Pseudocode to Source Code Translation with Transformer Model

Yuguang Duan (yduan38), Yiyin Shen (yshen82)

May 7, 2022

1 Abstract

We performed pseudocode to source code translation with the transformer model, which achieved state-of-the-art performance in various natural language tasks. We used the SPoC dataset developed by Kulal et al. (2019), with human-authored pseudocode as the source language and C++ code as the target language. We tried different model configurations during the training and compared their translation compilable and passed rate at both line and program levels on the test set. We also try to use different sizes of beam search in the translation process. Although we achieve a reasonable compilable rate (73%) and passed rate (51.6%) at the line level, the compilable rate (9.43%) and passed rate (1.1%) are on the lower end at the program level. One possible reason is that our dataset is too small, but the transformer model requires a large dataset to achieve better performance.

2 Related Work & Motivation

Kulal et al. [12] collected the SPoC (Search-based Pseudocode to Code) dataset, which contains C++ code, corresponding human-authored pseudocode, and test cases. They developed a Seq2Seq translation model with an LSTM encoder and decoder and an attention-based copying mechanism to translate pseudocode to C++ code. The model reaches a line-level passed rate of around 90% and a program-level passed rate of 44.7% (the definition of passed rate is presented in the evaluation section). Our project is mainly

inspired by this work for two reasons. First, the dataset is thoughtfully collected and organized. Second, even though the LSTM model has already achieved reasonably good performance, we are curious about the performance of other languages models' performance on the same task. The development of transformer model [17] is a recent groundbreaking improvement in language modeling. Transformer is a new architecture utilizing self-attention and position embedding to enable parallel training of millions of parameters on large datasets. Therefore, the goal of our project is to evaluate the performance of the transformer model on pseudocode and source code translation using the SPoC dataset.

Following Kulal et al., several studies used the SPoC dataset as their benchmark. They either improved upon the originally proposed model or adopted new models. The model proposed by Kulal et al. used beam search to produce a list of pseudocode translation candidates line by line, concatenate these line candidates to produce translation programs, and then exhaustively test generated programs. This ignores the dependencies between code lines and often results in missing parentheses and inconsistent variable names. To counter that, Zhong et al. [19] proposed a hierarchical beam search algorithm that incorporates syntactic and symbol table constraints. The syntactic constraints are grammatical rules to ensure correct parentheses and indentation, especially in high-level control structures such as if-else blocks, for and while loops, and function declarations. The symbol table ensures variable names are consistent within their scope. With the new hierarchical beam search, the model reaches a 55.1% passing rate on the SPoC dataset with far fewer beam search candidates. Acharjee et al. [1] used specialized word embedding techniques and a similar LSTM encoder-decoder model, but reach an amazing 88.7% passed rate.

3 Dataset

We used the SPoC (Search-based Pseudocode to Code) dataset developed by Kulal et al. [12] for training, evaluation, and testing. This dataset contains 18,356 programs with human-authored pseudocode and test cases. The programs and test cases were both scrapped from codeforces.com: the programs were accepted C++ solutions to easy-level competitive programming problems. First, programs with constructs having difficulties obtaining consistent pseudocode annotation were filtered out. Examples of such constructs are

macros, classes, structs, templates, switch statements, and mallocs. Then each program was decomposed and grouped into code lines with high-level descriptions. For example, "for (int i = 0; i < n; i++)" and "cin >> x[i];" were grouped into one code line and translated as "read n values into x". Finally, 59 crowd-workers were recruited from Amazon Mechanical Turk to annotate pseudocode for the processed code lines. This is valuable because most pseudocode from prior datasets are generated by rule-based heuristic templates, which contain a lower information content and are more coarse-grained than human-authored pseudocode. Table 1 demonstrates a source code program and its pseudocode program in the dataset.

Source Code	**Pseudocode**
int main() {	NA
int n;	create integer n
string str, s, ans, min = "z";	create strings str, s, ans, min with min = "z"
cin >> str >> n;	read str read n
for (int i = 0; i ¡ n; i++) {	for i = 0 to n exclusive
cin >> s;	read s
if (s == str) {	if s is str
min = str;	set min to str
break;	break loop
}	NA
if ((s.find(str) == 0) && (s < min)) min = s;	if first index of str in s is 0 and s is less than min, set min to s
}	NA
if (min == "z")	if min is "z"
cout << str << endl;	print str print newline
else	else
cout << min << endl;	print min print newline
return 0;	NA
}	NA

Table 1: Source code program vs. corresponding pseudocode program in the dataset

Table 2 lists statistics of the train, evaluation, test sets. Note we found some programs in the dataset don't compile or pass all the test cases and we excluded such programs in evaluation.

	Train Set	Evaluation Set	Test Set
Problems	330	89	100
Programs	11925	2010	1025
Lines of pseudocode	181862	19180	15183

Table 2: Statistics of the train, evaluation, test sets

4 Transformer Model

Most sequence-to-sequence neural network models have an encoder-decoder structure [3, 7, 16]. The encoder converts a series of symbol representations $(x_1, ..., x_n)$ into a series of continuous representations $z = (z_1, ..., z_n)$. The decoder then outputs an output sequence of symbols $(y_1, ..., y_m)$ one element at a time, given z. The model is auto-regressive at each step [10], using the previously created symbols as additional input when producing the next. The transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves in the first figure of Figure 1, respectively.

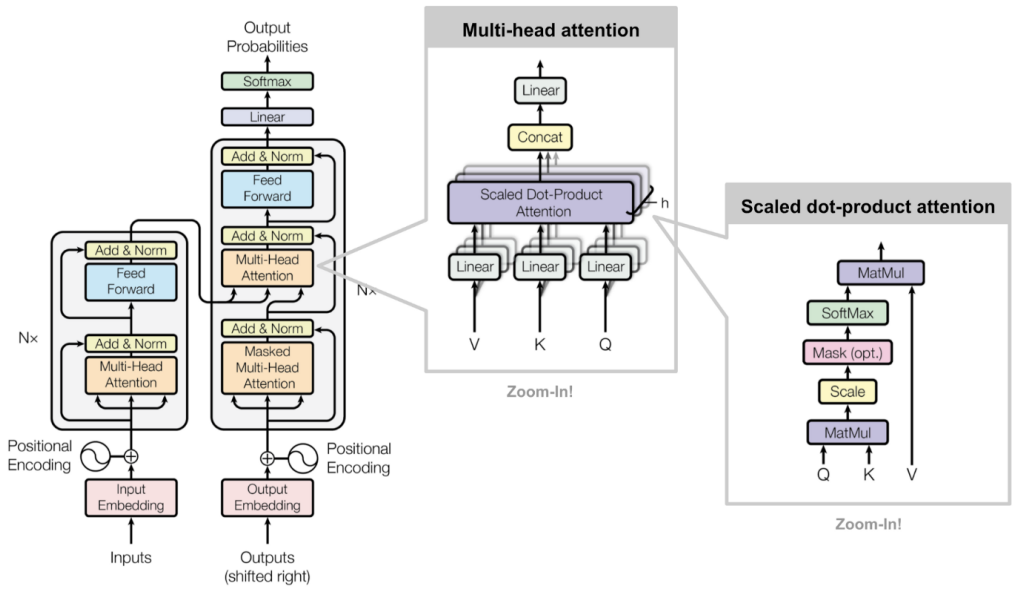

Figure 1: The architecture of Transformer

4.1 Encoder and Decoder Stacks

Encoder: The encoder is made up of $N = 6$ identical layers stacked on top of each other. Each layer is divided into two sub-layers. One is a multi-head self-attention mechanism, and the other is a simple, position-wise fully connected feed-forward network. Around each of the two sub-layers, a residual connection is employed [11] followed by layer normalization [2]. That is, each sub-output layer's output is $LayerNorm(x + Sublayer(x))$, where $Sublayer(x)$ is the function that the sub-layer implements. All sub-layers in the model, as well as the embedding layers, produce outputs of size $d_{model} = 512$ to facilitate these residual connections.

Decoder: The decoder is also made up of $N = 6$ identical layers. The decoder inserts a third sub-layer, which performs multi-head attention over the encoder stack's output, in addition to the two sub-layers in each encoder layer. Residual connections are used around each of the sub-layers, similar to the encoder, followed by layer normalization. The decoder stack's self-attention sub-layer is also modified to prevent positions from attending to subsequent positions. This masking, together with the fact that the output embeddings are offset by one position, ensures that predictions for position i can only rely on known outputs at positions less than i.

4.2 Attention

An attention function can be defined as a function that maps a query and a set of key-value pairs to an output, with the query, keys, values, and output all being vectors. The output is computed as a weighted sum of the values, with the weight assigned to each value determined by the query's compatibility function with the corresponding key.

4.2.1 Scaled Dot-Product Attention

The particular attention used in the Transformer model is called "Scaled Dot-Product Attention" (Figure 1). The input is made up of queries and keys of dimension d_k, as well as values of dimension d_v. The dot products of the query with all keys are computed, and then each is divided by $\sqrt{d_k}$, and a softmax function is applied to the values to obtain the weights.

In practice, the attention function on a set of queries is computed at the same time, which is packed into a matrix Q. The keys and values are also

crammed into matrices K and V. The output matrix is computed as follows:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

The two most commonly used attention functions are additive attention [3], and dot-product (multiplicative) attention. Except for the scaling factor of $\frac{1}{\sqrt{d_k}}$, dot-product attention is identical to our algorithm. The compatibility function is computed by additive attention using a feed-forward network with a single hidden layer. While the theoretical complexity of the two approaches is similar, dot-product attention is much faster and more space-efficient in practice because it can be implemented using highly optimized matrix multiplication code. While the two mechanisms perform similarly for small values of d_k, additive attention outperforms dot product attention without scaling for larger values of d_k [4]. It's possible that as d_k increases, the magnitude of the dot products increases, pushing the softmax function into regions with extremely small gradients 4. To compensate for this effect, the dot products are scaled by $\frac{1}{\sqrt{d_k}}$.

4.2.2 Multi-Head Attention

Instead of performing a single attention function on d_{model}-dimensional keys, values, and queries, linearly projecting the queries, keys, and values h times with different, learned linear projections to d_k, d_k, and d_v dimensions was more beneficial. Then the attention function is performed in parallel on each of these projected versions of queries, keys, and values, yielding d_v -dimensional output values. These are concatenated and projected again, yielding the final values shown in Figure 1.

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O$$

where

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in R^{d_{model}d_k}$, $W_i^K \in R^{d_{model}d_k}$, $W_i^V \in R^{d_{model}d_v}$ and $W^O \in R^{hd_vd_{model}}$. We use default $h = 8$ parallel attention layers, or heads, in this work for baseline. We also use

default $d_k = d_v = d_{model}/h = 64$ for each of these. The total computational cost is comparable to that of single-head attention with full dimensionality due to the reduced dimension of each head.

4.2.3 Applications of Attention in the Model

The Transformer uses multi-head attention in three different ways:

- The queries in "encoder-decoder attention" layers come from the previous decoder layer, and the memory keys and values come from the encoder output. This allows each decoder position to attend to all positions in the input sequence. This is similar to the typical encoder-decoder attention mechanisms found in sequence-to-sequence models like [3, 18, 9].

- Layers of self-attention are present in the encoder. All of the keys, values, and queries in a self-attention layer come from the same source, in this case, the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the encoder's previous layer.

- Similarly, decoder self-attention layers allow each position in the decoder to attend to all positions in the decoder up to and including that position. To keep the auto-regressive property, leftward information flow must be prevented in the decoder. We do this within scaled dot-product attention by masking out (setting to -∞) all values in the softmax's input that correspond to illegal connections. See Figure 1.

4.3 Position-wise Feed-Forward Networks

Aside from attention sub-layers, each layer in our encoder and decoder contains a fully connected feed-forward network that is applied to each position separately and identically. This is made up of two linear transformations separated by a ReLU activation.

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

While the linear transformations are the same across all positions, the parameters vary from layer to layer. Another way to put it is that this is two

convolutions with kernel size 1. The dimensionality of the input and output is $d_{model} = 512$, and the dimensionality of the inner-layer is $d_{ff} = 2048$.

4.4 Embeddings and Softmax

We use learned embeddings, as in other sequence transduction models, to convert input and output tokens to vectors of dimension d_{model}. To convert the decoder output to predicted next-token probabilities, we also use the standard learned linear transformation and softmax function. Similarly to [15], we share the same weight matrix between the two embedding layers and the pre-softmax linear transformation in our model. We multiply those weights by $\sqrt{d_{model}}$ in the embedding layers.

4.5 Positional Encoding

Although the Transformer model contains no recurrence or convolution, some information about the relative or absolute position of the tokens in the sequence is injected in order for the model to use the order of the sequence. To that end, "positional encodings" are added to the input embeddings at the encoder and decoder stack bottoms. The positional encodings and embeddings have the same dimension d_{model}, so the two can be added. There are numerous positional encodings, both learned and fixed [9].

In the Transformer, sine and cosine functions of different frequencies are used for positional encodings:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

where pos represents the position and i represents the dimension. That is, each positional encoding dimension corresponds to a sinusoid. The wavelengths are arranged in a geometric progression from 2π to 100002π. This function allows the model to easily learn to attend to relative positions, because PE_{pos+k} can be represented as a linear function of PE_{pos} for any fixed offset k.

5 Experiments

5.1 Training

We first tokenize the pseudocode and C++ code by space. In particular we add a token "*bos*" at the beginning of each line and a token "*eos*" at the end of each line.

For training, we use a baseline model with default parameters according to [17], as shown in Table 3. We then try to manipulate three parameters to see if they can improve the model performance: number of epochs, number of heads, and if projection weights of each head $W_i^Q, W_i^K, W_i^V, W_i^O$ is shared or not. Table 4 listed model names and their tuned parameters.

Parameter	Default Value
d_{model}	512
d_k	64
d_v	64
d_{ff}	2048
number of heads	8
number of layers	6
dropout	0.1
learning rate	0.0002
max epochs	10
batch size	128
max source sequence length	128
max target sequence length	128
warm-up steps	4000

Table 3: Default parameters in the Transformer

5.2 Evaluation

Let $c_1, \ldots, c_m$ be the code lines of original program and $p_1, \ldots, p_m$ be the corresponding annotated pseudocode lines. For each pseudocode line $p_i, 1 \leq i \leq m$, we use beam search to generate a ranked list of k (beam size) translation candidate lines, denoted $t_{i,1}, \ldots, t_{i,k}$. We then replace the original code line c_i with the translation line $t_{i,j}, 1 \leq j \leq k$. We call the program with the

Model Name	Parameter
ep5	baseline model with epoch = 5
ep10	baseline model with epoch = 10
h_ep5	model with head = 10 and epoch = 5
h_ep10	model with head = 10 and epoch = 10
s_ep5	model with shared weight and epoch = 5
s_ep10	model with shared weight and epoch = 10

Table 4: Model's names and tuned parameters

replaced translation line $R_{i,j}$. A translation line $t_{i,j}$ is considered **compaliable** if its replaced program $R_{i,j}$ is still compaliable. A translation line $t_{i,j}$ is considered **passed** if its replaced program $R_{i,j}$ passes all test cases. A pseudocode line p_i is considered **line-level compaliable** if any translation line $t_{i,j}$ is compaliable and p_i is considered **line-level passed** if any $t_{i,j}$ is passed. Thus, one of the most important metrics of this project is line translation compaliable and passed rate.

If a passed translation line can be found for all pseudocode lines, it will enter the next stage of evaluation. A list of ranked translation lines will be generated by the previous stage. A passed translation line is denoted as $pt_{i,j}$, where i represents the line number and j represents the rank of this passed candidate. The translation program T will be generated by concatenating all first-ranked passed translation lines. This translation program T will then be compiled, if a compilation error is pointing towards a passed translation line $pt_{i,j}$, the line will then be replaced by the next-in-line passed translation line $pt_{i,j+1}$. This process will be repeated until a translation program T is compilable. Then this translation program is deemed **program-level compilable**. Further, if this translation program passes all test cases, it is considered **program-level passed**. If all candidates of one of the translation lines are exhausted, and no translation program is compilable, the translation fails.

6 Results

Line-level and program-level compilable rate and passed rate of all tuned model mentioned in the Training section is summarized in Table 5. The baseline model with 10 epochs performs the best with 51.0% line-level passed

rate and 1.10% program-level passed rate. The performance is not ideal, and we will discuss the reason in the Limitation and Future Work section.

	line-level (out of 14765)		program-level (out of 1007)	
	compilable rate	passed rate	compilable rate	passed rate
ep5	9662 (65.4%)	7362 (49.9%)	79 (7.85%)	3 (0.30%)
ep10	9635 (65.3%)	7527 (51.0%)	75 (7.45%)	11 (1.10%)
h_ep5	9295 (63.0%)	6867 (46.5%)	44 (4.37%)	1 (0.1%)
h_ep10	9378 (63.5%)	7094 (48.0%)	61 (6.06%)	4 (0.40%)
s_ep5	9159 (62.0%)	7023 (47.6%)	53 (5.26%)	3 (0.30%)
s_ep10	9502 (64.4%)	7278 (49.3%)	64 (6.36%)	4 (0.40%)

Table 5: Line-level and program-level compilable rate and passed rate of tuned models

We also experimented with how beam size affects translation performance. Table 6 listed the performance of the baseline model with 5 and 10 epochs and beam sizes 5 and 10. Even though the compilable rates at the line and program levels increased by 10%, the passed rates at both levels remain roughly similar.

	line-level (out of 14765)		program-level (out of 1007)	
beam size	compaliable rate	passed rate	compilable rate	passed rate
ep5	9662 (65.4%)	7362 (49.9%)	79 (7.85%)	3 (0.30%)
ep10	9635 (65.3%)	7527 (51.0%)	75 (7.45%)	11 (1.10%)
beam10_ep5	10678 (72.4%)	7477 (50.6%)	94 (9.33%)	3 (0.30%)
beam10_ep5	10765 (73.0%)	7622 (51.6%)	95 (9.43%)	11 (1.10%)

Table 6: Line-level and program-level compilable rate and passed rate of baseline model with 10 epochs and beam size 1, 5, 10

7 Limitation and Future Work

We conjecture that the under-performance of the transformer model is mainly caused by the small dataset (about 90MB). Chen et al. [6] demonstrated that the transformer model actually performs worse than the LSTM model on small datasets, especially the decoder portion. Thus, expending the dataset

is a must for increasing the passed rate. Also, we can try to apply some tricks, such as GOLD [14] and tempering [8], to the model to improve performance.

Furthermore, as mentioned in the Dataset section, some source code programs in the dataset don't compile or pass all the test cases. Some programs are not compilable because they require a different version of C++ (the default version of our evaluation is C++14). Some are not compilable because we did not include some required libraries. We only included some commonly-used libraries, such as iostream, sstream, vector, map, set, queue, stack, math, numeric, etc. To improve the compilable rate of the original programs, we can accommodate other versions of C++ and include all used libraries.

7.1 Taking Advantages of Pre-trained Models

The transformer model led to the development of pre-trained models, such as BERT (Bidirectional Encoder Representations from Transformers) and GPT (Generative Pre-trained Transformer).

Codex [5] used GPT-3 model to synthesize Python docstrings to programs. They fine-tuned the model with 159 GB of Python files from GitHub and HumanEval, a dataset containing 10,000 competitive programming problems from codeforces.com with Python solutions. With 100 samples, it can solve 70.2% of the problem from the test set of HumanEval. DeepMind reported AlphaCode [13], a GPT model that is pre-trained by 715.1 GB of GitHub code from several popular languages (C++, C#, Go, Java, JavaScript, Lua, PHP, Python, Ruby, Rust, Scala, and TypeScript) and fine-tuned with the CodeContests, a 3GB dataset of competitive programming problems codeforces.com. AlphaCode achieved a ranking of the top 54.3% on average in codeforces competitions with more than 5,000 participants.

Given the scope of the project and limited computing resources, we are unable to use such large pre-trained models. It's definitely worth trying to use large pre-trained models if we can obtain a larger dataset.

8 Conclusion

In this project, we implemented and trained a transformer model and evaluated its performance on the SPoC dataset developed by Kulal et al. (2019). Among different parameters and beam sizes, the highest line-level compilable

rate is 73% and passed rate is 51.6% and the highest program-level compilable rate is 9.43% and passed rate is 1.1%. We conjecture the under-performance might caused by the small dataset, as studies have shown that transformer models are data hungry. For future work, we would like to expand the dataset and use pre-trained models.

References

[1] Uzzal Kumar Acharjee, Minhazul Arefin, Kazi Mojammel Hossen, Mohammed Nasir Uddin, Md Ashraf Uddin, and Linta Islam. Sequence-to-sequence learning-based conversion of pseudo-code to source code using neural translation approach. *IEEE Access*, 10:26730–26742, 2022.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014.

[4] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. Massive exploration of neural machine translation architectures, 2017.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[6] Mia Xu Chen, Orhan Firat, Ankur Bapna, Melvin Johnson, Wolfgang Macherey, George Foster, Llion Jones, Niki Parmar, Mike Schuster, Zhifeng Chen, et al. The best of both worlds: Combining recent advances in neural machine translation. *arXiv preprint arXiv:1804.09849*, 2018.

[7] 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, 2014.

[8] Raj Dabre and Atsushi Fujita. Softmax tempering for training neural machine translation models. *arXiv preprint arXiv:2009.09372*, 2020.

[9] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning, 2017.

[10] Alex Graves. Generating sequences with recurrent neural networks, 2013.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[12] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32, 2019.

[13] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[14] Richard Yuanzhe Pang and He He. Text generation by learning from demonstrations. *arXiv preprint arXiv:2009.07839*, 2020.

[15] Ofir Press and Lior Wolf. Using the output embedding to improve language models, 2016.

[16] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[18] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine

translation system: Bridging the gap between human and machine translation, 2016.

[19] Ruiqi Zhong, Mitchell Stern, and Dan Klein. Semantic scaffolds for pseudocode-to-code generation. *arXiv preprint arXiv:2005.05927*, 2020.