Long-Range Modeling of Source Code Files with eWASH: Extended Window Access by Syntax Hierarchy

Colin B. Clement
Microsoft Cloud and AI
colin.clement@microsoft.com

Shuai Lu
Microsoft Research
shuailu@microsoft.com

Xiaoyu Liu
Microsoft Cloud and AI
lixiaoyu@microsoft.com

Michele Tufano
Microsoft Cloud and AI
mitufano@microsoft.com

Dawn Drain
Microsoft Cloud and AI
dawndrain95@gmail.com

Nan Duan
Microsoft Research
nanduan@microsoft.com

Neel Sundaresan
Microsoft Cloud and AI
neels@microsoft.com

Alexey Svyatkovskiy
Microsoft Cloud and AI
alsvyatk@microsoft.com

Abstract

Statistical language modeling and translation with transformers have found many successful applications in program understanding and generation tasks, setting high benchmarks for tools in modern software development environments. The finite context window of these neural models means, however, that they will be unable to leverage the entire relevant context of large files and packages for any given task. While there are many efforts to extend the context window, we introduce an architecture-independent approach for leveraging the syntactic hierarchies of source code for incorporating entire file-level context into a fixed-length window. Using concrete syntax trees of each source file we extract syntactic hierarchies and integrate them into context window by selectively removing from view more specific, less relevant scopes for a given task. We evaluate this approach on code generation tasks and joint translation of natural language and source code in Python programming language, achieving a new state-of-the-art in code completion and summarization for Python in the CodeXGLUE benchmark. We also introduce new CodeXGLUE benchmarks for user-experience-motivated tasks: code completion with normalized literals, method body completion/code summarization conditioned on file-level context.

1 Introduction

Large transformer models (Vaswani et al., 2017) and the pre-training/fine-tuning paradigm (Devlin et al., 2018; Lewis et al., 2019; Radford et al., 2018) have become an essential part of state of the art natural language processing. Beyond the domain of natural language, these models and procedures have enabled rapid progress in the software engineering space, including applications in code completion (Svyatkovskiy et al., 2020, 2019; Clement et al., 2020; Raychev et al., 2014; Bruch et al., 2009), natural language to code (NL2Code), code feature summarization (Clement et al., 2020; Moreno et al., 2013; Scalabrino et al., 2017; Wan et al., 2018; Alon et al., 2018; Moreno et al., 2014), code search (Husain et al., 2019; Feng et al., 2020), unit test generation (Tufano et al., 2020) and even bug fixing (Drain et al., 2021) and detection (Zhai et al., 2020).

A major difference between transformers and their antecedents like recurrent neural networks (RNN) is their strictly enforced finite context window. Whereas an RNN can iteratively consume as many tokens as is required, transformers can only consume up to a finite amount decided at training time. Further, it is impractical to simply expand the window as the memory and compute requirements of the attention mechanism scale quadratically with context length. There have been efforts to economically expand the window by modifying the attention mechanism with low-rank queries and keys (Beltagy et al., 2020), sparse connections (Parmar et al., 2018; Zaheer et al., 2020), and more recently approximation by kernel methods (Choromanski et al., 2021). There have also been meth-

ods developed to condition generation on retrieved documents (Lewis et al., 2020b,a) for knowledge-intensive applications. Complimentary to these approaches, and consistent with any sequence model or architecture, we propose a method for extracting the most important features distant from the task at hand, implemented in this case using the syntax tree of source code.

A source code document has nested scopes and references to other documents/libraries, and software engineering tasks must leverage knowledge at all of these scales. Even single source code files often exceed the length of the context window, and so it is clear that progress in modeling source code requires overcoming this limitation. Instead of proposing a novel transformer architecture capable of ingesting more context, we propose a method of compressing the context of source code files using the syntax of the programming language itself. In this paper we introduce eWASH: Extended Window Access by Syntax Hierarchy, which leverages the syntax hierarchy of source code to give our models longer ranged vision by prioritizing higher level scopes and names for source code elements in a file which are not immediately in focus. Using eWASH assumes that higher level scopes like function signatures summarize the whole method, and viewed in this way eWASH could be applied to natural language by extracting key terms in a long document or summarizing features distant from the task at hand.

We start by explaining eWASH with a motivating example and define its features for three important software engineering tasks. The first is code completion, in which some number of tokens are predicted to extend a partially complete source code file. The second is method completion, wherein a whole method body is predicted from a method signature and docstring. The third is code summarization or docstring generation, wherein a method is mapped to a natural language docstring. We then discuss the Python training data and the models employed in this study, which include the auto-regressive GPT-C (Svyatkovskiy et al., 2020), an eWASH extended version called XGPT-C, Python method/docstring prediction model PyMT5 (Clement et al., 2020), similarly named XPyMT5, and Performer (Choromanski et al., 2021) and Reformer (Kitaev et al., 2020) baselines. We demonstrate through experiments the value of eWASH for the source code domain

with state of the art performance for code completion and method completion and code summarization or docstring generation. Finally, we study user-experience motivated properties of XGPT-C, and extend the source code benchmark set CodeXGLUE (Lu et al., 2021) with two new tasks: literal-normalized code completion and method completion. Surprisingly, eWASH excels at code completion even when the context window is not a limiting factor.

2 Motivating example

In this paper we consider three software engineering tasks: code completion, method completion, and docstring completion or code summarization. Code completion is the auto-regressive prediction of one or more tokens conditioned on a provided context. Figure 1 shows an incomplete Python program implementing a neural network. A developer would like a model which can predict the body of `ConvNet.forward` (method completion) by composing the layers defined in `ConvNet.__init__` and globally imported operations. There is a limited context window, illustrated at the left of the figure, and so while the model can (in this fortunate case) see the layer definitions, it is ignorant of the imports and the global `LOGGER` object.

In many cases predicting a whole method is not easy, but asking for a few members or a completed line (Svyatkovskiy et al., 2020) is still desirable. The bottom of fig. 1 shows the code completion task finishing an assignment and completing a line in a partial implementation of `ConvNet.forward`. Here, again, crucial information is missing from the input of the model, which will prevent it from being able to reference the imports. Further, one can easily imagine a scenario in which more spurious information is fed into the model instead of, for example, the definitions of the neural network layers in the `__init__`. How can we ensure the model is shown important information for predictions?

2.1 Extended Window Access by Syntax Hierarchy

Software developers carefully organize their code into scopes and portable elements using hierarchies like methods and classes, so we hypothesize that labels of these scopes are more important for long-range modeling than their contents. We propose

```python
import logging
import torch
import torch.nn as nn
import torch.nn.functional as F

LOGGER = logging.getLogger()


class ConvNet(nn.Module):
    """Basic few layer ConvNet"""
    num_class = 10

    def __init__(self):
        """Define network layers"""
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout2d(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, self.num_class)

    # target body
    def forward(self, x):
        """Evaluate Net on input x"""
```

```
x = self.conv1(x)
x = F.relu(x)
x = _____
```

XPyMT5 — Method completion

XGPT-C — Code completion

```python
x = self.conv1(x)
x = F.relu(x)
x = self.conv2(x)
x = F.relu(x)
x = F.max_pool2d(x, 2)
x = self.dropout1(x)
x = torch.flatten(x, 1)
x = self.fc1(x)
x = F.relu(x)
x = self.dropout2(x)
x = self.fc2(x)

output = F.log_softmax(x, dim=1)
return output
```

```python
x = self.conv1(x)
x = F.relu(x)
x = self.conv2(x)
```
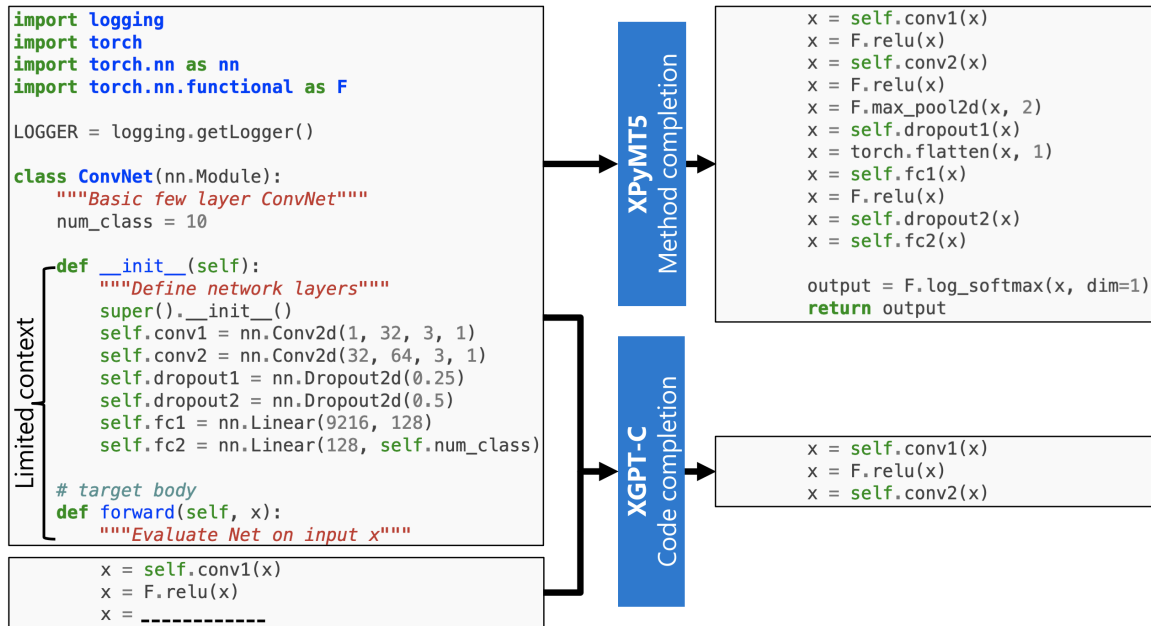
Figure 1: An example scenario where both method completion (top right) and code completion (bottom right) are performed by our eWASH models XPyMT5 and XGPT-C, respectively. Method completion aims to predict a whole method body conditioned on a signature and docstring and other context, and code completion aims to predict any number of tokens to complete a member, line, or even a scope context conditioned on any incomplete code string. In this case XPyMT5 is tasked with predicting the whole body of `ConvNet.forward`. In this case it's clear that both the layers assigned in `ConvNet.__init__` and the import statements above are important information. In another case XGPT-C aims to complete the assignment, and again the class attributes and import statements are important. Both models have a limit context window illustrated at left, and exclude important information.

eWASH, Extended Window Access by Syntax Hierarchy, in which we compress the context provided to our model by prioritizing, for example, function signatures over function bodies. Since most code is written inside methods, we center method bodies as the focus of the modeling task for eWASH, calling each method being modeled the 'focal method.'

Figure 2 shows how eWASH uses syntax hierarchies to prioritize elements of the context of the method and code completion example of Fig. 1. The focal method in this case is `ConvNet.forward`, but could be any other method in the module. The most important part for modeling the body of this focal method is its signature and docstring (if present) and containing class definition (if the focal method is a class method). After this we prioritize global import statements and assigned values (but not yet the assigned expressions), followed by class attributes, peer class method signatures, class docstring, peer class method docstrings, and finally global expressions and the code bodies of peer class methods.

In practice, eWASH is implemented by taking the concrete syntax tree of the source file and orga-

nizing the syntactic elements in our priority list, tokenizing each element, and then descending the priority list, taking elements until the context window has been filled. For training the method completion of XPyMT5, we arrange the eWASH context in the input with a control code to indicate which method is to be completed (`# target body` in Fig. 1), and we arrange the target to be the method body. eWASH yields N total training samples from a file with N total methods and class methods. For docstring completion or code summarization, the source contains the method signature and body, and the target contains the desired docstring, and a control code is used to instruct the model which task it is to perform, just like PyMT5 (Clement et al., 2020).

For code completion, as we use an autoregressive decoder in the form of XGPT-C there is no special 'position,' and so we create a rolling window across the focal method body. We reserve 3/4 (768/1024 tokens) of the tokens for the context, and 1/4 (256/1024 tokens) for the rolling window of the body. In the case of a method which exceeds 256 tokens, the training sample for that method is

decomposed into multiple 'windows,' and one file yields at least N training samples for a file with N method and class method definitions.
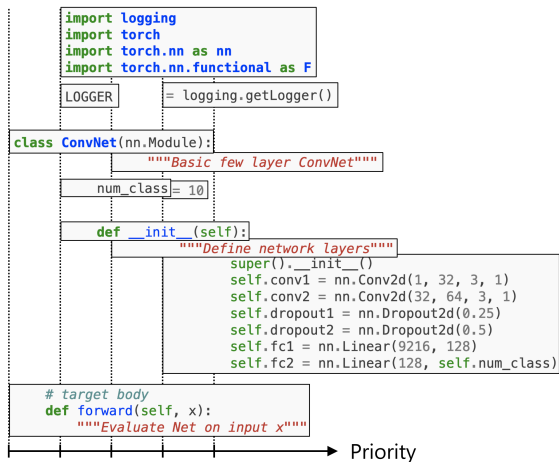
```python
import logging
import torch
import torch.nn as nn
import torch.nn.functional as F

LOGGER = logging.getLogger()

class ConvNet(nn.Module):
    """Basic few layer ConvNet"""
    num_class = 10

    def __init__(self):
        """Define network layers"""
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout2d(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, self.num_class)

    # target body
    def forward(self, x):
        """Evaluate Net on input x"""
```

Priority

Figure 2: An illustration of the syntax hierarchy of the input context in the method completion example from Fig. 1. The eWASH (Extended Window Access by Syntax Hierarchy) method selectively fills the model context going down the file, in the order of priority level indicated, and stops when the token budget of the model context is filled. eWASH presupposes that names of entities at higher scopes are more relevant to the task at hand than entities at lower scopes.

3 Dataset

3.1 Pre-training

The data for training is the same for both XGPT-C and XPyMT5, and consists of all 5+ star GitHub repositories which are primarily Python, filtered by files which were either Python 3 compliant or were successfully fixed by lib2to3. Further, there was a time limit of 10 seconds placed on the parsing process to eliminate files which are essentially data files as they tend to contain very large lists for example. Table 2 shows summary statistics of this dataset for a sense of scale.

3.2 Fine-Tuning and Evaluation

For evaluation and fine-tuning of code completion we used the Py150 (Raychev et al., 2016) from CodeXGLUE (Lu et al., 2021), and for method and docstring completion we used the CodeSearch-Net (Husain et al., 2019) dataset. Py150 is larger than CodeSearchNet for Python, but has selected repositories with good docstring coverage, allowing better evaluation of the method/docstring completion task.

4 Baseline Models

We consider state-of-the-art transformer models for code completion and code summarization tasks in the CodeXGLUE benchmark as our baselines. Namely, the generative pre-trained transformer model for code (Svyatkovskiy et al., 2020) (GPT-C) and the Python method text-to-text transfer transformer model (Clement et al., 2020) (PyMT5). We also experiment with two memory-efficient transformers—Reformer (Kitaev et al., 2020) and Performer (Choromanski et al., 2021)—which enables modeling of context lengths in excess of 1024 tokens.

4.1 GPT-C

GPT-C is an auto-regressive language model pre-trained on a large unsupervised source code corpora. Treating the source code data as a sequence of lexically-defined tokens, GPT-C extracts training samples as a sliding window from each source code file. This baseline uses an approach based on statistical language modeling of source code, with several normalization rules extracted from concrete syntax tree of a program. To overcome the issue of different styles and white space or tab conventions, it transforms the code into symbolic program tokens using custom tokenizer and regenerates the code with a common style. During pre-processing, GPT-C parses program code in each file, extracts information about token types, normalizes uncommon literals, trains a sub-token vocabulary, and encodes the files into sub-token sequences. This is done both for training and inference. GPT-C decoder-only model has about 125M parameters and a context length of 1024 tokens.

4.2 PyMT5

PyMT5 is a transformer encoder-decoder model jointly pre-trained on a large-scale corpus of Python source code and natural language contained in the docstring summaries. PyMT5 training samples are supervised pairs of function code features—function signatures, docstrings and bodies—extracted by means of a parser. PyMT5 is finetuned to translate between all non-degenerate combinations of code features in a multi-modal setting, e.g. simultaneously signature and docstring to body, signature to docstring and body, signature to body, etc. PyMT5 only uses information from a single method and naturally is missing imports, peer class and method definitions, and global as-

Model	PPL	ROUGE-L Prec.	Recall	Edit dist.	EM@5 (%)	Size
Performer	2.06	0.69	0.80	85.1	41.2	125M
Reformer	2.02	0.70	0.81	86.3	46.1	116M
XGPT-C	**1.35**	**0.85**	**0.93**	**90.8**	**49.4**	125M
GPT-C, Norm Literals	1.83	0.81	0.94	89.0	46.3	125M
XGPT-C, Norm Literals	**1.25**	**0.90**	**0.96**	**93.7**	**62.4**	125M

Table 1: Evaluation results comparing XGPT-C decoder-only model trained with extended hierarchical context with baselines on code completion from sampled methods. Model performance metrics are reported on test samples from the CodeXGLUE code completion task as described in Sec. 7.2.

Lang	Repos	Files	Methods	Classes	Lines
Python	238k	4.3M	15M	10.6M	1.4B

Table 2: Summary statistics of our Python parallel corpus compared to others presented in the literature. CSN contains 500k Python methods with docstrings, among 6 other languages. Our parallel corpus is 3× as large as the next largest, and over 15× the size of the next largest Python parallel corpus.

signments. PyMT5 has 406M parameters and a context width of 1024 tokens for both the encoder and decoder.

4.3 Memory-Efficient Transformers

Reformer and Performer transformer models attempt to break the infamous quadratic attention bottleneck and allow for efficient modeling with much longer than the standard 1024 token context window. Reformer (Kitaev et al., 2020) includes three memory optimizations: reversible layers (to trade off memory with time), axial positional embeddings, and a bucketed attention. Performer (Choromanski et al., 2021) develops a linear approximation to the attention layer $AV \approx \sigma(Q^T K)V$, where K, Q, and V are the key, query and value matrices of the attention mechanism and A is the softmax kernel approximation, and exploits the linearity to improve computational efficiency.

5 Defining the Tasks at Hand

5.1 Code completion

Code completion is the auto-regressive completion of source code tokens, as illustrated in the bottom of Fig. 1. We perform code completion as defined in CodeXGLUE (Lu et al., 2021) as well as using normalized literals. The literal normalization improves user experience of the code completion tool (Svyatkovskiy et al., 2020) by abstracting per-

sonally identifiable information and encouraging the model to focus on code modeling over arbitrary strings. Names, the phone number, IP addresses, and more may be preserved in the string or numeric literals. We normalize the literals in source codes to some special tokens. Considering that frequently used literals may contain useful information, e.g. `"__main__"` or `"utf-8"`, we preserve the 200 most frequent string and 30 most frequent numeric literals.

5.2 Method completion

Method completion is the prediction of a method body implementation conditioned on a signature, optional docstring, and any more context. The authors of PyMT5 performed this task using no context beyond the focal method, and XPyMT5 uses eWASH compressed file-level context. We contribute method and docstring completion conditioned on file-level information as a task to CodeXGLUE, based on the CodeSearchNet dataset in order to bolster its user-experience motivated tasks.

5.3 Docstring Completion/Code Summarization

Docstring completion is the prediction of a docstring conditioned on a focal method and optional context, and was also performed by PyMT5 on focal-methods alone. Code summarization is closely related, as docstrings often express a summary of their method, but also include annotated arguments, return values, exceptions, and even test cases via `doctest`. We train on docstring completion but evaluate on the CodeSearchNet dataset which attempts to remove everything but the summary which is assumed to be in the first paragraph of the docstring.

6 Model Training

6.1 XGPT-C

We trained XGPT-C on the Python dataset described in 3. Each training sample is a method body along with its corresponding extended context. In XGPT-C, we follow GPT-C (Svyatkovskiy et al., 2020), using a multi-layer Transformer decoder as the model architecture and the causal language model as the training objective. We use 12 layers Transformer decoder, 12 attention heads with 768 hidden dimension in total and sentencepiece [1] BPE vocabulary of size 50,000. The total number of model parameters is 125M. The pre-training period takes 2 weeks on sixteen 32GB Tesla V100 GPUs, and all hyperparameters were left as Svyatkovskiy et al. (2020).

6.2 XPyMT5

We trained XPyMT5 on the same Python dataset as XGPT-C. Similar to PyMT5, Each Python file yielded between N and $3N$ training samples where N is the number of methods and class methods in the file. Each method teaches the model to complete the method body conditioned on its signature (and docstring if it exists), to predict the docstring (if it exists) from the method, and to predict the whole method from just the docstring (if it exists). In this way, XPyMT5 also can jointly predict code and natural language, but we did not include all degenerate combinations like PyMT5 as the training set was already much larger due to the extended context.

XPyMT5 uses the same whitespace-augmented GPT-2 (Radford et al., 2018) tokenizer as PyMT5, with about a vocabulary size of 50,000, and is the same architecture and hyperparameters as PyMT5 with 12 layers and 406M parameters. XPyMT5 was trained on 16 32GB Tesla V100 GPUs for 4 weeks, about 10 epochs total, using the same hyperparameters as reported by Clement et al. (2020). XPyMT5 was initialized with the English pre-trained BART (Lewis et al., 2019) weights (with whitespace embeddings) and pre-trained using the BART de-noising objective for 5 weeks on the same hardware as above.

6.3 Reformer/Performer

We trained both Performer and Reformer models on the Python dataset described in 3 but without

eWASH. Each training sample is a whole source code file literal normalization applied. We adapt the open-sourced model implementations, [23] setting the architecture parameters of each to be as close to the same parameter count as XGPT-C as possible. Both used 12 layers, a context length of 4096, and 768 hidden dimensions. All other hyperparameters were unchanged from their default.

7 Evaluation

7.1 Metrics

The metrics we used to evaluate eWASH and thus XGPT-C and XPyMT5 are consistent with GPT-C and PyMT5 and other works in the literature. We report the longest common subsequence ROUGE-L as we expect in a developer tool scenario that users will want predicted code with the fewest edits. To that end, we also report the edit distance between the truth and hypothesis. In order to compare to other code completion models we report Exact-Match@N (EM) metrics (Rajpurkar et al., 2016), which counts the fraction of exactly correct predictions of some length (@N). For method and docstring completion we report BLEU-4 and ROUGE-L metrics, but not exact matches as it is too strict to meaningfully interpret for longer source-target pairs. For method completion we also report the fraction of syntactically correct methods as judged by Python 3.8 syntax.

7.2 Experimental Conditions

We aim to evaluate how well XGPT-C model can infer developers' true intents. We randomly selected 833 unique Python functions from the code completion test benchmark in CodeXGLUE (Lu et al., 2021), and, except for the first two tokens in each line, prompted the model at all other points inside the methods. The predictions are compared to the true continuation of the code. For method and docstring completion, the CodeSearchNet repositories and specific commit hashes were re-downloaded in order to extract the eWASH features in addition to the individual methods released. We will release this expanded CSN dataset and task to CodeXGLUE to improve its user-experience motivated metrics. Inference in all cases was performed with beam search with a beam width of 5.

---

[1]https://github.com/google/sentencepiece

[2]https://github.com/lucidrains/reformer-pytorch
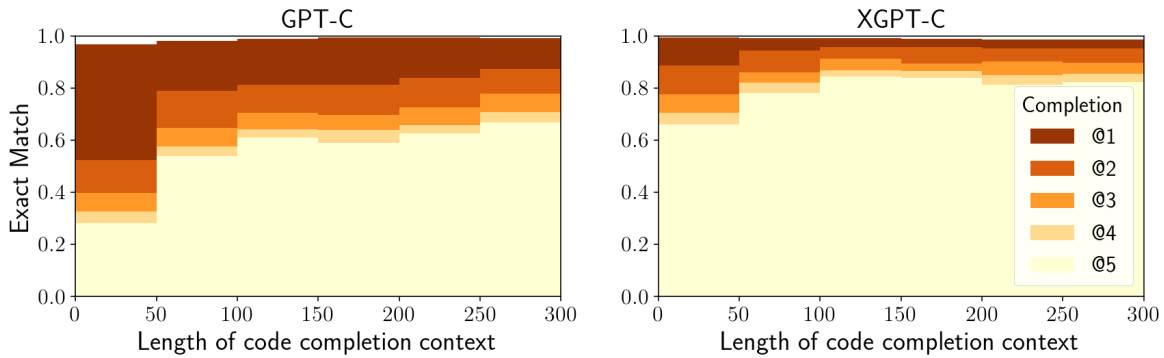[3]https://github.com/lucidrains/performer-pytorch

Figure 3: Comparing baseline GPT-C with XGPT-C in an offline evaluation of ExactMatch@1-5 code completion as a function of total token context length for the normalized literal scenario. Surprisingly, eWASH leads XGPT-C to benefit most over GPT-C at the shorter context lengths. XGPT-C also more exactly predicts tokens with longer context as well.
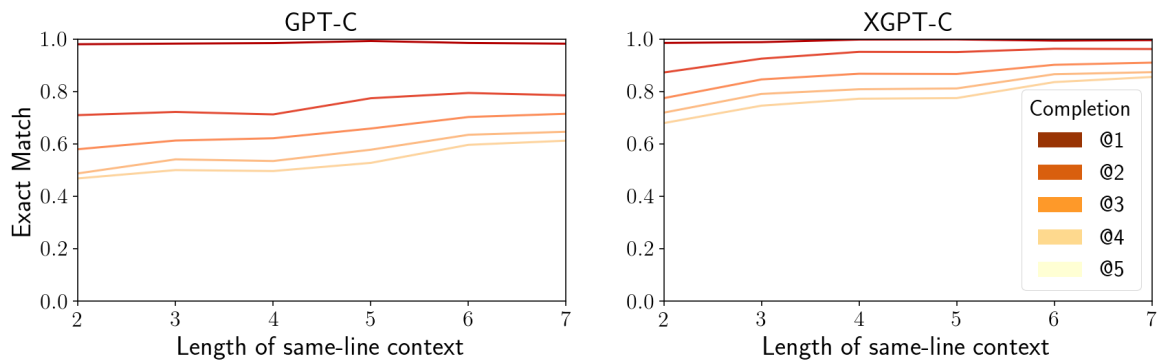

Figure 4: Comparing baseline GPT-C with XGPT-C in an offline evaluation of ExactMatch@1-5 code completion as a function of local, same-line context length for the normalized literal scenario. XGPT-C is better than GPT-C by all measures, showing how leveraging longer range context can most help developers even when a line being edited has a few tokens.

| | | **Exact Match** | | | | |
	@1	@2	@3	@4	@5	Total
GPT-C top-1	96.0	68.5	56.3	49.6	46.3	63.1
top-5	98.8	81.0	70.5	63.5	59.6	74.5
XGPT-C top-1	98.0	81.7	71.9	66.6	62.4	75.9
top-5	98.9	94.3	87.7	83.1	79.7	88.7

Table 3: Code completion evaluated on the CodeXGLUE test set by ExactMatch@1-5 and overall EM results for XGPT-C and GPT-C.

7.2.1 Code Completion Evaluation Results

As shown in Table 1, eWASH allows XGPT-C to beat both the GPT-C baseline and the memory efficient transformers on all the metrics computed. About 10% of our Python test files were greater than 1024 tokens in length, and evaluating separately on that subset yielded slight improvements of Performer/Reformer, but Performer only beat XGPT-C in terms of EM@5 at 55.9%. These evaluations were performed for source code inside methods, as the eWASH technique follows the syntactic

hierarchy used by developers. Note that the bottom lines are trained and evaluated on the normalized literal dataset. XGPT-C sees a large absolute increase in ExactMatch@5 of 13% with normalized literals showing that, in addition to protecting user data, normalizing literals is an important part of a good IDE programming assistant.

Hellendoorn et al. (2019) showed that artificial evaluation scenarios are often much more forgiving than real-world scenarios. To better evaluate whether these models can predict a developer's intent we compute the ExactMatch@1-5 benchmark, described in Sec. 7.1, broken down by total token context and length of same-line context. Figure 3 shows EM@1-5 metrics for the normalized literal scenario binned by the context length for the completion for GPT-C (left) and XGPT-C (right). It is clear that in all measured cases eWASH allows XGPT-C to better predict exact matches. Perhaps most strikingly, the largest relative increase in EM occurs for shorter context lengths, so that the

4719

syntactic hierachy hypothesis underlying eWASH appears most beneficial for context lengths well within the context window.

Figure 4 shows the same EM metrics broken down by same-line context length, to test how much the most proximal tokens matter for prediction. We see the same overall benefit of eWASH in XGPT-C, and only a slow increase as a function of same-line context. The average line length in our data is 18 tokens, so with 7 tokens of same-line context, XGPT-C can complete 5 tokens exactly more than 80% of the time while GPT-C can do so just shy of 60% of the time. Again, this is very interesting as eWASH confers great benefit even when context lengths do not exceed the context window, and supports our hypothesis that user-defined syntax hierarchies are very important signals for predicting method bodies.

Modern IDE environments like Visual Studio Intellicode (Visual Studio Intellicode) can present multiple predictions, which Hellendoorn et al. (2019) showed can improve real-world user acceptance. Table 3 shows the overall ExactMatch@1-5 metrics for code completion regardless of context length. XGPT-C is the clear winner again for all the EM metrics, boosting total exact matches by over 12% for top-1 predictions and reaching 88.7% overall for top-5 predictions. We interpret this to mean that eWASH will enable superior on-line user acceptance of code completions.

7.3 Method Completion Evaluation Results

We evaluate eWASH for method generation, illustrated in the top of Fig. 1. Table 4 shows the comparison between XPyMT5 and PyMT5 and, and PyMT5 is superior in all the source-target comparison metrics. Syntax correctness is slightly lower, but the difference is not necessarily meaningful. The ROUGE-L metrics are dramatically improved, and is not necessarily surprising as XPyMT5 is conditioned on much more information than PyMT5. The syntax correctness of our fine-tuned models is slightly lower than the 92.1% reported by Clement et al. (2020).

7.4 Docstring Completion Evaluation Results

Table 5 compares XPyMT5 to PyMT5 for docstring completion (or code summarization as CodeSearch-Net removes the variable annotations). Again there is a large improvement in performance across all metrics, with a striking doubling of the ROUGE-L F1 score with eWASH features.

8 Conclusions

Inspired by the performance of transformer models, their limited context window size, and the especially long-range nature of source code as documents, we developed Extended Window Access by Syntax Hierarchy. Our hypothesis was that the syntax hierarchy imposed by developers is a real signal of importance in a task context, and that methods, containing most lines of code, are most dependent on the higher-level scopes of their file-level attributes. Our XGPT-C results for code completion supported this hypothesis, and, strikingly, offered most relative benefit for shorter context lengths. We showed with strict exact match metrics that eWASH allows a large relative improvement in code completion predictions. Finally we show dramatic improvement in method completion and code summarization with XPyMT5. eWASH can be applied to any programming language and in principal any language with hierarchical syntactic or stylistic structure. For this reason we believe eWASH to be a general purpose modeling approach for more optimally using finite context windows on structured documents, and could improve natural language understanding tasks as well. Further, any model, even the largest GPT-3 language model (Brown et al., 2020) can leverage the eWASH feature. Accompanying this manuscript we submit 3 new tasks to CodeXGLUE to bolster its user-experience motivated metrics: literal-normalized code completion, method-level code completion, and method/docstring completion conditioned on whole-file context.

References

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering*

Model	RL Prec.	Recall	F1	BLEU-4	Syntax (%)	Model size
PyMT5 Baseline	0.33	0.46	0.35	0.27	**89%**	406M
XPyMT5	**0.52**	**0.64**	**0.55**	**0.31**	88%	406M

Table 4: Evaluation results for XPyMT5 multi-mode encoder-decoder model trained with extended hierarchical context and various baselines on the task of method generation given a natural language description. Model performance metrics are reported on CodeSearchNet test sample in Python programming language.

Model	RL Prec.	Recall	F1	BLEU-4	Model size
CoTexT (CodeXGLUE leader)				0.197	
PyMTBaseline	0.32	0.37	0.32	0.28	406M
XPyMT5	**0.58**	**0.66**	**0.66**	**0.47**	406M

Table 5: Detailed evaluation results for XPyMT5 model trained with extended hierarchical context and various baseline on code summarization task. Model performance metrics are reported on CodeSearchNet test sample in Python programming language.

conference and the ACM SIGSOFT symposium on the foundations of software engineering, pages 213–222.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. 2021. Rethinking attention with performers. *ICLR 2021*.

Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. Pymt5: Multi-mode translation of natural language and python code with transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9052–9065.

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

Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. 2021. Generating bug-fixes using pretrained transformers. *arXiv preprint arXiv:2104.07896*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, pages 1536–1547.

Vincent J Hellendoorn, Sebastian Proksch, Harald C Gall, and Alberto Bacchelli. 2019. When code completion fails: A case study on real-world completions. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 960–970. IEEE.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer. *ICLR*.

Mike Lewis, Marjan Ghazvininejad, Gargi Ghosh, Armen Aghajanyan, Sida Wang, and Luke Zettlemoyer. 2020a. Pre-training via paraphrasing. *arXiv preprint arXiv:2006.15020*.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*.

Patrick Lewis, Ethan Perez, Aleksandara Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020b. Retrieval-augmented generation for knowledge-intensive nlp tasks. *arXiv preprint arXiv:2005.11401*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.

Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 23–32. IEEE.

Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. 2014. Automatic generation of release notes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 484–495.

Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. 2018. Image transformer. In *International Conference on Machine Learning*, pages 4055–4064. PMLR.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428.

Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2017. Automatically assessing code understandability: How far are we? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 417–427. IEEE.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: code generation using transformer. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1433–1443. ACM.

Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2727–2735.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. Generating accurate assert statements for unit test cases using pretrained transformers.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Visual Studio Intellicode. 2021. Visual studio intellicode. https://visualstudio.microsoft.com/services/intellicode/. Accessed: 2021-02-08.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407.

Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big bird: Transformers for longer sequences. *arXiv preprint arXiv:2007.14062*.

Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. Cpc: Automatically classifying and propagating natural language comments via program analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1359–1371.