# Documentation Is All You Need

**Felipe Meneses**
Department of Computer Science
Stanford University
`fbomfim@stanford.edu`

## Abstract

Documentation, which provides a high-level description of the task performed by the source code, is a must-have for team-based software development groups. Conversely, even with various tools that have been developed to aid the programmer during its creation, crafting correct and consistent documentation for source codes remains a labor-intensive task. Thus, significant gaps in documentation are ubiquitous and considerably contribute to the inefficiency of program comprehension by software engineers. In this paper, I try to generate function descriptions from Python source code using a 178M parameter Transformer with pre-trained embeddings. Nonetheless, our final model was not able to reach state of the art performance.

## 1 Introduction

In Xia el al.'s paper *Measuring Program Comprehension: A Large-Scale Field Study with Professionals*[11] published on IEEE, the authors highlight the precarious conditions that program comprehension tools for software engineers find themselves. An empirical estimate shows that they spend, on average, approximately 58% of their time on trying to understand codes. This number gets even worse for junior developers.

Conversely, crafting correct and consistent documentation, a critical tool for easing code-comprehension tasks, persists as a burdensome effort. In low-resource software development environments, building documentation is not an option, and that can accrue lots of damage to a team's performance through time.

Hence, there is a need to make the process of building documentation leaner and more efficient. This paper thus aims to train an algorithm to perform code summarization tasks, which entails extracting high-level natural language descriptions of the codebase from raw low-level code. More precisely, the model takes as input a function's source code, and then use a Transformer with pre-trained embeddings to predict its description.

## 2 Related work

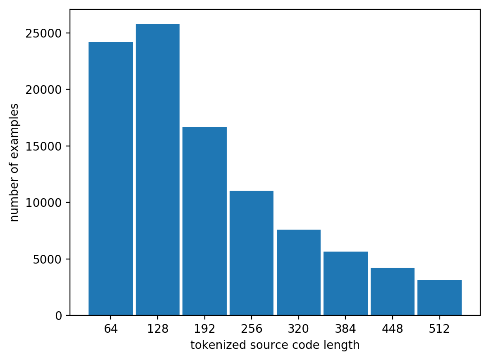
Wan et al.'s *Improving Automatic Source Code Summarization via Deep Reinforcement Learning*[9] uses an actor-critic network following a Hybrid2Seq architecture (primarily based on LSTMs) for their actor-network to construct function descriptions for Python code. Their deep reinforcement learning approach is ingenious because it restricts their model's image to a better description space. Their critic's reward system tries to minimize the distance between the output and the ground truth with respect to the BLEU metric.

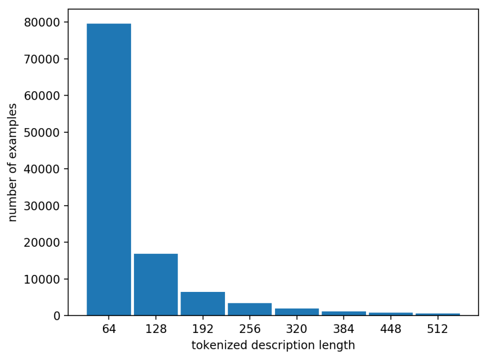
Alon et al.'s *code2seq: Generating Sequences from Structured Representations of Code* [1] uses better code embeddings for Java to create a state-of-the-art model using LSTMs. Those embeddings are based on previous work done in code2vec[2], which combines the source code's Abstract Synthax Trees (ASTs) and sequence to achieve great results.

Both models achieve outstanding performance in their tasks. Conversely, they are all LSTM based. Recently, we have seen that Generative Pre-trained Transformers [6] perform much better in NMT tasks. Even though code2seq shows that it can outperform the pure Tranformer [8] in those tasks, I believe that the improvement was achieved through its embeddings, but not its architecture. This, I believe that a Transformer model with large pre-trained embeddings should be able to outperform those models.

## 3 Dataset

The model is trained and validated on a real-world dataset of 108,726 Python code snippets and its corresponding descriptions taken from Wan et al.'s paper *Improving Automatic Source Code Summarization via Deep Reinforcement Learning*[9]. After tokenizing the examples, we find that the mean description length is 75.4 tokens with a standard deviation of 152.3, and the mean source code length is 288.5 tokens with a standard deviation of 889.8. However, if we restrict the tokenized length of the descriptions to 128, and the source code length to 512 we only reduce the dataset to 89,882 function-description pairs but will make our training much more efficient in terms of memory utilization. Furthermore, for functions under 512 tokens, the distribution is much smoother. Here is a plot of the bucketized data distribution:





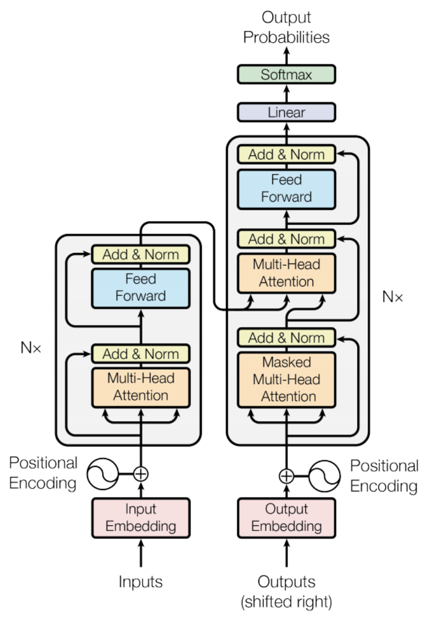
I decided to split our dataset into 80:10:10 for training, dev, and test sets, respectively, and also to do further preprocessing by assigning the examples into buckets based on the source code length. Next, I add start of sentence and end of sentence tokens to the descriptions. Finally, I split the data points into batches, tokenize them, and apply padding based on which bucket they reside. The result left us with a matrix of dictionary indices of shape $(batch\_size, bucket)$ for the inputs and $(batch\_size, 128)$ for the outputs. From now on, we will use source code and inputs interchangeably, likewise with descriptions and outputs.

## 4 Methods

To accomplish the proposed task I use a Transformer [8] with pre-trained GloVe[5] and GPT-2 [6] embeddings for the inputs and outputs respectively.

### 4.1 Transformer

The Transformer is an architecture that follows an encoder-decoder structure. The encoder maps an input vector $x \in \mathbb{R}^n$ to a sequence of continuous representations $z \in \mathbb{R}^n$. Given $z$, the decoder then generates an output sequence $y \in \mathbb{R}^m$ of symbols one element at a time. At each step, the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next. Hence, to train it, we need shift all the inputs to the right so the model learns to predict the next most probable word. The architecture looks like this:



Note that the first step of the decoder is Masked Multi-Head attention. Masking plays a crucial role in the model. Combined with the output embeddings offset by one position, it ensures that the predictions will depend only on the known outputs, i.e., the previous output tokens.

## 4.2 Attention

An attention function is mapping a query $Q$ and a set of key-value pairs $(K, V)$ to an output. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key. The attention function in the transformer is given by:

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

where $d_k = dim(K)$.

## 4.3 Embeddings

Ramachandran et al. [7] demonstrated that seq2seq models benefit from being initialized with pre-trained language models as encoders and decoders. Hence, both our encoder and decoder embeddings are initialized with pre-trained embeddings.

For our encoder, we pre-trained a GloVe language model on Python code for both functions and lines, and for 100 epochs with $d_{model} = 768$. The embeddings seemed to be more reasonable for the function trained code. So we used those for our encoder embeddings. For our decoder, we used pre-trained embeddings from HuggingFace's implementation of GPT-2 small. [10]

Choosing $N$, the number of encoder-decoder layers stacked to be 6 (same as in Vaswani et al.[8]), the number of heads in the Multi-Head Attention Layer to be 8, and $d_{model} = 768$, yielded a model with 178,249,298 parameters. We chose to use Adam[3] as our optimizer.

## 5 Experiments/Results/Discussion

Across the span of the quarter, I have trained 5 different versions of this Transformer. What was described above was the final model we used. For the first 2, I did not use triangle masking[1], and, looking at the results, I believe that because the model was able to peek ahead, it just learned how to copy the inputs. Since in the prediction step, we start with an empty string, the model would not predict anything. To prevent that from happening, I added masking. While looking at the data, I noticed that all the data points started with a quotation mark, and hence I used that as our beginning of sentence ($\langle bos \rangle$) token. As one may note, this was a poor choice because the quotation marks would also come at the end of the sentence. Hence, the probabilities we found were significantly skewed towards finishing the sentence immediately. For the fourth model, we added a special $\langle bos \rangle$ token. However, the issue faced was that we were not right-shifting the outputs in the decoder. Again, that caused the model to learn how to copy the output. After fixing that, we obtained our final model. Next, we will take a look at the training process and results obtained.

### 5.1 Training

I built the model[2] using PyTorch[4], and trained it on a computer equipped with 4vCPUs with 25GB of RAM and an NVIDIA Tesla K80 GPU with 12 GB of memory. To minimize the strain on the machine, I bucketize the inputs based on tokenized input length, and diminish the batch size accordingly. More precisely, we divided them into buckets of [64, 128, 192, 256, 320, 384, 448, 512] – the batch sizes are respectively [32, 16, 16, 16, 16, 16, 16, 8].

For our loss function, I used negative log likelihood (NLL) with a log-softmax activation applied on our linear output matrix for each column $x$ and target class $t$. This choice is a standard practice in NMT tasks.

$$loss(x, t) = -\log\left(\frac{\exp(x_t)}{\sum_j \exp(x_j)}\right)$$

The model is optimized using Adam, with learning rate $\alpha = 0.01$, betas $\beta_1 = 0.9$, and $\beta_2 = 0.999$. After looking at other papers, and the loss plot below, I believe the learning rate chosen was a bit too low. The Adam parameters were the standard ones learned in class. After training the model for 29 epochs, this is the loss plot we obtained for epochs 4 - 29:

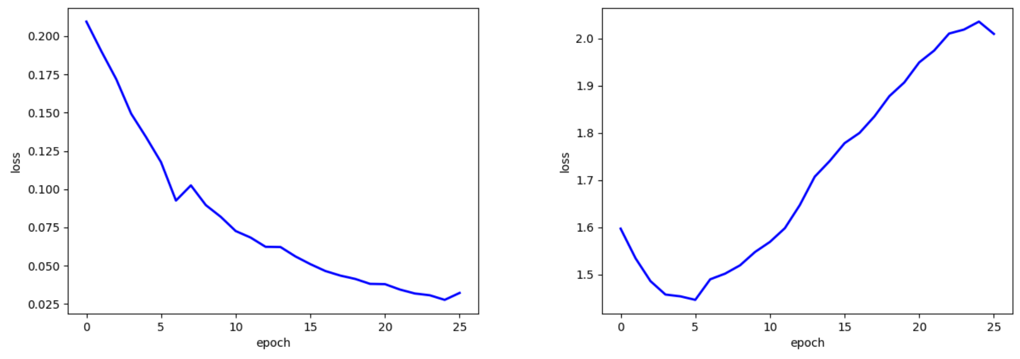
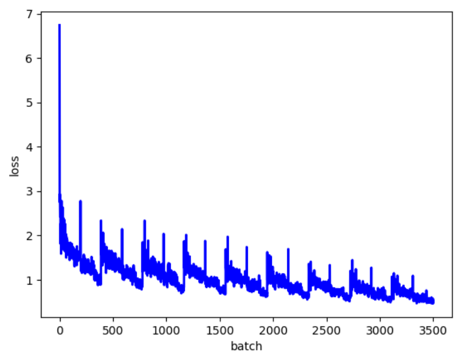


Figure 1: Left: training set loss; Right: dev set loss

and this is a more granular loss for the first 10 epochs:

[1]Triangle masking is used in the decoder so the model does not peek at the subsequent outputs. This is used during the attention step.

[2]Github link: `https://github.com/Felipebpm/init-ai`



The saw-shaped loss is explained by the different sequence lengths for each bucket, which slightly disturb the distribution of the data. We conjecture that this is part of the explanation for why the model fit the dev set so poorly.

I have also experimented with $\alpha' = 0.001$ and $\alpha'' = 0.0001$, but the model seemed to be learning too slowly[3].

### 5.2 Results

When analyzing the loss for both train and dev sets, we can see there is a huge variance problem in the model. The fact that the dev set loss distances itself so early from the training set loss indicates that the model might be overtly big for our dataset.

Another issue is that the striking disconnect between the sets seems to indicate that there was a representation issue in our algorithm. I assume the GloVe embeddings were insufficient in capturing the essence of the code and hence some of the meaning was dissipated through that layer, significanly hurting the generalizability of the model. A more interesting approach would be to use abstract synthax trees, such as in code2vec[2], with some form of tree encoder to capture more of the structural nature of source code.

Given that qualitatively the model was not working as expected, I invested most of the time in finding the issue and correcting it as fast as I could. Unfortunately, given the size and nature of the model I was not able to do so. Conversely, I am certain that the model performs worse than the baseline, an implementation of Wan et al.'s paper[9] Hybrid2Seq+Attn model, which scores 0.19 in the BLEU-1 score.

## 6 Future Work

Given more time and computational resources, I would have spent more time gathering data to solve our variance issue. Furthermore, I firmly believe this model could be vastly improved by using better code embeddings such as the one described in code2vec[2] and larger embeddings for the decoder, such as GPT-2 Large. I would also try to come up with an utility metric for a given description and try to combine Wan et al.'s deep reinforcement learning approach with our model.

## 7 Conclusion

There is a clear need to make the process of documenting codebases leaner and more efficient. In this paper we designed a Transformer to extract high-level descriptions from low-level source code. In the end I was able to train a 178M parameter Transformer over 30 epochs where the model ends up overfitting the training set. Finally, the biggest lesson from this paper is how important code embeddings are in code-summarization tasks. The best papers on the subject seek better code representation, and that's where our future effort should be put in.

[3]We have changed the learning rate in between epochs, from smallest to biggest.

## References

[1] Uri Alon, Omer Levy, and Eran Yahav. "code2seq: Generating Sequences from Structured Representations of Code". In: *CoRR* abs/1808.01400 (2018). arXiv: `1808.01400`. URL: `http://arxiv.org/abs/1808.01400`.

[2] Uri Alon et al. "Code2Vec: Learning Distributed Representations of Code". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 40:1–40:29. ISSN: 2475-1421. DOI: `10.1145/3290353`. URL: `http://doi.acm.org/10.1145/3290353`.

[3] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (2014). arXiv: `1412.6980 [cs.LG]`.

[4] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[5] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global Vectors for Word Representation". In: (2014), pp. 1532–1543. URL: `http://www.aclweb.org/anthology/D14-1162`.

[6] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: (2019).

[7] Prajit Ramachandran, Peter J. Liu, and Quoc V. Le. "Unsupervised Pretraining for Sequence to Sequence Learning". In: (2016). arXiv: `1611.02683 [cs.CL]`.

[8] Ashish Vaswani et al. "Attention is All you Need". In: (2017). Ed. by I. Guyon et al., pp. 5998–6008. URL: `http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf`.

[9] Yao Wan et al. "Improving Automatic Source Code Summarization via Deep Reinforcement Learning". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. Montpellier, France: ACM, 2018, pp. 397–407. ISBN: 978-1-4503-5937-5. DOI: `10.1145/3238147.3238206`. URL: `http://doi.acm.org/10.1145/3238147.3238206`.

[10] Thomas Wolf et al. "HuggingFace's Transformers: State-of-the-art Natural Language Processing". In: *ArXiv* abs/1910.03771 (2019).

[11] X. Xia et al. "Measuring Program Comprehension: A Large-Scale Field Study with Professionals". In: *IEEE Transactions on Software Engineering* 44.10 (Oct. 2018), pp. 951–976. DOI: `10.1109/TSE.2017.2734091`.