Improving Code Generation From Descriptive Text By Combining Deep Learning and Syntax Rules

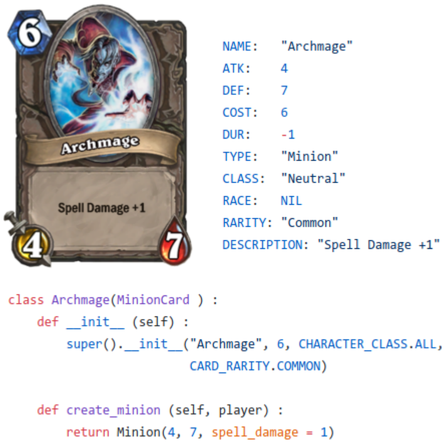
Xiangru Tang[a,b], Zhihao Wang[c], Jiyang Qi[c], Zengyang Li[a,b,*]
[a]School of Computer Science, Central China Normal University, Wuhan, China
[b]Hubei Provincial Key Laboratory of Artificial Intelligence and Smart Learning, Central China Normal University, Wuhan, China
[c]School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
xrtang@mails.ccnu.edu.cn, zhihaowang@hust.edu.cn, jyqi@hust.edu.cn, zengyangli@mail.ccnu.edu.cn

***Abstract*—Code generation is a model-driven engineering approach that enables developers to generate source code automatically and achieves extremely high development productivity. Specifically, generating code from a descriptive text reduces the time and expense of software development significantly. However, the performance of existing methods is not satisfying, since they are either of low accuracy (lack of specifics of the generated code) or too complicated (lack of efficiency in training). In this work, we proposed three novel methods by combining neural architectures and syntax rules, aiming at explicitly capturing the syntactical characteristics of target code. First, we proposed three models based on the Combination of Deep learning and Syntax rules (CDS models). Then, we evaluated CDS models with BLEU metric by comparing our models with existing methods. The results show that our models outperform existing methods for the challenging code generation task. Finally, we conducted a comparative study between the three CDS models. With further analysis we provided advice on the choice of neural architectures by considering both task accuracy and efficiency. Experimental results show that (1) there is a trade-off between speed and accuracy of the model, and (2) one of our CDS models (i.e., the CDS-POOLING model) outperforms other existing methods for the challenging code generation task.**

***Index Terms*—code generation, neural network, abstract syntax tree, encoder-decoder**

I. INTRODUCTION

Code generation is a process of generating source code from sentences that describe code functionality [1]–[3]. Firstly, generating code automatically with given descriptive sentences as constraints is significantly faster than writing the code manually. Moreover, the generated code works in an expected way and is more maintainable and extendable. In contrast, with code written manually, different developers tend to use different styles, which may lead to software errors. Thus, code generation is of great significance throughout the software development lifecycle.



```
NAME:    "Archmage"
ATK:     4
DEF:     7
COST:    6
DUR:     -1
TYPE:    "Minion"
CLASS:   "Neutral"
RACE:    NIL
RARITY:  "Common"
DESCRIPTION: "Spell Damage +1"
```

```
class Archmage(MinionCard ) :
    def __init__ (self) :
        super().__init__("Archmage", 6, CHARACTER_CLASS.ALL,
                        CARD_RARITY.COMMON)

    def create_minion (self, player) :
        return Minion(4, 7, spell_damage = 1)
```

Fig. 1. An example of Hearth Stone dataset

However, code generation is challenging because its output must abide by the syntax rules strictly, and the arithmetic speed for the purpose of practicability should also be considered. Considerable effort has been invested in code generation, which results in several types of code generation approaches. However, they often fail to generate executable code correctly because they hardly capture complicated code structures.

Recently, deep learning approaches based on neural networks have shown significant performance improvement on many artificial intelligence tasks. Public datasets speed up the development of the code generation area. One of high-quality datasets is Hearth Stone [1], which aims to generate Python code based on Hearth Stone card description. An example is shown in Fig 1, in which a piece of code lies along the bottom. Each card is identified by ten attributes (e.g., name and attack) and has a text box to describe the effect of the card.

In this work, we proposed three models based on the Combination of Deep learning and Syntax rules (CDS models). This work involves some neural architectures (e.g, self-attention network, CNN) which have already resulted in significant progress in the field of natural language processing. Meanwhile, considering the essential difference between

---

* Corresponding author.

This work is partially supported by the National Natural Science Foundation of China (NSFC) under the Grant Nos. 61702377 and 61773175.
DOI reference number: 10.18293/SEKE2019-170

[1] available at https://github.com/deepmind/card2code

code and natural language, our work also takes syntax rules into account. Specifically, we represented the code in a tree structure. In addition, considering the poor performance of Recurrent Neural Network in existing methods [1]–[4], we adopted the pooling operation and self-attention mechanism in code generation tasks. Moreover, we explored the strengths and weaknesses of our CDS models (i.e., CDS-POOLING, CDS-CNN, and CDS-SAN models).

The key contributions of our study are described as follows:

a) We proposed three novel code generation frameworks, based on the Combination of Deep learning and Syntax rules (CDS models). To the best of our knowledge, we are the first to use Transformer model (self-attention network) and pooling operation in code generation. And our high-performance models lead to a significant improvement of BLEU score.

b) We conducted a comparative study between three models we proposed. We also thought that researchers need a comprehensive analysis of the task and should adopt simple and effective networks, when using deep learning methods.

II. RELATED WORK

Generating code in software engineering has a long history. Some early works focus on domain specific languages [5], [6]. For general-purpose code generation, such as [7], which tries to generate the code for parsing input documents. It was presented that data driven methods instead of instead of manual methods are used in manufacturing model, and code could be generated automatically.

There are some early works using syntax rules only. (1) Parser Generation approach: some tools such as template engine [8] were used to automatically generate parser, but it is too complicated and cannot cover every scenario. (2) Model Driven approach: an entire application or just its skeleton is generated. (3) Database-related approach: usually the programmer defines a database schema, from which entire CRUD (Create, Retrieve, Update, and Delete) operations or just the code to handle the database can be generated. (4) Metaprograming approach: some researchers developed a new language which could manipulate another piece of source code; it means that the source code is just another data structure that can be manipulated [9]. (5) Retrieval-based approach: some researchers leveraged subtree retrieval mechanism, which can explicitly output existing code examples. [10].

Recently, neural networks are introduced to code generation. Encoder-Decoder architecture has shown good performance in practical applications, such as machine translation, dialogue system, and image captioning. The encoder processes an input sequence $x = (x_1, ..., x_m)$ of m elements and returns state representations $z = (z_1, ..., z_m)$. The decoder takes z and generates the output sequence $y = (y_1, ..., y_n)$ left to right. To generate output y_{i+1}, the decoder computes a new hidden state h_{i+1} based on the previous state h_i, an embedding g_i of the previous target language word y_i, as well as a conditional input c_i derived from the encoder output z. Based on this generic formulation, various encoder-decoder architectures have been proposed.

Recent works on neural machine translation mostly base on sequence-to-sequence models are worth referring for us. [11], [12] and many works following adopt attention based models to get better performance with longer sentences. [13] uses CNNs to build sequence-to-sequence model which is faster and allows to discover compositional structure. Transformer is proposed in [14] first. Although originally used in translation tasks, self-attentive feed-forward sequence models have been shown to achieve impressive results on many sequence modeling tasks [14]–[16]. For code generation task, some researchers attempted to adopt sequence-to-sequence models [1] or models based on abstract syntax tree [2]–[4], [17] to get valid program. The decoder of neural network models above are all based on RNNs except for [17] which uses CNNs to capture information.

Moreover, some researchers found that much simpler word-embedding-based architectures exhibit impressive performance, compared with more-sophisticated models using RNN or CNN [14]. There are some related works which introduced pooling to some tasks. [18]–[20] show that average pooling can obtain impressive accuracy on both sentence and document-level sentiment analysis, factoid question-answering and text classification tasks with much less training time than competing methods.

III. ARCHITECTURE

We first define the code generation task as below:

Given a descriptive text q, our target is to generate code (e.g., Python code), specifically in an AST a format. In this paper, we start with the syntactic code generation model proposed in [2]. It focuses on generating AST from text, and then converting it to concrete code. Formally, our goal is to find a best generated AST $\hat{a}$ as Eq.(1).:

$$\hat{a} = \arg\max_{a} p(a|q) \tag{1}$$

$$p(a|q) = \prod_{t=1}^{T} p(y_t | y_{<t}, q) \tag{2}$$

where $y_{<t}$ represents $y_1 ... y_{t-1}$ and T is the number of total time steps.

Our CDS models can be divided into three dimensions: pooling based, CNN based, and self-attention network based. We train these three models independently, with input of (a) predicted structure of AST, (b) name of variables, and (c) name of functions containing syntax information.

A. CDS-POOLING

Word embeddings can learn a lot from rich unstructured descriptive text, which are widely adopted as building blocks in the area of Natural Language Processing (NLP). Word embeddings can cluster similar words in semantical level by representing each word as a fixed-length vector and encoding the linguistic regularities and patterns implicitly. Thus, our

CDS-POOLING model is closely related to Deep Averaging Network, which demonstrates that the average pooling operation achieves tremendous results for some NLP tasks. Pooling operations capture high level semantic features and low level word characters information, just same as a method of information fusion. Moreover, we explored different pooling operations, rather than only average-pooling.

Average Pooling: Average pooling computes the element-wise average over word-vectors for the descriptive text, which average the value of K dimensions for all word embedding. Thus, Average Pooling is able to obtain a representation z with the same dimension as the embedding itself.

$$z = \frac{1}{L}\sum_{i=1}^{L} v_i \tag{3}$$

Max Pooling: Max Pooling extracts the most salient features from every word-embedding dimension, by taking the maximum value along each dimension of the word vectors.

$$z = \text{Max-pooling}(v_1, v_2, ..., v_L) \tag{4}$$

where the j-th component of z is the maximum element in the set $v_{1j}, ..., v_{Lj}$, where v_{1j} is, for example, the j-th component of v_1. With this pooling operation, those words that are unimportant or unrelated to our task will be ignored in the encoding process.

Hierarchical Pooling: Both average- and max- pooling do not take word order into consideration, which could be useful for code generation tasks. Thus, we also proposed a hierarchical pooling layer, where the two abstracted pooling features are concatenated together to represent the sentence embeddings. However, Hierarchical Pooling learns fixed-length representations for the n-grams that appear in the corpus, rather than just capturing their occurrences via count features, which is more suitable for code generation.

For all CDS-POOLING variants above, there is no additional network to extract features. We just apply max-poling after embedding, and then attention mechanism is used to acquire information between features. Finally, two layers of fully connect neural network and a softmax layer is applied to get the classification result. Thus, CDS-POOLING model quickly captures only intrinsic word-embedding information for code generation. In experiments, we proved that the CDS-POOLING model significantly promote the precision and accelerate the calculation speed.

B. CDS-CNN

The Convolutional Neural Network (CNN) [21] is another strategy extensively employed encode text sequences. Convolution operation can be described as using filters $\mathbf{w} \in \mathbb{R}^{hk}$ to capture word-level features. For each kernel F, a convolution operation uses a $D \times K$ slide. First, We applied the embedding mentioned above as input, which is a tensor of $D \times L$ dimensions. L represents the length of descriptive sentence and D is the embedding dimensions. Then, for each step, we summed the weighted value in filters and applied nonlinear activation operation to obtain the filter. Finally, the filters become a vector which represents the output. In practice, several layers are applied to capture the hierarchical information.

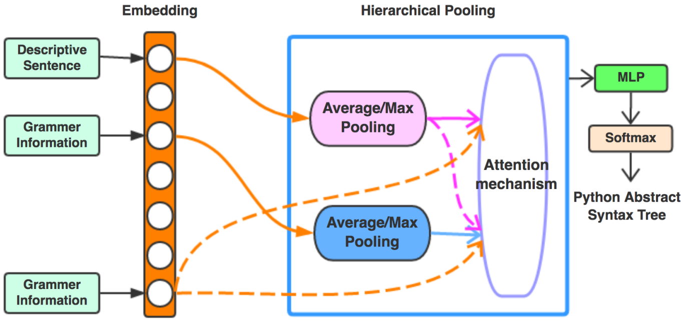


Fig. 2. Simple Word-Embedding-Based Model

We learned from [22] and applied residual structure in our model. What's more, we also conducted batch normalization [23] to speed up the training process and improve the model performance. Our CDS-CNN model is showed in Fig 3. The embedding input has the shape of $N \times L \times D$. Our designed interact-CNN takes into account the relationship between natural language and semantic structure. Moreover, we use the concatenation operation to mix the information from natural language and AST together. But attention is needed for our operation of transposing the length-dimension and embedding-dimension to fully mix the information. At the final part of interact-CNN, we applied max-pooling to get the hierarchical information. Through the interact-CNN part, the shape of our tensor is reduced to $N \times D$. At the final part of Fig 3, there are multilayer perceptron (MLP) and softmax layer to generate final AST. CNN can be fully trained in parallel to better exploit the floating-point computation capacity of GPU, compared with RNN in the train step.

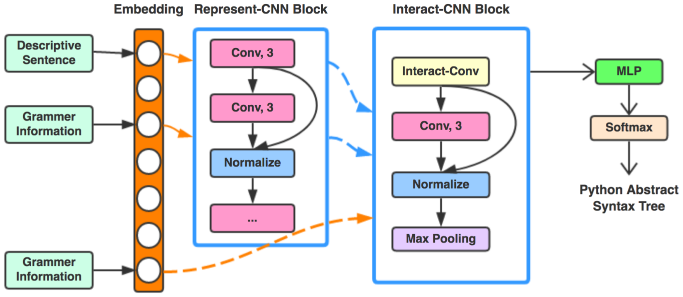


Fig. 3. Encoder-Decoder CNN

C. CDS-SAN

Transformer architecture [14] is based entirely on attention mechanisms and achieves best performance in the neural machine translation. In this work, we adopted the self-attention network architecture, which is a modified version of Transformer for two reasons. Firstly, it could be trained fast for its parallelizable structure, while traditional RNN model is computational and time consuming due to its recurrent structure. Secondly, it naturally constructs the long-term dependence via the attention mechanism, see Eq 4. It implies that Transformer architecture learns non-local dependencies between tokens regardless of the distance between them.

The self-attention network (SAN) is a special case of the attention mechanism, which models the dependencies between tokens from the same sequence [14]. In our work, the self-attention layer aims to create the embedding representations of the original text input. Specifically, given the text $H = (h_1, h_2, ..., h_L)$ carrying the semantics of the original review along all time steps from the encoder, self-attention network first yields a weight matrix $A^{enc} = ({a_1}^{enc}, {a_2}^{enc}, ..., {a_L}^{enc}$, computed as $A^{enc} = softmax({w_2}^{enc} tanh({w_1}^{enc} H^T))$, where ${w_1}^{enc}$ is a parameter vector and ${w_2}^{enc}$ is a parameter matrix. $Softmax$ function is used to normalize the attention weights.

Since embeddings represent linguistic context information weakly, we use self-attention network to encode input for a better representation and drop out the decode part.

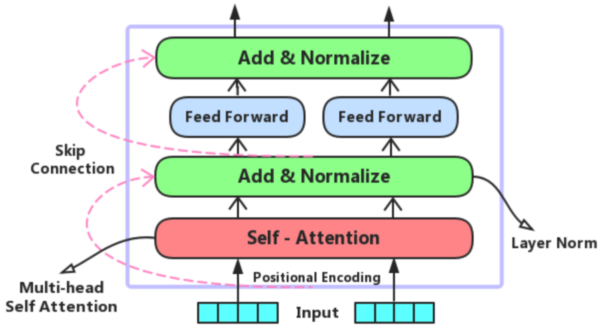


Fig. 4. Self-attention Network

IV. EXPERIMENT

Our experiment aims to answer the following research questions (RQs):

RQ1:How do CDS models perform?

Our work intends to make a comparative study of the differences between our CDS models and existing state-of-the-art methods (e.g., LPN [1], SNM [2], ASN [3], and SEQ2TREE [4] all mentioned above in section II).

RQ2:Can we find a trade-off between training speed and accuracy of the result?

Most models with more expressive composition functions perform well? Speed and accuracy, which matters more in reality? This work includes a rigorous evaluation regarding the added value of sophisticated composition functions.

A. Dataset

We used Hearth Stone dataset introduced by [1] as a practical code-generation application. It has 655 cards in total, 533 cards for training, 66 cards for validating, and 66 cards for testing. Each card is identified by ten attributes (e.g., TYPE and ATK) and has an functional describe in the text box. Code implementations of these cards implement the game logic and card effects once per turn.

B. Evaluation Metrics

We used billingual evaluation understudy score (BLEU) [24] as the evaluation metric in our experiment. Although BLEU is developed for translation tasks originally, it can be used to evaluate programs in this work through measuring how close the generated code is to the ground truth code in terms of n-grams.

To obtain the BLEU score, we computed modified $n\text{-}grams$ precision (p_n) first. The modified $n\text{-}grams$ prediction is computed as follows. All candidate $n\text{-}grams$ counts and their corresponding maximum reference counts are collected. In the code generation task, the reference is executable code.

$$p_n = \frac{\sum\limits_{C \in \{Candidates\}} \sum\limits_{n\text{-}gram \in C} Count_{clip}(n\text{-}gram)}{\sum\limits_{C' \in \{Candidates\}} \sum\limits_{n\text{-}gram' \in C'} Count(n\text{-}gram')} \tag{5}$$

Then, we define BP to penalize the generated program shorter than ground truth. c is the length of generated code and r is the reference code length.

$$\text{BP} = \begin{cases} 1 & if \ c > r \\ e^{1-r/c} & if \ c \leq r \end{cases} \tag{6}$$

Finally, we get

$$\text{BLEU-N} = \text{BP} \cdot \exp(\sum_{n=1}^{N} w_n \log p_n) \tag{7}$$

In our experiment, we used $N = 4$ and uniform weights $w_n = 1/N$.

Additionally, we also calculated the ACCURACY score which represents the percentage of 66 test codes that can be executed and absolutely correct.

C. Experiment Hyperparameters

For the CDS-POOLING model, our character embedding has 256 dimensions. We used a dropout rate of 0.7; for the CDS-CNN model, we set the embedding size as 128 to concentrated information and hidden-layer size as 128 to extract features. The dropout rate is 0.5 because CNN model is more complicated than pooling. For the CDS-SAN model, we followed the base Transformer [14], with 4 encoder layers of 128 hidden dimensions, and 8 attention heads per-layer. And the CNN filters followed the specifications of [21]. The dropout rate is also 0.5, same as CDS-CNNẆe optimized these models with Adam with default hyperparameters and set batch size as 64. It took us 9 hours to train a pooling model on a NVIDIA Tesla P100. We used a beam search for generating the program, and computed BLEU scores to measure performance on the testing set.

TABLE I

EXPERIMENTAL RESULTS

Model	**ACCURACY**	**BLEU**
SEQ2TREE	1.5	53.4
LPN	6.1	67.1
SNM	16.2	75.8
ASN	18.2	77.6
CDS-POOLING	15.6	**78.9**
CDS-CNN	**19.7**	77.0
CDS-SAN	16.7	77.8

D. Experimental results

Table I presents the results of our CDS models in code generation, in comparison with exisiting methods, e.g., LPN [1], SNM [2], ASN [3], and SEQ2TREE [4], which are the state-of-the-art models in the code generation area. The results show that adopting CDS-POOLING to capture the syntax rules yields a better performance with much less training time than other methods. Besides, compared with RNN, CNN is better suited to capturing the structural information of long sentences. Finally, theself-attention network based model also outperforms existing methods.

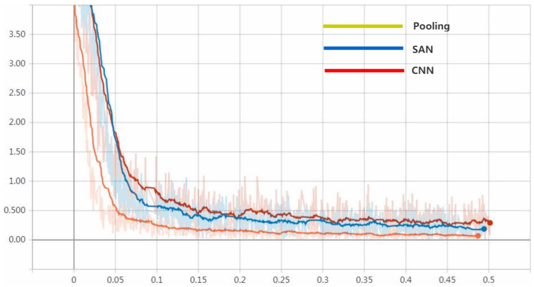


Fig. 5. Loss Function Over Time

E. Result Analysis and Case study

The results demonstrate that CDS-POOLING is particularly better than any other existing methods in both speed and efficiency. As for CDS-SAN, when we made this model more complicated, the ACCURACY score fell but BLEU score rose, which means CDS-SAN can handle the detail better. CDS-CNN model is good at generating more completely correct code.

As mentioned in [3], Hearth Stone contains classes with similar structures, thus the code generation task can predigest into the generation of tree-like AST and what we need to do is to fill in tokens with certain variables and values. Nevertheless, certain errors must occur because that Hearth-Stone's code contains complicated logic, which result in a low accuracy [2]. To see more details, we presented an example of code we generated by CDS-POOLING in table II. It illustrates that our model can handle complicated code syntax effectively. However, our generated code does not absolutely match the reference code, but it is correct in general and can be executed.

To be more specific, the reason why the CDS-POOLING has such an impressive result is that pooling operation builds an information map to achieve a downsampling-process, which throws away indifferent information. In contrast, the CDS-CNN model tends to generate a structural correct code, which leads to a higher ACCURACY (more absolutely correct codes) but a similar BLEU compared with previous works.

TABLE II

THE COMPARING OF GENERATED CODE AND REFERENCE CODE

Descriptive Text:

```
annoy-o-tron name_end 1 atk_end 2 def_end 2 cost_end -1 dur_end
minion type_end neutral player_cls_end mech race_end
common rarity_end b taunt /b nl b divine shield /b
```

Generated Code:

```
class AnnoyoTron(MinionCard ) :
    def __init__ (self) :
        super().__init__("Annoy-o-Tron", 2,
          CHARACTER_CLASS.ALL, CARD_RARITY.COMMON,
          minion_type = MINION_TYPE.MECH, divine_shield = True)
    def create_minion (self, player) :
        return Minion(1, 2, taunt = True, divine_shield = True)
```

Reference Code:

```
class AnnoyoTron(MinionCard ) :
    def __init__ (self) :
        super().__init__("Annoy-o-Tron", 2,
          CHARACTER_CLASS.ALL, CARD_RARITY.COMMON,
          minion_type = MINION_TYPE.MECH)
    def create_minion (self, player) :
        return Minion(1, 2, divine_shield = True, taunt = True)
```

V. DISCUSSION AND ANALYSIS

RQ1: How do CDS models perform?

The results demonstrate that CDS-POOLING is particularly better than any other existing methods in both speed and efficiency. When we made model more complicated, the ACCURACY result fell, but the BLEU score rose (see CDS-SAN), which means that it can handle the details better. And it clarifies CDS-CNN model is skilled at generating more completely correct code. Results also illustrate that simple pooling operation (CDS-POOLING) is surprisingly effective at representing longer sequence. CDS-POOLING figures over all elements of th tensor, and CDS-POOLING can be fully trained in parallel to better exploit the floating-point computation capacity of GPU, compared with RNN in the train step. In addition, CDS-POOLING is more amenable to optimization because the number of non-linearities unit is fixed, moreover, it not be affected by the text input's length. The max-pooling is working as follows: we can consider the embedding size used as 256 represent 256 kinds of semantics, and the max-pooling operator is to extract the semantics in sentences.

RQ2: Can we find a trade-off between training speed and accuracy of the result?

The experiment demonstrated that CDS-POOLING is the most effective approach outperforming other complicated models. This also tells us that a simple and effective method should be a prevailing concern. Additionally, it helps researchers to avoid the blind use of deep learning algorithms when solving software engineering problems.

Considering the nature of this problem, more sophisticated models reveal outstanding results but are excessively computationally expensive, because they need to optimize thousands of parameters, e.g., RNN or CNN. On the contrary, maybe some simpler models can be robust, which only compute the sentence embedding by simply adding or averaging operation over the word embedding, just as our CDS-POOLING. But that also means, such a simple pooling operation does not take word-order information into account. However, pooling operation has the advantage of having significantly fewer parameters, which means it can train much faster and obtain a equally good precision, comparing to RNN or CNN. Thus, there is a trade-off between training speed and efficiency.

VI. CONCLUSION AND FUTURE WORK

In this work, we are the first to introduce pooling operation, fully convolutional model, and self-attention network for code generation tasks to the best of our knowledge. Furthermore, analysis shows that the pooling based model is the most efficient. Thus, we achieved state-of-the-art results in the code generation task. Specifically, on Hearth Stone dataset we outperformed all the previous methods by 78.9 BLEU. In addition, we provided some advice to researchers that they must consider practical reality of target tasks, and do not fall into the trap of using complicated deep learning models blindly. In conclusion, our work has theoretical significance and practical value in the field of software engineering.

In the future, we plan to adopt more models in code generation tasks, aiming at seeking ways to improve the performance. However, there is a syntactic difference between various program languages. Thus, we also plan to apply CDS models into more code generation datasets.

REFERENCES

[1] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kočiský, F. Wang, and A. Senior, "Latent predictor networks for code generation," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2016, pp. 599–609.

[2] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2017, pp. 440–450.

[3] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2017, pp. 1139–1149.

[4] L. Dong and M. Lapata, "Language to logical form with neural attention," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2016, pp. 33–43.

[5] N. Kushman and R. Barzilay, "Using semantic unification to generate regular expressions from natural language," in *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2013, pp. 826–836.

[6] M. Raza, S. Gulwani, and N. Milic-Frayling, "Compositional program synthesis from natural language and examples." in *IJCAI*, Q. Yang and M. Wooldridge, Eds. AAAI Press, pp. 792–800.

[7] T. Lei, F. Long, R. Barzilay, and M. Rinard, "From natural language specifications to program input parsers," in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2013, pp. 1294–1303.

[8] M. M.-H. Tso, "Context-sensitive template engine," Jul. 4 2000, uS Patent 6,085,201.

[9] T. Veldhuizen, "Method and apparatus for generating inline code using template metaprograms," Nov. 10 1998, uS Patent 5,835,771.

[10] S. A. Hayati, R. Olivier, P. Avvaru, P. Yin, A. Tomasic, and G. Neubig, "Retrieval-based neural code generation," *CoRR*, vol. abs/1808.10025, 2018.

[11] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014.

[12] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2015, pp. 1412–1421.

[13] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional sequence to sequence learning," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 1243–1252.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 5998–6008.

[15] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018.

[16] N. Kitaev and D. Klein, "Constituency parsing with a self-attentive encoder," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2018, pp. 2676–2686.

[17] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural CNN decoder for code generation," *CoRR*, vol. abs/1811.06837, 2018.

[18] M. Iyyer, V. Manjunatha, J. Boyd-Graber, and H. Daumé III, "Deep unordered composition rivals syntactic methods for text classification," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, 2015, pp. 1681–1691.

[19] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*. Association for Computational Linguistics, 2017, pp. 427–431.

[20] D. Shen, G. Wang, W. Wang, M. Renqiang Min, Q. Su, Y. Zhang, C. Li, R. Henao, and L. Carin, "Baseline needs more love: On simple word-embedding-based models and associated pooling mechanisms," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2018, pp. 440–450.

[21] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2014, pp. 1746–1751.

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

[23] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015.

[24] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 2002.