In *ICML*, 2008.

Denil, Misha, Demiraj, Alban, Kalchbrenner, Nal, Blunsom, Phil, and de Freitas, Nando. Modelling, visualising and summarising documents with a single convolutional neural network. *arXiv preprint arXiv:1406.3830*, 2014.

Dyer, Chris, Ballesteros, Miguel, Ling, Wang, Matthews, Austin, and Smith, Noah A. Transition-based dependency parsing with stack long short-term memory. In *ACL*, 2015.

Gousios, Georgios and Spinellis, Diomidis. GHTorrent: Github’s data from a firehose. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, 2012.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Grefenstette, Edward, Hermann, Karl Moritz, Suleyman, Mustafa, and Blunsom, Phil. Learning to transduce with unbounded memory. In *NIPS*, 2015.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Delving deep into rectifiers: surpassing human-level performance on ImageNet classification. *arXiv preprint arXiv:1502.01852*, 2015.

Hermann, Karl Moritz, Kocisky, Tomas, Grefenstette, Edward, Espeholt, Lasse, Kay, Will, Suleyman, Mustafa, and Blunsom, Phil. Teaching machines to read and comprehend. In *NIPS*, 2015.

Hindle, Abram, Barr, Earl T, Su, Zhendong, Gabel, Mark, and Devanbu, Premkumar. On the naturalness of software. In *International Conference on Software Engineering (ICSE)*, 2012.

Hinton, Geoffrey, Srivastava, Nitsh, and Swersky, Kevin. Neural networks for machine learning. *Online Course at coursera. org, Lecture*, 6, 2012.

Joulin, Armand and Mikolov, Tomas. Inferring algorithmic patterns with stack-augmented recurrent nets. *arXiv preprint arXiv:1503.01007*, 2015.

Karaivanov, Svetoslav, Raychev, Veselin, and Vechev, Martin. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 2014.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. ImageNet classification with deep convolutional neural networks. In *NIPS*, 2012.

LeCun, Y., Boser, B., Denker, J. S., Howard, R. E., Habbard, W., Jackel, L. D., and Henderson, D. Nips. chapter Handwritten Digit Recognition with a Back-propagation Network. 1990.

LeCun, Yann, Bottou, Leon, Bengio, Yoshua, and Haffner, Patrick. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

Liblit, Ben, Begel, Andrew, and Sweetser, Eve. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*, 2006.

Maas, Andrew L., Hannun, Awni Y., and Ng, Andrew Y. Rectified linear units improve restricted Boltzmann machines. In *ICML Workshop on Deep Learning for Audio, Speech, and Language Processing (WDLASL)*, 2013.

Maddison, Chris and Tarlow, Daniel. Structured generative models of natural source code. In *ICML*, 2014.

Mnih, Volodymyr, Heess, Nicolas, Graves, Alex, et al. Recurrent models of visual attention. In *NIPS*, 2014.

Mou, Lili, Li, Ge, Zhang, Lu, Wang, Tao, and Jin, Zhi. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, 2016.

Movshovitz-Attias, Dana, Cohen, WW, and W. Cohen, William. Natural language models for predicting programming comments. In *ACL*, 2013.

Nair, Vinod and Hinton, Geoffrey E. Rectified linear units improve restricted Boltzmann machines. In *ICML*, 2010.

Neelakantan, Arvind, Le, Quoc V, and Sutskever, Ilya. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*, 2015.

Nguyen, Anh Tuan, Nguyen, Tung Thanh, and Nguyen, Tien N. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014.

Nguyen, Tung Thanh, Nguyen, Anh Tuan, Nguyen, Hoan Anh, and Nguyen, Tien N. A statistical semantic language model for source code. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013.

Oda, Yusuke, Fudaba, Hiroyuki, Neubig, Graham, Hata, Hideaki, Sakti, Sakriani, Toda, Tomoki, and Nakamura, Satoshi. Learning to generate pseudo-code from source code using statistical machine translation. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015.

Papineni, Kishore, Roukos, Salim, Ward, Todd, and Zhu, Wei-Jing. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 2002.

Piech, Chris, Huang, Jonathan, Nguyen, Andy, Phulsuksombati, Mike, Sahami, Mehran, and Guibas, Leonidas J. Learning program embeddings to propagate feedback on student code. In *ICML*, 2015.

Raychev, Veselin, Vechev, Martin, and Krause, Andreas. Predicting program properties from “big code”. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2015.

Reed, Scott and de Freitas, Nando. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Rush, Alexander M, Chopra, Sumit, and Weston, Jason. A neural attention model for abstractive sentence summarization. In *EMNLP*, 2015.

Snoek, Jasper, Larochelle, Hugo, and Adams, Ryan P. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

Srivastava, Nitish, Hinton, Geoffrey E., Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 2014.

Sutskever, Ilya, Martens, James, Dahl, George, and Hinton, Geoffrey. On the importance of initialization and momentum in deep learning. In *ICML*, 2013.

Szegedy, Christian, Liu, Wei, Jia, Yangqing, Sermanet, Pierre, Reed, Scott, Anguelov, Dragomir, Erhan, Dumitru, Vanhoucke, Vincent, and Rabinovich, Andrew. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

Takang, Armstrong A, Grubb, Penny A, and Macredie, Robert D. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 1996.

Tu, Zhaopeng, Su, Zhendong, and Devanbu, Premkumar. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014.

Vinyals, Oriol, Fortunato, Meire, and Jaitly, Navdeep. Pointer networks. In *NIPS*, 2015.

Xu, Kelvin, Ba, Jimmy, Kiros, Ryan, Cho, Kyunghyun, Courville, Aaron C, Salakhutdinov, Ruslan, Zemel, Richard S, and Bengio, Yoshua. Show, attend and tell: Neural image caption generation with visual attention. In *ICML*, 2015.

Zaremba, Wojciech and Sutskever, Ilya. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.