# Deep Transfer Learning for Source Code Modeling

Yasir Hussain[a,*], Zhiqiu Huang[a,b,c,*], Yu Zhou[a], Senzhang Wang[a]

[a]*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics (NUAA), Nanjing 211106, China*
[b]*Key Laboratory of Safety-Critical Software, NUAA, Ministry of Industry and Information Technology, Nanjing 211106, China*
[c]*Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 210093, China*

**Abstract**

In recent years, deep learning models have shown great potential in source code modeling and analysis. Generally, deep learning-based approaches are problem-specific and data-hungry. A challenging issue of these approaches is that they require training from starch for a different related problem. In this work, we propose a transfer learning-based approach that significantly improves the performance of deep learning-based source code models. In contrast to traditional learning paradigms, transfer learning can transfer the knowledge learned in solving one problem into another related problem. First, we present two recurrent neural network-based models RNN and GRU for the purpose of transfer learning in the domain of source code modeling. Next, via transfer learning, these pre-trained (RNN and GRU) models are used as feature extractors. Then, these extracted features are combined into *attention* learner for different downstream tasks. The *attention* learner leverages from the learned knowledge of pre-trained models and fine-tunes them for a specific downstream task. We evaluate the performance of the proposed approach with extensive experiments with the source code suggestion task. The results indicate that the proposed approach outperforms the state-of-the-art models in terms of accuracy, precision, recall, and F-measure without training the models from scratch.

*Keywords:* Transfer Learning, Deep Neural Language Models, Source Code Modeling, Attention Learning.

## 1. Introduction

Source code suggestion and syntax error fixing are vital features of a modern integrated development environment (IDE). These features help software developers to build and debug software rapidly. Recently, deep learning-based language models have

*Corresponding author
*Email addresses:* `yaxirhuxxain@nuaa.edu.cn` (Yasir Hussain), `zqhuang@nuaa.edu.cn` (Zhiqiu Huang), `zhouyu@nuaa.edu.cn` (Yu Zhou), `szwang@nuaa.edu.cn` (Senzhang Wang)

shown great potential in various source code modeling tasks [2, 5, 18, 25, 32, 33, 34, 51, 55, 65].

Some researchers have worked on source code suggestion [32, 65, 51] task in which they suggest the next possible source code token. They take a fixed size context prior to the prediction position as features and help the software developers by suggesting the next possible code token. Some researchers have worked on syntax error detection and correction [18, 25] problem. They consider the source code syntax as features and use them for the correction of the syntax errors found in a source code file. Several researchers have focused on the code summarization task [34, 33, 2] in which they help summarize the source code for better understand the working of it. Further, some works focus on method naming [5] which gives a meaningful name to a source code method. Some works are focused on source code generation [55] task in which they use the natural language queries to help generate the source code.

A challenging issue of these approaches is that they are problem-specific which requires training from starch for a different related problem. Further, deep learning-based approaches are data-hungry which means they require training on large data set to produce satisfactory results. Furthermore, deep learning models requires days to train while training on a large dataset. To overcome these issues, we exploit the concept of transfer learning in this work. In transfer learning, the learned knowledge from a pre-trained model is extracted and then be used for a similar downstream task [54].

This work proposes a transfer learning-based approach that significantly improves the performance of deep learning-based source code models. First, we exploit the concept of transfer learning for deep learning-based source code language models. The key idea is to use a pre-trained source code language model to transfer the learned knowledge from it for a different related problem. We train two different variants of recurrent neural network-based models RNN and GRU for the purpose of transfer learning. Then, we combine the learned knowledge of pre-trained (RNN and GRU) models into *attention* learner for a downstream task. The *attention* learner leverage from the learned knowledge of pre-trained models and fine-tunes it for a specific downstream task. Via transfer learning, pre-trained models are used to extract generalized features and then fine-tune them for a target task without requiring the model training from scratch. We evaluate the proposed approach with the downstream task of source code suggestion.

## 2. Proposed approach

This section discusses the proposed framework in detail. The Fig. 1 shows the overall architecture of the proposed approach. In this section, we first discuss the pre-trained models for the purpose of transfer learning. Next, we discuss the extraction of knowledge from pre-trained models combined with *attention* learner for the downstream task of source code suggestion. The *attention* learner is used to fine-tune the models for the target task by paying attention to the features which are related to the target task. The details about each step are given in the following subsections.

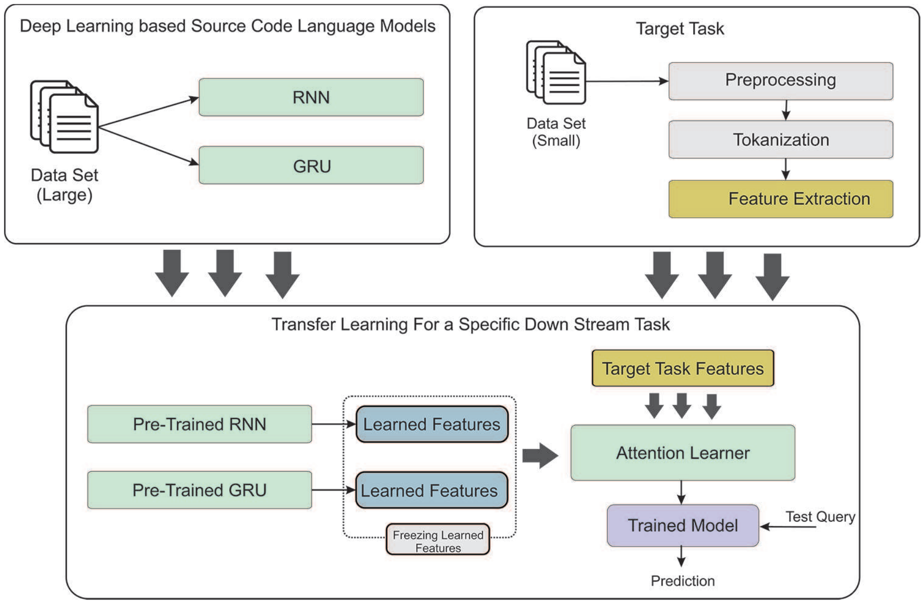
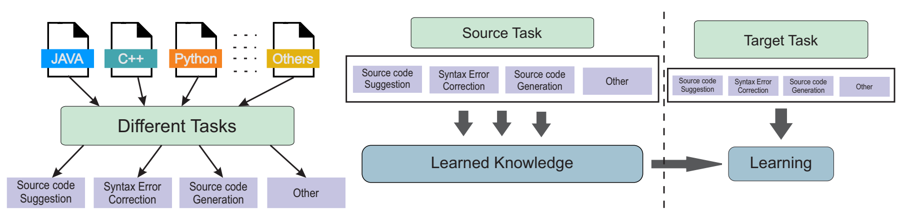



Figure 1: Overall framework of the proposed approach




(a) (b)

Figure 2: Difference between traditional learning and transfer learning based approaches. (a) Traditional learning approach; (b) Transfer learning based approach.

## *2.1. Transfer Learning*

In Transfer learning the knowledge learned in solving one problem is transferred and fine-tuned for another related problem. In recent years, transfer learning methods have been successfully applied in different fields such as metric learning [30], machine learning [17], and dimensionality reduction [47]. Further, transfer learning has been extensively studied for various tasks in the field of image and text classification [36, 57, 69, 67, 38]. Fig. 2 shows the difference between traditional learning and transfer learning-based approaches for source code modeling.

For the purpose of transfer learning, we first need a pre-trained model. There are several CNN (GoogLeNet [58] , VGGNet [12] and ResNet [27]) and NLP (BERT [14], Transformer-XL [13], OpenAIs GPT-2 [49]) based models for image and text classification respectively. The source code strictly follows the rules defined by their grammar[1], thus these models are not suitable for our purpose. In this work, we first train two variants of recurrent neural networks-based models RNN and GRU for the purpose of transfer learning in the field of source code. We choose RNN [65, 51] and GRU [32, 25] because of their recent success in the modeling of source code. To train the models for transfer learning first, we gather the data set used in previous studies [32, 29, 44]. Table 1 shows the details of the data set used to build pre-trained models. By combining all collected projects, we end up with 13 million code tokens with a large vocabulary of size 177,342.

Table 1: Data set used to pre-train models for transfer learning. The table shows name of the project, version of the project, line of code (LOC), total code tokens and unique code tokens found in each project.

| Projects | Version | LOC | Total | Vocab Size |
|---|---|---|---|---|
| ant | 1.10.5 | 149,960 | 920,978 | 17,132 |
| cassandra | 3.11.3 | 318,704 | 2734218 | 33,424 |
| db40 | 7.2 | 241,766 | 1,435,382 | 20,286 |
| jgit | 5.1.3 | 199,505 | 1,538,905 | 20,970 |
| poi | 4.0.0 | 387,203 | 2,876,253 | 47,756 |
| maven | 3.6.0 | 69,840 | 494,379 | 8,066 |
| batik | 1.10.0 | 195,652 | 1,246,157 | 21,964 |
| jts | 1.16.0 | 91,387 | 611,392 | 11,903 |
| itext | 5.5.13 | 161,185 | 1,164,362 | 19,113 |
| antlr | 4.7.1 | 56,085 | 407,248 | 6,813 |
| Total | | 1,871,287 | 13,429,274 | 177,342 |

### 2.2. *Pre-Training Models for Transfer Learning*

All models are trained on Intel(R) Xeon(R) Silver 4110 CPU 2.10GHz x 32 cores and 128GB of ram running Ubuntu 18.04.2 LTS operating system, equipped with the latest NVIDIA GeForce RTX 2080. The Table. 2 shows the architecture of trained models used for transfer learning. We follow the same approach used in previous works [32, 65, 51] to pre-process the data set. To build a global vocabulary system [65], we remove all code tokens appearing less than three times in the collected data set which ends up with the vocabulary size of 88,013 unique code tokens. We map the vocabulary to a continuous feature vector of dense size *300* similar to Word2Vec [52]. We use *300* hidden units with context size ($\tau$) of *20* as studied by White et al. [65]. For each model training we employ *Adam* [37] optimizer with the default learn rate of *0.001*. To control overfitting, we use *Dropout* [20]. Each model is trained until it converges by employing *early stop* [5] with the patience of three consecutive hits on the validation loss. One important thing to point out here is that the training process of

[1] `https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html`

Table 2: Deep learning models architecture summary used to pre-train source code models for transfer learning purpose.

| | Type | Size | Activations |
|---|---|---|---|
| Input | Code embedding | 300 | |
| Estimator | RNN,GRU | 300 | tanh |
| Over Fitting | Dropout | | |
| Output | Dense | $V$ | softmax |
| Loss | Categorical cross entropy | | |
| Optimizer | Adam | | |

these models is one time and do not need retraining. The trained models are publicly available[2] for the purpose of transfer learning.

### *2.3. Learning to Transfer Knowledge*

For transfer learning first, we prepared the pre-trained models as described earlier in this section. Then, we use these pre-trained models to transfer the learned knowledge for a downstream task. A key insight is to freeze the learned knowledge in pre-trained models to keep the learned knowledge unchanged and fine-tunes the learned knowledge for a target task. Recently, *attention*-based approaches have shown great potential in various fields such as speech recognition [11], machine translation [40, 6], and more [62, 5]. In proposed approach, we use the *attention* learner to fine-tune the model for a target task. The *attention* learner pays attention to the task-specific features to achieve optimal performance. The Fig. 3 shows the architecture design of our proposed transfer learning-based *attention* model. We show the effectiveness of the proposed approach with the downstream task of source code suggestion. A source code suggestion engine recommends the next possible source code token given a context.

#### *2.3.1. PreProcessing*

This section briefly introduces each of the strategic steps that we apply for the task of source code suggestion. Following common practices[65], we perform normalization, tokenization and feature extraction. For the illustrating example, Table 3 shows the effect of each preprocessing step. We discuss each step in detail in the following subsections.

*Normalization*

One of the vital preprocessing steps is to normalize the data set. Usually, a data set contains some values which are unnecessary for a particular task, these type of values will intensely upset the outcome of the analysis. For this purpose, we normalize the source code files by removing all blank lines, inline and block-level comments. We

[2] Trained Models: `https://github.com/yaxirhuxxain/TransferLearning`

Table 3: An example of preprocessing steps.

<table>
<tr><td>Original Source code</td><td><pre>
/* This is a simple Java program.
   FileName : "HelloWorld.java". */
class HelloWorld
{
    // Your program begins with a call to main().
    // Prints "Hello, World" to the terminal window.
    public static void main(String args[])
    {
        System.out.println("Hello, World");
    }
}
</pre></td></tr>
<tr><td>Normalized Source Code</td><td><pre>
class HelloWorld
{
    public static void main(String args[])
    {
        System.out.println(StringVal);
    }
}
</pre></td></tr>
<tr><td>Tokenized Source Code</td><td><pre>
class HelloWorld { public static void main ( String args [] ) { System . out . println ( StringVal
HelloWorld { public static void main ( String args [] ) { System . out . println ( StringVal )
{ public static void main ( String args [] ) { System . out . println ( StringVal ) ;
public static void main ( String args [] ) { System . out . println ( StringVal ) ; }
static void main ( String args [] ) { System . out . println ( StringVal ) ; } }
</pre></td></tr>
<tr><td>Vectorized Source Code</td><td><pre>
1 2 3 4 5 6 7 8 9 10 11 12 3 13 14 15 14 16 8 17
2 3 4 5 6 7 8 9 10 11 12 3 13 14 15 14 16 8 17 12
3 4 5 6 7 8 9 10 11 12 3 13 14 15 14 16 8 17 12 18
4 5 6 7 8 9 10 11 12 3 13 14 15 14 16 8 17 12 18 19
5 6 7 8 9 10 11 12 3 13 14 15 14 16 8 17 12 18 19 19
</pre></td></tr>
</table>

replace all constant numerical values to their generic types (e.g. 1 = IntVal, 1.2 = FloatVal) and replace constant strings with a generic *StringVal* token.

*Tokenization*

After normalizing source code files, we tokenize the source code files. Tokenization is the process of extracting terms/words from the data set. For this purpose, each source code file is parsed into a sequence of space-separated code tokens. Each sequence is then parted into multiple subsequences of fixed size context *20* [65].

*Feature Extraction*

To convert the source code sequences into a form that is suitable for training deep learning models we perform a series of transformations. First, we replace common tokens occurring only once in the corpus with a special token *unk* to build a global vocabulary system. Next, we build the vocabulary where each unique source code token corresponds to an entry in the vocabulary. Then each source code token is assigned a unique positive integer corresponding to its vocabulary index to convert the sequences (feature vectors) into a form that is suitable for training a deep learning model.

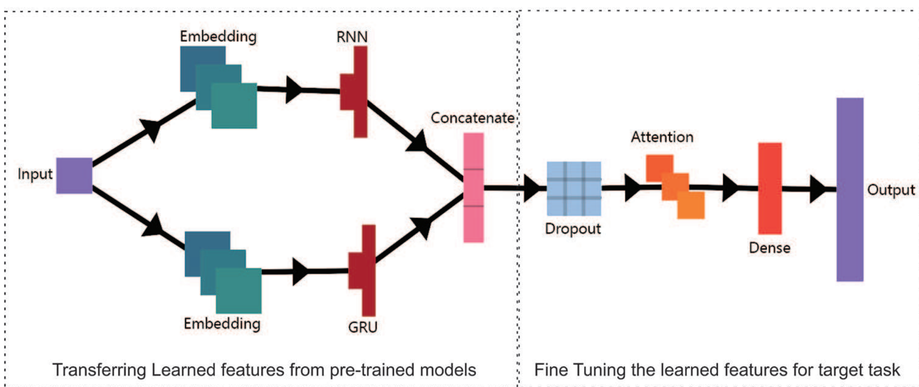



Figure 3: Proposed transfer learning based attention model architecture.

## 3. Evaluation

In this section, we evaluate the effectiveness of our proposed approach by investigating the following research questions:

- RQ1: Does the proposed approach outperform the state-of-the-art approaches? if yes, to what extent?
- RQ2: How well the proposed approach performs in terms of source code suggestion task as compared to other baseline approaches?
- RQ3: Does normalization help to improve the performance of the proposed approach? If yes, to what extent?

Table 4: Proposed transfer learning based attention model architecture summary.

| | Layers | Type | Size | Activations |
|---|---|---|---|---|
| Frozen | Input<br>Estimator<br>Combining | Code embedding<br>RNN,GRU<br>Concatenate | 300<br>300 | tanh |
| Fine Tuned | Over Fitting<br>Attention<br>Output<br>Loss<br>Optimizer | Dropout<br>Attention Learner<br>Dense<br>Categorical cross entropy<br>Adam | $V$ | softmax |

To answer the research question (RQ1), we compare the performance of the proposed approach with the state-of-the-art approaches. To answer the research question (RQ2), we evaluate and compare the proposed approach for source code suggestion tasks with other baseline approaches. To answer the research question (RQ3), We conduct a comparative analysis to show the impact of normalization on model performance.

### *3.0.1. Data set*

To empirically evaluate our work, we collected java projects from *GitHub* a well-known open-source software repositories provider. We gather the top five java projects sorted by the number of star from *GitHub* at the time of this study. We download the latest snapshot of the project usually named as the *master branch*. Here, we choose the projects which are not used while training the pre-trained models discussed in Section 2. The Table 5 shows the version of each project, the total number of code lines, total code tokens and unique code tokens found in each project.

Table 5: List of projects used for the evaluation of this work.

| | | | Code Tokens | |
|---|---|---|---|---|
| Projects | Version | LOC | Total | Vocab Size ($V$) |
| elastic-search | v7.0.0 | 210,357 | 1,765,479 | 24,691 |
| java-design-patterns | v1.20.0 | 30,784 | 200,344 | 5,649 |
| RxJava | v2.2.8 | 257,704 | 1,908,258 | 12,230 |
| interviews | v1.0 | 13,750 | 80,074 | 1,157 |
| spring-boot | v2.2.0.M2 | 224,465 | 1,813,891 | 34,609 |

### *3.1. Process and Metrics*

*Process:*

We train several baseline models for the evaluation of this work. The proposed approach is evaluated in the following manner;

- We train RNN [51] based model as baseline similar to White et al. [65].
- We train GRU based deep neural model as baseline similar to Cho et al.[10]
- We train transfer learning-based *attention* model by following the proposed approach as discussed in Section 2.

We choose the approach proposed by White et al. [65] for comparison because they have shown the effectiveness of their approach with the similar task of source code suggestion and as far as we know, considered as the state-of-the-art approach. We train the GRU [10] based model as the baseline because GRU based model is an advanced version of RNN which removes the vanishing gradient problem and performs better[32]. To empirically evaluate our work, we repeat our experiment on each project separately. We randomly partition the projects into ten equal lines of code folds from which one fold is used for testing, one fold is used for model parameter optimization (validation) and rest of the folds are used for model training. The Table. 4 shows the proposed transfer learning-based attention model architecture. First, we preprocess the data set as discussed earlier in Section 2.3.1. Then, we map the vocabulary to a continuous feature vector of dense size *300* similar to Word2Vec [52]. We use *300* hidden units with context size ($\tau$) of *20* as studied by White et al. [65]. For each model training we employ *Adam* [37] optimizer with the default learn rate of *0.001*. To control overfitting, we use *Dropout* [20]. Each model is trained until it converges by employing *early stop* [5] with the patience of three consecutive hits on the validation loss.

*Metrics:*

For the evaluation of the proposed approach, we choose similar metrics as in previous studies. We choose top-k accuracy [65, 51] and Mean Reciprocal Rank (MRR) [32, 44, 18] metrics for the evaluation of this work. Further, to evaluate the performance of the proposed approach we measure the precision, recall and F-measure scores which are widely used metrics [5]. Furthermore, to evaluate the significance of the proposed approach we perform ANOVA statistical testing.

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN} \tag{1}$$

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

$$\text{F-measure} = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{4}$$

Where *true positive (TP)* defines the total number of positive instances being identified as positive, *true negative (TN)* defines the total number of negative instances being identified as negative, *false positive (FP)* defines the total number of positive instances mistakenly identified as false and *false negative (FN)* defines the total number of positive instances mistakenly identified as negative.

## 4. Results

In this section, we will discuss and compare the results of our proposed approach with other baseline models.

### *4.0.1. Comparison against the baseline approaches*

The top-k accuracy score of the proposed approach and the baseline approaches are presented in Table 6. Form this table and Fig. 4 we make the following observations

- The average accuracy rate of RNN based model is *45.01%@k=1*, *65.56%@k=5* and *68.55%@k=10*, for the GRU based model is *50.06%@k=1*, *64.38%@k=5* and *73.27%@k=10*, while the proposed approach's average score is *66.15%@k=1*, *90.68%@k=5* and *93.97%@k=10* which is much higher as compared to the baseline approaches.
- On average the proposed approach improves the accuracy *(k@1)* by *21.14%* from RNN and *16.09%* from GRU based model.
- Results suggests that by employing the transfer learning-based *attention* model it significantly improves the model performance.

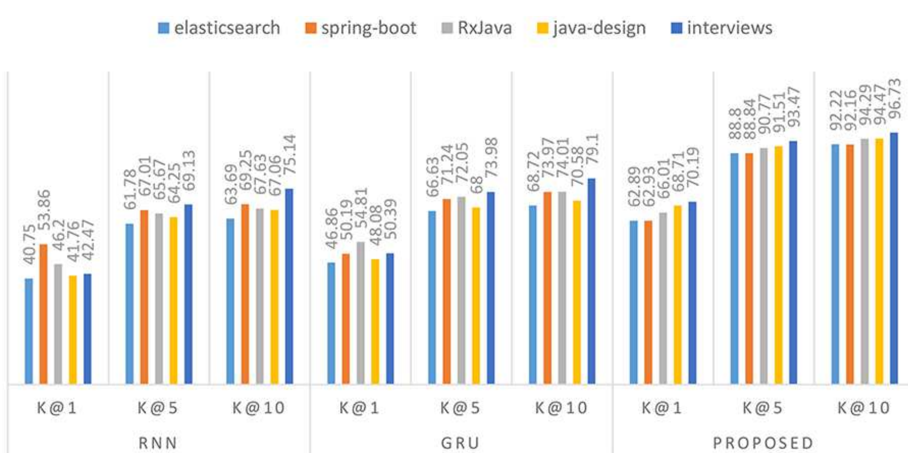



Figure 4: Top-k accuracy comparison.

Further, to evaluate the performance of the proposed approach we measure the precision, recall and F-measure scores. Table 7 exhibits the precision, recall and F-measure scores. We make the subsequent interpretations from Table 7 and Fig. 5,

Table 6: Top-k Accuracy scores with and without proposed approach

| | K | RNN | GRU | **Proposed** |
|---|---|---|---|---|
| elasticsearch | 1 | 40.75 | 46.86 | **62.89** |
| | 5 | 61.78 | 66.63 | **88.80** |
| | 10 | 63.69 | 68.72 | **92.22** |
| spring-boot | 1 | 53.86 | 50.19 | **62.93** |
| | 5 | 67.01 | 71.24 | **88.84** |
| | 10 | 69.25 | 73.97 | **92.16** |
| RxJava | 1 | 46.20 | 54.81 | **66.01** |
| | 5 | 65.67 | 72.05 | **90.77** |
| | 10 | 67.63 | 74.01 | **94.29** |
| java-design | 1 | 41.76 | 48.08 | **68.71** |
| | 5 | 64.25 | 68.00 | **91.51** |
| | 10 | 67.06 | 70.58 | **94.47** |
| interviews | 1 | 42.47 | 50.39 | **70.19** |
| | 5 | 69.13 | 73.98 | **93.47** |
| | 10 | 75.14 | 79.10 | **96.73** |
| **Average** | 1 | 45.01 | 50.06 | **66.15** |
| | 5 | 65.56 | 70.38 | **90.67** |
| | 10 | 68.55 | 73.27 | **93.97** |

- The proposed approach's average F-measure is *68.36*, while RNN and GRU gain much lower score of *39.73* and *46.20* respectively.
- The proposed approach's minimum F-measure is much higher than the maximum F-measure of the baseline approaches.
- The results advise that the anticipated method outclasses the state-of-the-art methods in precision, recall, and F-measure.

#### *4.0.2. RQ2: Comparative analysis for Source Code Suggestion Task*

To further quantify the accuracy of the proposed approach for source code suggestion task, we measure the *Mean Reciprocal Rank (MRR)* scores of each model. The MRR is a rank-based evaluation metric which produces a value between *0-1*, where the value closer to *1* indicates a better source code suggestion model. The MRR can be expressed as

$$MRR(C) = \frac{1}{|C|} \sum_{i=1}^{|C|} \frac{1}{y^i} \tag{5}$$

where $C$ is code sequence and $y^i$ refers to the index of the first relevant prediction. $MRR(C)$ is the average of all sequences $C$ in the test data set.

Table 7: Precision, Recall and F-measure comparison with baseline approaches

| | | RNN | GRU | **Proposed** |
|---|---|---|---|---|
| elasticsearch | Precision | 30.43 | 38.91 | **63.82** |
| | Recall | 40.75 | 46.86 | **82.89** |
| | F-measure | 34.84 | 42.52 | **72.11** |
| spring-boot | Precision | 37.29 | 40.77 | **62.39** |
| | Recall | 43.86 | 50.19 | **62.93** |
| | F-measure | 40.31 | 44.99 | **62.65** |
| RxJava | Precision | 41.15 | 46.54 | **66.97** |
| | Recall | 46.20 | 54.81 | **66.01** |
| | F-measure | 43.53 | 50.34 | **66.48** |
| java-design | Precision | 37.68 | 41.23 | **69.09** |
| | Recall | 41.76 | 48.08 | **68.71** |
| | F-measure | 39.62 | 44.39 | **68.89** |
| interviews | Precision | 38.11 | 47.00 | **71.02** |
| | Recall | 42.47 | 50.39 | **70.19** |
| | F-measure | 40.17 | 48.63 | **70.60** |
| **Average** | Precision | 36.93 | 42.89 | **66.66** |
| | Recall | 43.01 | 50.07 | **70.15** |
| | F-measure | 39.73 | 46.20 | **68.36** |

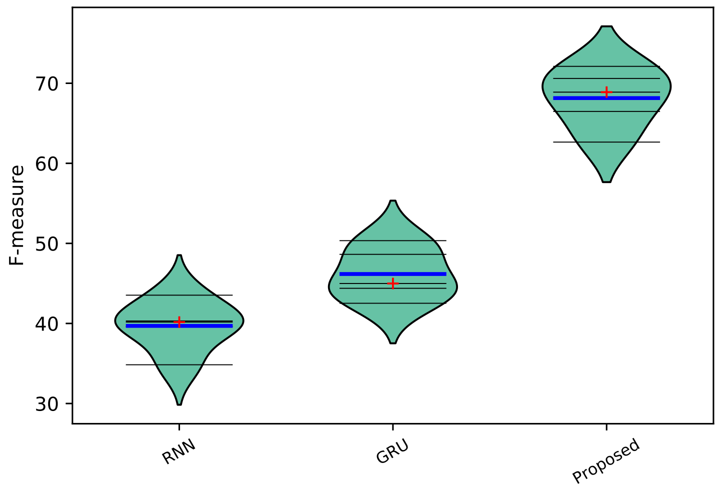



Figure 5: F-measure distribution.

The results of all models are presented in the Table 8. The average MRR score of RNN is *0.5156* and the average score of GRU is *0.5749*, while the average score of the proposed approach is *0.7618* which is much higher. The results suggest that out of four source code suggestions the proposed approach can suggest three predictions at its first rank. From the results, we conclude that the proposed approach significantly outperforms the baseline approaches.

Table 8: MRR scores with and without proposed approach

| | RNN | GRU | **Proposed** |
|---|---|---|---|
| elasticsearch | 0.4851 | 0.5405 | **0.7344** |
| spring-boot | 0.5161 | 0.5672 | **0.7363** |
| RxJava | 0.5403 | 0.6085 | **0.7619** |
| java-design | 0.5082 | 0.5625 | **0.7805** |
| interviews | 0.5284 | 0.5960 | **0.7960** |
| **Average** | 0.5156 | 0.5749 | **0.7618** |

To further validate the statistical significance, we employ the ANOVA One-Way statistical test. We conduct the AVOVA test with its default settings ($\alpha = 0.05$) using Microsoft Excel, and no modifications were made. Comparing (Table 9) the proposed approach with the best baseline (GRU), we found $F > F\text{-}crit$ and $P\text{-}value < \alpha$ is true in all cases (Accuracy, MRR, Precision, Recall and F-measure); therefore, we reject the null hypothesis, suggesting that using diverse methods has statistically significant variance in performance.

Table 9: ANOVA Analysis.

| *Source* | *SS* | *df* | *MS* | *F* | *P-value* | *F-crit* |
|---|---|---|---|---|---|---|
| *Accuracy (K@1)* | | | | | | |
| Between Groups | 646.416 | 1 | 646.416 | 64.04975 | 4.35463E-05 | 5.317655 |
| Within Groups | 80.73924 | 8 | 10.092405 | | | |
| Total | 727.1552 | 9 | | | | |
| *MRR* | | | | | | |
| Between Groups | 646.416 | 1 | 646.416 | 64.04975 | 4.35463E-05 | 5.317655 |
| Within Groups | 80.73924 | 8 | 10.092405 | | | |
| Total | 727.1552 | 9 | | | | |
| *Precision* | | | | | | |
| Between Groups | 1412.295 | 1 | 1412.29456 | 108.0003 | 6.36396E-06 | 5.317655 |
| Within Groups | 104.6141 | 8 | 13.07676 | | | |
| Total | 1516.909 | 9 | | | | |
| *Recall* | | | | | | |
| Between Groups | 1008.016 | 1 | 1008.016 | 29.81202 | 0.000601504 | 5.317655 |
| Within Groups | 270.4992 | 8 | 33.812405 | | | |
| Total | 1278.515 | 9 | | | | |
| *F-measure* | | | | | | |
| Between Groups | 1206.92 | 1 | 1206.92196 | 99.9581 | 8.5015E-06 | 5.31766 |
| Within Groups | 96.5942 | 8 | 12.07428 | | | |
| Total | 1303.52 | 9 | | | | |

Where, SS = sum of squares, df = degree of freedom, MS = mean square.

#### *4.0.3. Impact of Normalization*

The evaluation outcomes of the projected method for normalized source code and non-normalized source code are presented in Table 10. From the results, we observe that the normalization of source code improves the model performance significantly. On average the proposed approach with normalization achieves the accuracy score of 66.15@k=1 where without normalization the accuracy drops to 56.27@k=1. From the results (Table 10), we conclude that the normalization process significantly affects the model performance.

### *4.1. Discussion and Future Work*

The proposed approach attains the finest performance due to several different reasons. First, the proposed approach takes leverage from pre-trained models by transferring the learned features from them. Second, the *attention* learner fine-tunes the

Table 10: Impact of Normalization

| | Accuracy | Precision | Recall | F-measure | MRR |
|---|---|---|---|---|---|
| Normalized | 66.15 | 66.66 | 70.15 | 68.36 | 0.7618 |
| Non-Normalized | 56.27 | 57.25 | 62.14 | 54.66 | 0.6524 |

model by paying attention to only task-specific features and does not increase the computational complexity which resulted in better performance. Consequently, transfer learning-based *attention* model has better generalization capability without training the model from scratch.

The broader impact of our work is to show that transfer learning could be beneficial in the domain of source code modeling. This work is the first step in this direction and results encourage future research on it. The work can be improved in several different ways. First, the performance of the proposed approach can be improved by hyper-parameter optimization [42]. Second, the proposed approach can be improved by using complex architectures such as transformers [15] and stacked neural networks [63]. Another possible path for improvement is to train the model on an even larger data set. In the future, we consider exploiting these possibilities.

## 5. Limitations and Threats to Validity

A risk to construct validity is the selection of assessment metrics. To alleviate this threat, we use several different evaluation metrics. We use the Top-k accuracy metric as done by former studies [65, 29, 44]. We use the precision, recall, and F-measure [5] metrics for the evaluation of the proposed approach. These metrics are most generally used for the model evaluation purpose. Moreover, we evaluate the proposed approach with MRR [44, 18] metric which is a ranked based metric. Further, to show the statistical significance of the proposed approach we adopt the ANOVA statistical testing.

A risk to internal validity is the employment of the baseline methods. We re-implement the baseline approaches by following the process described in the original manuscripts. To alleviate this risk, we twofold the implementations and results. Conversely, there could be some unobserved inaccuracies. Another risk is the choice of hyper-parameters for deep learning methods. The change in training, validation or testing set or the variation in hyper-parameters may impact the performance of the anticipated method.

A threat to external validity is related to the generality of results. The data set used in this study is collected from *GitHub*, a well-known source code repositories provider. It is not necessary that the projects used in this study represent other languages or Java language source code entirely.

## 6. Related Work

In this section, we present background study on deep learning, transfer learning and source code language models. We focus on how these approaches can help improve the source code modeling by employing the transfer learning approach.

### 6.1. Source Code Modeling

Hindle et al. [29] have shown how natural language processing techniques can help in source code modeling. They provide a *n-gram* based model which helps predict the next code token in *Eclipse IDE*. Tu et al. [60], proposed a cache-based language model that consists of an *n-gram* and a *cache*. Hellendoorn et al. [28] further improved the cache-based model by introducing nested locality. Another approach for source code modeling is to use probabilistic context-free grammars(PCFGs) [7]. Allamanis et al. [3] used a PCFG based model to mine idioms from source code. Maddison et al. [41] used a structured generative model for source code. They evaluated their approach with *n-gram* and *PCFG* based language models and showed how they can help in source code generation tasks. Raychev et al.[50] applied decision trees for predicting API elements. Chan et al. [8] used a graph-based search approach to search and recommend API usages.

Recently there has been an increase in API usage [64, 35, 16] mining and suggestion. Thung et al. [59] introduced a recommendation system for API methods recommendation by using feature requests. Nguyen et al. [46] proposed a methodology to learn API usages from byte code. Hussain et al. [32] proposed GRU based model for source code suggestion and completion task (completion of a whole line of code). Allamanis et al. [3] introduced a model which automatically mines source code idioms. A neural probabilistic language model introduced in [1] that can suggest names for the methods and classes. Franks et al. [19] created a tool for Eclipse named *CACHECA* for source code suggestion using a *n-gram* model. Nguyen et al. [45] introduced an *Eclipse plugin* which provide code completions by mining the API usage patterns. Chen et al. [9] created a web-based tool to find analogical libraries for different languages.

A similar work conducted by Rabinovich et al. [48], which introduced an abstract syntax networks modeling framework for tasks like code generation and semantic parsing. Sethi et al. [56] introduced a model which automatically generate source code from deep Learning-based research papers. [4], Allamanis et al. proposed a bimodal to help suggest source code snippets with a natural language query. Recently deep learning-based approaches have widely been applied for source code modeling. Such as code summarization [33, 2, 24], code mining [61], clone detection [39], API learning [23] etc.

Recently, recurrent neural networks [68, 43, 10] has attracted much attention in various fields such as image generation [22], speech recognition [21], text classification [70] and more [53, 66]. More recently, deep learning has shown great potential for the modeling of source code [32, 51, 65, 26, 18]. Raychev et al. [51] used RNN for the purpose of code completion specifically focusing on suggesting source code method calls. Similarly, White et al. [65] applied RNN based deep neural network for source code completion task. Generally, these approaches are problem-specific which requires training from starch for a different related problem. In this work, we exploit the concept of transfer learning that transfers the learned knowledge from a pre-trained model and then fine-tunes it for a different downstream task.

### 6.2. Transfer Learning

Transfer learning as the name suggests intending to transfer knowledge (features) learned in solving one problem into another related problem. Hu et al. [30] have proposed a transfer metric learning approach for visual recognition in cross-domain datasets. Duan et al. [17] have proposed a kernel learning approach for the detection of cross-domain keyframe feature changes. Pan et al. [47] have proposed a dimensionality reduction method which uses the transfer learning approach by minimizing the distance between distributions between target and source domains. Khan et al. [36] have proposed a deep transfer learning approach for the detection of breast cancer by using pre-trained GoogLeNet, VGGNet, and ResNet. Huang et al. [31] have proposed a transfer learning-based approach for Synthetic Aperture Radar (SAR) classification with limited labeled data. Kraus et al. [38] proposed a decision support system by using deep neural networks and transfer learning for financial disclosures. In this work, we exploit the transfer learning approach for the purpose of source code modeling. Instead of using a single model for transferring knowledge, in this work we use a novel approach that combines two different recurrent neural networks into an attention learner for different source code modeling tasks.

## 7. Conclusion

In this work, we proposed a deep learning-based source code language model by using the concept of transfer learning. First, we exploit the concept of transfer learning for neural language-based source code models. Next, we presented RNN and GRU based pre-trained models for the purpose of transfer learning in the domain of source code. Both models are trained on over 13 million code tokens and do not need retraining and can directly be used for the purpose of transfer learning. We evaluated the proposed approach with the downstream task of source code suggestion. We evaluated the proposed approach extensively and compared it with the state-of-the-art models. The extensive evaluation of this work suggests that the proposed approach significantly improves the model's performance by exploiting the concept of transfer learning.

## Acknowledgments

This work was supported by the National Key R&D (grant no. 2018YFB1003902) and Qing Lan Project.

## References

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 38–49.

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A Convolutional Attention Network for Extreme Summarization of Source Code.

[3] Miltiadis Allamanis and Charles Sutton. Mining Idioms from Source Code. pages 472–483.

[4] Miltiadis Allamanis, Daniel Tarlow, and Andrew D Gordon. Bimodal Modelling of Source Code and Natural Language. *Icml*, pages 2123–2132.

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019.

[6] Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. End-to-end attention-based large vocabulary speech recognition. pages 4945–4949, 2016.

[7] Pavol Bielik, Veselin Raychev, and Martin Vechev. PHOG: Probabilistic Model for Code. *International Conference on Machine Learning (ICML'16)*, pages 2933–2942.

[8] Wing-Kwan Chan, Hong Cheng, and David Lo. Searching Connected API Subgraph via Text Phrases. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 10:1–10:11.

[9] Chunyang Chen and Zhenchang Xing. Similartech: Automatically recommend analogical libraries across different programming languages. pages 834–839, 2016.

[10] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.

[11] Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. pages 577–585, 2015.

[12] Dan C. Ciresan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. *CoRR*, abs/1202.2745, 2012.

[13] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *CoRR*, abs/1901.02860, 2019.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[16] Andrea Renika Dsouza, Di Yang, and Cristina V. Lopes. Collective intelligence for smarter API recommendations in python. *Proceedings - 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation, SCAM 2016*, pages 51–60, 2016.

[17] Lixin Duan, Ivor W Tsang, Dong Xu, and Stephen J Maybank. Domain transfer svm for video concept detection. pages 1375–1381, 2009.

[18] Dhvani Patel Abram Hindle Jose Nelson Amaral Eddie Antonio Santos, Joshua Charles Campbell. Syntax and sensibility: Using language models to detect and correct syntax errors. *3D Digital Imaging and Modeling, International Conference on*, pages 311–322, 2018.

[19] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. Cacheca: A cache language model based code suggestion tool. pages 705–708, 2015.

[20] A. Gajbhiye, S. Jaf, N. A. Moubayed, A. S. McGough, and S. Bradley. An Exploration of Dropout with RNNs for Natural Language Inference. *ArXiv e-prints*, October 2018.

[21] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. pages 6645–6649, 2013.

[22] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

[23] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep API learning. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, pages 631–642.

[24] Latifa Guerrouj, David Bourque, and Peter C. Rigby. Leveraging Informal Documentation to Summarize Classes and Methods in Context. *Proceedings - International Conference on Software Engineering*, 2:639–642, 2015.

[25] Rahul Gupta, Aditya Kanade, and Shirish Shevade. Deep Reinforcement Learning for Programming Language Correction.

[26] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. DeepFix : Fixing Common C Programming Errors by Deep Learning. *Aaai'17*, 1(Traver):1345–1351, 2017.

[27] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.

[28] Vincent J Hellendoorn and Premkumar Devanbu. Are Deep Neural Networks the Best Choice for Modeling Source Code? *Fse*, pages 763–773.

[29] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the Naturalness of Soft. *Proc. of the \nth{34} Int. Conf. on Soft. Eng.*, (5):837–847.

[30] Junlin Hu, Jiwen Lu, and Yap-Peng Tan. Deep transfer metric learning. pages 325–333, 2015.

[31] Zhongling Huang, Zongxu Pan, and Bin Lei. Transfer learning with deep convolutional neural network for sar target classification with limited labeled data. *Remote Sensing*, 9(9):907, 2017.

[32] Yasir Hussain, Zhiqiu Huang, Senzhang Wang, and Yu Zhou. Codegru: Context-aware deep learning with gated recurrent unit for source code modeling. 2019.

[33] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing Source Code using a Neural Attention Model. *Acl*, pages 2073–2083.

[34] R. Ranca M. Allamanis M. Lapata C. Sutton J. Fowkes, P. Chanthirasegaran. Autofolding for source code summarization. *IEEE Transactions on Software Engineering*, 43(12):1095–1109, 2017.

[35] Iman Keivanloo, Juergen Rilling, and Ying Zou. Spotting working code examples. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 664–675.

[36] SanaUllah Khan, Naveed Islam, Zahoor Jan, Ikram Ud Din, and Joel JP C Rodrigues. A novel deep learning based framework for the detection and classification of breast cancer using transfer learning. *Pattern Recognition Letters*, 2019.

[37] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[38] Mathias Kraus and Stefan Feuerriegel. Decision support from financial disclosures with deep neural networks and transfer learning. *Decision Support Systems*, 104:38 – 48, 2017.

[39] Balwinder Kumar and Satwinder Singh. Code Clone Detection and Analysis Using Software Metrics and Neural Network-A Literature Review. 3(2):127–132, 2015.

[40] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015.

[41] Chris J. Maddison and Daniel Tarlow. Structured Generative Models of Natural Source Code. *arXiv preprint arXiv:1401.0514*, pages 1–17.

[42] Pawel Matuszyk, Renê Tatua Castillo, Daniel Kottke, and Myra Spiliopoulou. A Comparative Study on Hyperparameter Optimization for Recommender Systems. *Workshop on Recommender Systems and Big Data Analytics*.

[43] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. 2010.

[44] Anh Tuan Nguyen, Trong Duc Nguyen, Hung Dang Phan, and Tien N. Nguyen. A deep neural network language model with contexts for source code. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 323–334.

[45] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. pages 69–79, 2012.

[46] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. Learning API Usages from Bytecode: A Statistical Approach.

[47] Sinno Jialin Pan, James T Kwok, Qiang Yang, et al. Transfer learning via dimensionality reduction. 8:677–682, 2008.

[48] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract Syntax Networks for Code Generation and Semantic Parsing.

[49] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[50] Veselin Raychev, Pavol Bielik, Martin Vechev, Andreas Krause, Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016*, (1):761–774.

[51] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14*, pages 419–428.

[52] Xin Rong. word2vec parameter learning explained. *CoRR*, abs/1411.2738, 2014.

[53] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. 2014.

[54] Milad Salem, Shayan Taheri, and Jiann-Shiun Yuan. Utilizing transfer learning and homomorphic encryption in a privacy preserving and secure biometric recognition system. *Computers*, 8(1):3, 2019.

[55] Akshay Sethi, Anush Sankaran, Naveen Panwar, Shreya Khare, and Senthil Mani. DLPaper2Code: Auto-generation of Code from Deep Learning Research Papers.

[56] Akshay Sethi, Anush Sankaran, Naveen Panwar, Shreya Khare, and Senthil Mani. DLPaper2Code: Auto-generation of Code from Deep Learning Research Papers.

[57] H. Shin, H. R. Roth, M. Gao, L. Lu, Z. Xu, I. Nogues, J. Yao, D. Mollura, and R. M. Summers. Deep convolutional neural networks for computer-aided detection: Cnn architectures, dataset characteristics and transfer learning. *IEEE Transactions on Medical Imaging*, 35(5):1285–1298, May 2016.

[58] C. Szegedy, , , P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, June 2015.

[59] Ferdian Thung, Shaowei Wang, David Lo, and Julia Lawall. Automatic recommendation of API methods from feature requests. *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, (November):290–300, 2013.

[60] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 269–280.

[61] B C Canada Va. MAPO : Mining API Usages from Open Source Repositories. pages 54–57.

[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. pages 5998–6008, 2017.

[63] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(Dec):3371–3408, 2010.

[64] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage API usage patterns from source code. *IEEE International Working Conference on Mining Software Repositories*, pages 319–328, 2013.

[65] Martin White, Christopher Vendome, Mario Linares-Vasquez, and Denys Poshyvanyk. Toward Deep Learning Software Repositories. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345.

[66] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

[67] Y. Yuan, X. Zheng, and X. Lu. Hyperspectral image superresolution by transfer learning. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 10(5):1963–1974, May 2017.

[68] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[69] W. Zhang, R. Li, T. Zeng, Q. Sun, S. Kumar, J. Ye, and S. Ji. Deep model based transfer and multi-task learning for biological image analysis. *IEEE Transactions on Big Data*, pages 1–1, 2018.

[70] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. pages 649–657, 2015.