LEARNING REPRESENTATIONS OF SOURCE CODE FROM STRUCTURE & CONTEXT

Masters Thesis

submitted on *March 15th 2019*
presented on *April 8th 2019*

at the École Polytechnique Fédérale de Lausanne,
in the faculty of Microengineering,

conferring the degree of *Master Sc. Eng.*
in *Robotics and Autonomous Systems* to

Dylan Teo Sommers Bourgeois

Supervised by:

Professor Jure Leskovec, *Stanford University*
Professor Pierre Vandergheynst, *EPFL*
Michaël Defferrard, *EPFL*
Dr. Michele Catasta, *Stanford University*

Declaration of authorship

I, Dylan Teo Sommers Bourgeois, declare that this Master Thesis and the work presented in it are my own.

In Palo Alto, March 15th 2019.

[illegible]

Official Project Description

Large codebases are routinely indexed by standard Information Retrieval systems, starting from the assumption that code written by humans shows similar statistical properties to written text [Hindle et al., 2012]. While those IR systems are still relatively successful inside companies to help developers search on their proprietary codebase, the same cannot be said about most of public platforms: throughout the years many notable names (Google Code Search, Koders, Ohloh, etc.) have been shut down. The limited functionalities offered, combined with the low quality of the results, did not attract a critical mass of users to justify running those services. To this date, even GitHub (arguably the largest code repository in the world) offers search functionalities that are not more innovative than those present in platforms from the past decade.

We argue that the reason why this happens has happened can be imputed to the fundamental limitation of mining information exclusively from the textual representation of the code. Developing a more powerful representation of code will not only enable a new generation of search systems, but will also allow us to explore code by functional similarity, *i.e.*, searching for blocks of code which accomplish similar (and not strictly equivalent) tasks.

In this thesis, we want to explore the opportunities provided by a multimodal representation of code: (1) hierarchical (both in terms of object and package hierarchy), (2) syntactical (leveraging the Abstract Syntax Tree representation of code), (3) distributional (embedding by means of co-occurrences), and (4) textual (mining the code documentation). Our goal is to distill as much information as possible from the complex nature of code.

Recent advances in deep learning are providing a new set of techniques that we plan to employ for the different modes, for instance Poincaré Embeddings [Nickel and Kiela, 2017] for (1) hierarchical, and Gated Graph NNs [Li et al., 2016] for (2) syntactical. Last but not the least, learning multimodal similarity [McFee and Lanckriet, 2011] is an ulterior research challenge, especially at the scale of large codebases – we will explore the opportunities offered by a framework like GraphSAGE [Hamilton et al., 2017] to harmonize a large graph with rich feature information.

Acknowledgements

First and foremost, I would like to thank my parents to whom I am eternally grateful for their unwavering love and support. I owe them, very literally, everything.

Next, I would like to thank those who made this project possible, in particular Michele “pirroh” Catasta, not only for providing me with this opportunity but also for being a mentor, a friend during these times and those to come. Thank you to Pr. Jure Leskovec for welcoming me into his lab, Pr. Pierre Vandergheynst and Michaël Defferrard for accepting to supervise me during this time.

Thank you to the entire SNAP laboratory for the time spent in the lab and on the slopes. In particular, thank you to the 452 family - Baharan “Amazing” Mirzasoleiman”, Robert “Robissimo” Palovics, and of course Michele. Many thanks to Adrijan Badaraschia his support, Rex Ying and Marinka Zitnik for the very interesting work we were able to pursue, and plenty of others for the interesting discussions, research and otherwise.

Thank you to all those who made my time at Stanford as enjoyable as it was, in particular Dylan Debbaut and Sveva Fornari.

Finally, I would like to thank my other family, my incredible friends: Maxime Aboukrat, Tristan Besson, Remi Bouché, Cyril Deroy, Antoine Mougeot, Frederic Jacobs, Romac Ruetschi, with a special mention to Victor Saade for inspiring me to always reach forwards.

Palo Alto, March 15th 2019 D. B.

Contents

Introduction

The use of software is ubiquitous in the modern world, where much of the infrastructure is dependent on the reliable operation of computer programs. Before they are executed to accomplish their designated tasks, programs are written and represented as text, and are usually designed by humans. This is a difficult and intricate affair, and as such it is error-prone and time-consuming for the developer.

To reduce this inherent complexity, programming languages provide abstractions and formalisms that promote re-use and consistency in problem-solving. They exhibit a uniform interface through a formal grammar, governing the way their logic can be composed. By this definition, the idea of deriving a program's properties from first principles is undeniably powerful, as demonstrated by the powerful tools developed in logico-deductive settings, from static analysis to constraint solving, the verification of properties such as memory safety, and even automated theorem proving.

Despite operating at a high level of abstraction, the act of programming remains for the most part a human endeavour. As such, it presents a set of regularities that couples with the formalism of a language's definition. By analogy, while natural language could be assumed to be infinitely rich in its expressive power, the majority of utterances are restricted to a very limited subset of content, vocabulary and grammar. This insight into the regularity of human speech has led to major advances in the field of Natural Language Processing, where instead of attempting to model language from first principles, researchers focused on capturing the regularities in the set of available communications. The proposed models can embody the current state of language, instead of producing grammatically correct but convoluted forms. They also do not require explicit reasoning about the intent of the content they observe: if language is assumed to be a vehicle for communication, it can be argued that its meaning can only exist in the context of its interpretation. Instead, these models assume that multiple intents can co-exist in the model of language, a superposition of meanings that only collapses when it observed, *i.e.* when language actually manifests itself as a communication, heard or read, rather than when it is spoken or written.

In this work, we argue that, conceptually, the same insights can lead to models of source code that are able to capture the programmer's intent and its properties without requiring its explicit interpretation, which can be understood as its execution. Instead, we can analyze the regularities of its unrealized form. To do so, we must manage some of source codes' characteristics. Programs are inherently multi-modal, existing concurrently at multiple levels of abstraction, from raw textual input down to binary executable programs, each with their own format, structure and accompanying semantic information. The inability to account for these characteristics would inevitably hinder a models' abilities to truly capture program semantics.

To this end, we propose to leverage a universal, standardized representation of code in the Abstract Syntax Tree (AST). Generated by the language's lexer, the AST is an intermediate representation, lying between raw text and low-level machine code. As such, it structures the programmer's input, a noisy but semantically rich signal, around the language's fixed grammar, *i.e.* its syntax, expressed through keywords and their interactions. Working at this level of granularity retains local co-occurrences around a considered token, regularities defined through its local neighborhood. This so-called *context* is syntactically structured, providing the model with rich information in the way dependencies form in source code.

Leveraging such structure for learning code representations has been used in the field before, but it usually relies on hand-crafted features or the measurement of a constant, restricted set of structural properties. By analogy with advances in computer vision, we propose to advance beyond edge-detectors in order to learn more semantically-relevant filters on code graphs. These deeper representations capture regularities in source code, providing rich semantic representations, which act as valuable features which we exploit on several standard tasks, assumed to be proxies for a understanding of code's behaviour, properties or intent.

Our proposed approach can be seen as a generalization of the popular TRANSFORMER architecture to arbitrarily structured input, namely in the form of graphs. As such, it enables concurrently leveraging the local regularities that lie in a node's neighborhood as well as the global context in which these elements interact. This concurrently produces deep contextual representations at the node and at the graph level, opening applications in either of these task spaces. We also show that the model exhibits multi-task learning capabilities. By pre-training our model on a semi-supervised task, for which data is abundant, we provide the model with a strong initial inductive bias, which can then be transferred to more specialized, fully supervised tasks after fine-tuning.

1 On the Naturalness of Source Code

You shall know a word by the company it keeps.

*F*irth, J. R. 1957

1.1 A mathematical description of language

Given its importance to the transmission of information, the understanding and description of the rules of language is paramount. Under the impulse of linguists, initial models were logic-based, like expert systems in the early ages of AI. However these categorical grammars are not predictive models, limiting their usability in downstream applications. They also do not reflect natural language, in the sense that a grammatically correct formulation is not necessarily informative or usable: the majority of human communication is constructed only from a subset of these rules, which can also be bent for stylistic or semantic effect while retaining meaning.

Indeed, an exhaustive formulation is not always necessary to understand the meaning being conveyed. Languages often contain many more words than are commonly used: everyday utterances operate on a restricted subset of the exhaustive description of language. Early on, linguists, such as J.R. Firth, recognized that this repetitiveness of natural language could be leveraged to understand meaning. Zellig Harris laid down a foundational basis for thinking of language in probabilistic terms.

> All elements in a language can be grouped into classes whose relative occurrence can be stated exactly. However, for the occurrence of a particular member of one class relative to a particular member of another class, it would be necessary to speak in terms of probability, based on the frequency of that occurrence in a sample.
>
> *– Z. Harris (1954)*

Early on, these these regularities were be described through empirical observations, like Zipf's law (1935) which states that the frequency of any word is inversely proportional to its rank in the frequency table, as shown in Fig. 1.1.

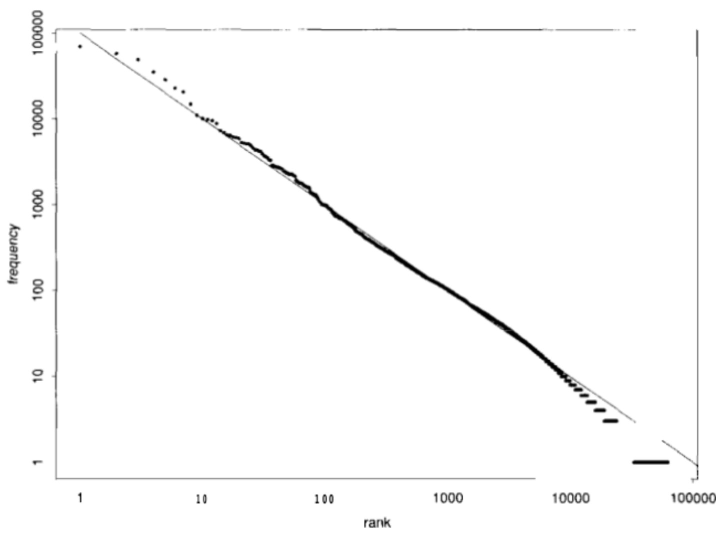


Figure 1.1 – Zipf's Law on the Brown Corpus (Source: *Alan Du*)

One way of formally measuring the regularity of a language is to compute its entropy, which measures irregularity, disorder or uncertainty. Shannon himself used the definition of entropy he had developed in his Information Theory treatise to measure regularities in the English language at a character level, discovering surprising regularity [Shannon, 1950]. He split text into successive groups of n characters, a rolling window which observes random variables drawn from a 27 letter alphabet. These chunks are called n-grams. From there, the entropy of the distribution of n-grams measures how "suprising" these elements are in the context of the English language. Measures showed an empirical cross-entropy of 1.3 bits. Compared to a uniform estimate of 4.75 bits, this result demonstrated significant regularities in the structure of text.

Of course this is only an upper bound, given that the true probability distribution of natural language is ill-defined: it is difficult to assume the existence of a ground truth, a universal rule of language. Note that the debate is still open in the linguistics community, with works like Noam Chomsky's Universal Grammar arguing for an underlying structure of human communication. This is again a fundamental difference in framing between statistical linguistics and language derived from first-order principles. The former defines language as the set of existing communications, with the ability to adapt to whatever linguistic constructs appear or disappear. The latter assumes an underlying universal representation of language which is to be uncovered. While attempts at proving either postulate seem ill-fated, statistical linguistics have shown great promise thanks to their pragmatic assumptions, opening additional degrees of freedom in the treatment of language.

This idea to empirically capture the regularities of language gave rise to the field probabilistic language modelling. Given some vocabulary or lexicon $\mathscr{V}$, the set of all possible valid sequences within a grammar $\mathscr{G}$ is defined as $\mathscr{T}$. A language model is then defined as the probability distribution over all possible sequences $t \in \mathscr{T}$.

In practice, this distribution is learned from a maximum likelihood estimate (MLE) over a corpus $C \subseteq \mathscr{T}$, again estimating from some set of text. Fundamentally, the most common approach is the same as [Shannon, 1950]: extending the character level regularities to words, the goal is to sequentially estimate how likely a word w_i is to follow a set of other tokens, called a *context*.

Then, the chain rule along with the joint and conditional probabilities [1] of each word in t yields the general language model formulation, in the form of the likelihood $p(t)$:

$$\begin{aligned} p(t) = p(w_1 \dots w_n) &= p(w_1)\dots p(w_n|w_1 \dots, w_n) \\ &= \prod_{i=1}^{n} p(w_i|w_1 \dots w_{i-1}) \end{aligned} \tag{1.1}$$

Continuing the analogy to [Shannon, 1950]'s experiment, the "quality" of this model can be measured by its ability to reduce the uncertainty of a word w_i, or equivalently maximize its likelihood, given some context. In a sense, this measures the models ability to capture any regularity in the set $\mathscr{T}$. This regularity is captured through a *perplexity* metric, or in its log-form, its cross-entropy. For a given language model $p(t)$, where context is defined as an n-gram, the cross-entropy H is expressed as:

$$H(p) = -\frac{1}{n}\log_2 p(t) = -\frac{1}{n}\sum_{i=1}^{n}\log_2 p(w_i|w_1 \dots w_{i-1}) \tag{1.2}$$

The underlying postulate for this treatment is the *statistical semantics hypothesis*, which states that statistical patterns of human word use can be leveraged to figure out what people mean [Weaver, 1955, Furnas et al., 1983], formalizing the intuition that human language operates on a subset of all possible linguistic formulations. This paved the way for early applied successes as well. The empirical distributions were learned from new, large-scale corpora such as the Canadian parliamentary proceedings or similar outputs from the European Parliaments, both of which are translated, and hence aligned across multiple languages. The early language models were then used for initial automatic translation efforts, such as the early Georgetown Experiment [2] or the SHRDLU conversational agent developed at MIT (1960).

[1]The assumptions behind the conditional distribution are discussed in more detail in Section 2.1.1.

[2]Which ambitiously predicted that automatic translation would be solved in the next "three to five years", back in 1954. [Hutchins et al., 1955]

1.2 Learning large-scale textual embeddings

The *statistical semantics hypothesis* that motivates the probabilistic treatment of language is only a general statement that subsumed more specific hypotheses, themselves driving different approaches to the problem of language modelling.

Initial approaches worked on the simple *bag of words hypothesis*, wherein the comparison of word frequency distributions within a document and across documents is expected to reveal relevant properties of the corpus [Salton et al., 1975]. Its simplicity yet effectiveness made it the de-facto standard for decades, with refinements to the counting mechanisms yielding tools like TF-IDF, which dominated textual Information Retrieval (IR) benchmarks for years.

However these methods rely on the factorization of large (co-)occurrence matrices, which can quickly become intractable for large corpora and vocabularies. Instead, a new class of systems was proposed based on the *distributional hypothesis*, stating that patterns which co-occur in similar contexts tend to have similar meaning [Harris, 1954, Firth, 1957, Deerwester et al., 1990]. This would become the dominating paradigm for years to come.

While initial approaches would still rely on low-rank factorization techniques, the advent of neural estimators quickly showed promise. Indeed, a language model is well-suited for an optimization-based formulation: a probabilistic model is to be learned (Eq. 1.1) under a cross-entropy minimization constraint (Eq. 1.2). Language models could hence be parameterized by some non-linear function, such as a neural network [Bengio et al., 2003, Collobert and Weston, 2008], which replaces the explicit probabilistic formulation with a non-linear predictor. A nuance to the apparent rift between the two models was added by [Levy and Goldberg, 2014b], showing that neural language models in fact reduce back to the matrix factorization approach.

The parameterization of language models allows efficient encoding, projecting the input into a vector space, whose distance metric should represent characteristics of the language that is being modeled. This word representation is called an *embedding*.

While representations like co-occurrence matrices are sparse representations, parameterized models can learn dense and compact representations, which only depends linearly on the size of the vocabulary - instead of a quadratic relationship for sparse representations. Compressing the representation space makes it much easier to actually encode relationships, as distances and relationships between semantic entities can get diluted in high-dimensional spaces. They also provide easy ways to distribute pre-trained word embeddings for large vocabularies, that can easily be plugged in to an downstream task [Turian et al., 2010, Mikolov et al., 2013, Pennington et al., 2014], for massive performance gains across the board with little overhead.

On the surface, these embeddings managed to capture stunning semantic similarities between words, with the now famous analogical relations such as $v_{king} - v_{man} + v_{woman} \sim v_{queen}$, where v_w is the vector representation of word w [Levy and Goldberg, 2014a]. This is a hallmark of *distributed representations*, as the meaning they capture is distributed across multiple feature dimensions, enabling the previous semantic arithmetic. However, if the corpus on which these associations are learned signal biased associations, these biases will persist in the representations. Despite qualitatively reasonable results, it is a reminder that these embeddings require deeper probing to study the relationship with ground truths that they actually carry [Bolukbasi et al., 2016, Caliskan et al., 2017]. As always, the model's quality is highly dependent on the dataset's quality. Most recently, OpenAI's GPT-2 proved exactly this point, providing state-of-the-art performance solely through the collection and processing of a high quality dataset [Radford et al., 2019], containing much more diversity than the traditionally-used Wikipedia or Brown corpora.

1.3 The Naturalness assumption

Like human language, code can be described formally by a set of tokens - a vocabulary - and the rules through which they are allowed to interact - a grammar. In the programming language community, the majority of research has been dominated by the *formal* approach. Since programming languages exist specifically within the class of formal languages, practitioners argue that it is natural to approach software-related problems within the same realm.

These rigorous methods do indeed show tremendous promise in their ability to reason about program logic [Suter et al., 2011] or verify their properties [Calcagno and Distefano, 2011], find vulnerabilities [Livshits and Lam, 2005], prove correctness with formal verification [Agerholm and Larsen, 1999, Halpern et al., 2001], provide elegant and powerful abstractions for their description [Landin, 1966].

Foundational to the treatment of code in general, these grammars are essential to the programming pipeline. The raw textual representation of code needs to be processing through parsers, interpreters and compilers, whose architectures are expressly created to capture computational properties of the considered formalism [McCarthy, 1960].

However, in dealing with source code in an environment derived from first principles, the rich expressiveness of formal syntax can sometimes obstruct the treatment of actual code, which people write.

It is not difficult to imagine that, like natural language, where the majority of utterances fall in a fairly restricted and repetitive set of words, source code could show some form of regularity. This hypothesis seems particularly plausible given that most code is written by humans, which usually exhibit a similar bias towards simple and repetitive forms. The formal nature of source code even encourages the re-use of patterns or even entire blocks of logic, and imposes strong "best practices" or conventions [Gabel and Su, 2010].

To test this hypothesis, [Hindle et al., 2012] measured the repetitiveness and predictability of source code, comparing it to that of natural language. This is called the *naturalness hypothesis*, proposed by [Hindle et al., 2012]. The authors reproduced a similar methodology to [Shannon, 1950], computing the cross-entropy of n-gram models trained on the textual representation of source code. By measuring and comparing regularities across different levels of granularity, they are able to characterize their distribution, answering three structural questions:

RQ 1 Do n-gram models capture regularities in software?
RQ 2 Is the local regularity that the statistical language model captures merely language-specific or is it also project-specific?
RQ 3 Do n-gram models capture similarities within and differences between project domains?

This foundational study found that indeed, source code follows similar co-occurrence patterns and exhibits repetitiveness in its local context, similar to the regularities found in natural language. On average, source code even presents lower entropy (with a lower bound around 2 bits, compared to 7 bits on their corpus of natural language, and 20 bits for a uniform distribution). However this lower entropy is not solely the product of the smaller vocabulary or simpler syntax of the Java language which they used. Indeed, they show in *RQ 2* that each project they considered has its own "flavor" of regularity, meaning the language models are capturing regularities beyond differences in vocabulary. They even show that these discrepancies persist across domains, with code specific to a certain application also showing its own form of regularity.

This line of research provided a sound footing for the advent of statistical learning on source code. The fact that source code exhibits regularities at levels analogous to those found in natural language strongly suggests that methods that have found much success there should transfer well to the study of source code. Indeed, the fundamental shift to corpus-based statistical methods in the field of NLP has enabled tremendous advances in the application of real-world text-based systems, across tasks, from Information Retrieval to translation. If the fundamental assumption that motivated this paradigm shift in the first place can be related to the *naturalness hypothesis*, then it can also be argued that the same statistical treatment of source code could be conducted, hopefully to similar avail.

1.4 Distributed representations from large source code corpora

With the advent of large-scale source code repositories, from massive open-source projects like Linux to the democratization of the containers that host them, like Github or SourceForge, an unprecedented amount of data about code is now available. This data also includes a swath of labels or meta-information at different scales, from commit histories modeling program edits [Yin et al., 2019] or code reviews [Zimmermann et al., 2004] to multi-modal programmer interactions in question & answer platforms like StackOverflow.

The broad availability of training data is precisely what empowered the statistical revolution in Natural Language Processing (see Section 1.2). If the regularities highlighted by the *naturalness assumption* were to hold, these successes should intuitively be reproducible in a new, data-driven approach to modelling source code properties. This idea led to the development of "*big code*", a data-centric way of mining software repositories. [Raychev et al., 2015]

From this assumption, new class of tools which do not require the formal correctness of logic-based approaches were designed [White et al., 2015, Vechev and Yahav, 2016]. The ability to leverage information provided by other people's code through the modeling of best practices or common patterns could greatly benefit the development process, making it faster and potentially helping reduce the number of errors the programmers makes. Code editors (IDE) could natively help developer's with tasks as mundane as completion or API-recommendations [Hindle et al., 2012, Bhoopchand et al., 2016, Bielik et al., 2016], leveraging the collective knowledge of other developers [Bruch et al., 2009]. The repetitiveness of many tasks is a strong motivator for their replacement by automated methods, from boilerplate completion to the promotion of best practices [Allamanis et al., 2014, Pu et al., 2016]. Powerful maps could also be drawn between natural language and source code [Yin et al., 2018a, Oda et al., 2015, Gulwani and Jojic, 2007], supercharging code search engines [Gu et al., 2016], enhancing the quality of code documentation [Neubig, 2016, Hu et al., 2018], assisting bug discovery [Williams and Hollingsworth, 2005] or even bringing the act of programming closer to the *literate programming* dream of Donald Knuth [Knuth, 1984].

Early approaches to the statistical treatment of source code corpora naturally followed the waves of representation learning for natural language, assuming the distributional assumption would hold true for code as well. Initial language models, such as [Wang et al., 2016, Raychev et al., 2014, Dam et al., 2016], were learned on raw text data, providing evidence that the naturalness assumption held. For example, [Allamanis et al., 2015] learned distributed representations of variables and methods, finding that they were indeed able to encode common semantic properties from the regularities present in source code. [Alon et al., 2019] also found evidence of semantic arithmetic in their embedding space, dubbed CODE2VEC.

These representations - and their variants like [Mou et al., 2016] - can then be used to predict sequences of identifier sub-tokens [Allamanis et al., 2015], API calls [Acharya et al., 2007, Nguyen et al., 2017], to review student programming assignments [Piech et al., 2015] or map the solution to the original problems [Mou et al., 2016]. They can be used as an advanced auto-completion tools [Hindle et al., 2012, Bhoopchand et al., 2016], including for user-provided tokens like Variable Names [Raychev et al., 2014, Allamanis et al., 2014], which can be useful for example to deobfuscate Android applications [Bichsel et al., 2016].

The complete meaning of a word is always contextual, and no study of meaning apart from a complete context can be taken seriously.

*F*irth, J. R. 1935

1.5 Contextualized representations

By computing a single representation for each word, *i.e.* embedding a vocabulary, distributional methods like WORD2VEC failed to capture much of the more nuanced richness and expressive power of natural language, ignoring polysemy entirely [Neelakantan et al., 2014]. Indeed, like Firth, inspired by Wittgenstein before him, [3] had noted, a word's meaning is inherently contextual.

Polysemy could theoretically be learned by associating each word to its different meanings, like a dictionary definition, similar to WORDNET [Fellbaum, 1998]. Instead, modern contextual representations refine the vector representation of a word based on the representations of its neighbours, *i.e.* its context. The most popular variant of this deep contextualization, ELMO [Peters et al., 2018a], aggregates from an arbitrarily sized context through the use of Recurrent Neural Architectures, a popular choice for aggregating information from sequences of variable length [Sundermeyer et al., 2012]. However, instead of using the final state of this representation as an embedding, ELMO proposes to linearly compose the successive internal states of the recurrent models themselves. In effect, the model learns to compose information from different scales, enabling precise modulation of a word's representation based on its context. These enhanced representations can then be appended as additional feature information to existing, static embeddings such as WORD2VEC, and show significant gains in a set of tasks on text.

In general, most representation or interpretation problems in NLP come from this difficulty to model ambiguities. They are usually grouped into two categories:

Semantic Understanding often requires the disambiguation of synonyms, the inclusion of global contexts, ...

Syntactic The mapping from syntax to meaning is not injective: the syntactic decomposition (such as through a parse tree) is not unique.

The same ambiguities are present in the context of source code. For example, at the single token level, the meaning of a language keyword is highly dependent on the context in which it is used. These specific tokens are very particular to programming languages, and act as

[3] "The meaning of a word is its use in the language. [...] One cannot guess how a word functions. One has to look at its use, and learn from that." - Ludwig Wittgenstein, *(Phil. Investigations, 80, 109)*

strong context modifiers [Deissenbock and Pizka, 2005]. Even with the ability to execute or evaluate code, much ambiguity still remains to be resolved, for instance it is not trivial to match variables to instances. If we are to provide useful tools for the development process, our models must necessarily take into account these contextual dependencies.

2 Code: A structured language

2.1 On the Markov Assumptions of Language Models

2.1.1 n-grams as Markov chains

n-gram models aim to predict a word w_i based on its n-neighbor history, *i.e.* its context $w_{i-n}, \dots, w_{i-1}$. In this sense, it can be interpreted as a Markov chain of order n [Markov, 1913]. This is a simple linear graphical model, wherein each word w_i is modeled by a single random variable whose values depends on its predecessors, as illustrated in Figure 2.1.

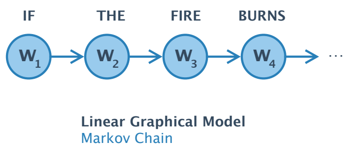


Figure 2.1 – n-gram language model as a Markov Chain.

This assumption serves to greatly simplify the language model formulation by linearizing the dependencies in the provided context. In Eq. 1.1, the likelihood of a given word w_i was conditioned on the distributions of all the previous words in the sequence. With the Markov assumption, this dependence is limited only to the individual contributions of the n previous factors:

$$p(w_i|w_1 \dots w_{i-1}) \approx p(w_i|w_{i-n}, \dots, w_{i-1}) = \frac{p(w_{i-n}, \dots, w_i)}{p(w_{i-n}, \dots, w_{i-1})} \tag{2.1}$$

In practice, this approach does show strong empirical evidence in the case of natural language. While in theory grammatical relationships can be arbitrarily distant and convoluted, in practice context is often extremely local. For example [Collins, 1997] showed that 74% of the dependencies in the Penn Treebank, a classic dataset for syntactic annotation [Marcus et al., 1994], were within a single word of distance. However, as language-oriented systems are designed to handle more and more complex tasks, this assumption may be detrimental. This is especially the case in more challenging tasks, which require relational reasoning on text, the modeling of long-ranging dependencies, and a variety of subtle linguistic properties such as polysemy, entailment, contradictions, ... These linguistics properties are currently benchmarked against complex multi-task settings like the General Language Understanding Evaluation benchmark (GLUE) [Wang et al., 2018a], on which linear context models fail miserably. This failure led to the larger adoption of LMs that adapt to the context of the word at hand, which greatly improves results on these semantic tasks - ELMo is for example used as the baseline for the GLUE benchmark.

2.1.2 Contextualized LMs as Markov Random Fields

By introducing more context into the estimation of the probability distribution over possible words, contextualized Language Models also modify the Markov assumptions made by n-grams, detailed in Section 2.1.1. Each word w_i is still assumed to be a random variable, an instance drawn from a graphical model. However, here the graphical model can be updated for a more general dependency structure. This class of models can be represented through a Markov Random Field (MRF), which defines an undirected graph as the graphical model instead of a simple linear chain, as shown in Fig. 2.2.

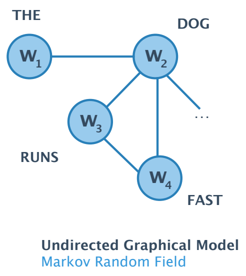


Figure 2.2 – A general Markov Random Field language model.

The Markov Random Field formulation expresses the language model as a probability distribution over C, a set of cliques (fully connected subgraphs). Using the notation from Eq. 1.1:

$$p(w_1, \ldots, w_n) = \frac{1}{Z} \prod_{c \in C} \phi_c(w_c) \tag{2.2}$$

where Z is defined as the normalizing constant:

$$Z = \sum_{w_1,\ldots,w_n} \prod_{c \in C} \phi_c(w_c) \tag{2.3}$$

Note that the definition of a clique is not restricting. The factor graph can contain cliques at the level of single nodes, edges (which are cliques of two nodes), motifs, ... This flexible definition opens up its application to any type of relationship between words, from the simple linear model - single-node cliques - of n-grams as a superset of Markov chains, to more complex cliques in linear context composition schemes like ELMo, as illustrated in Fig. 2.3.

This formulation is not used explicitly in modern language models, one reason being the prohibitive (exponential) cost of computing the normalizing constant Z when the defined cliques are too large or if there are many nodes in the graph. However, it provides a useful way to reason about the dependencies being captured by a given language model. For example, log-bilinear Neural Language Models [Mnih and Teh, 2012] (a generalized version of the popular WORD2VEC) can be seen as optimizing the pseudo-likelihood of a low-rank Markov sequence with cliques of 2 edges [Jernite et al., 2015].

It can also be suitable to use the formulation to sample from the language model: the un-normalized log-probabilities obtained in Eq. 2.2 can be used to find the most likely sentence within a set of sentences conditioned on a given context. In the ranking formulation, Eq. 2.3 does not need to be computed, rendering the model much more efficient. Gibbs sampling this likelihood distribution allows for sequential generation of sentences from the likelihood of the LM, as shown in [Wang and Cho, 2019].

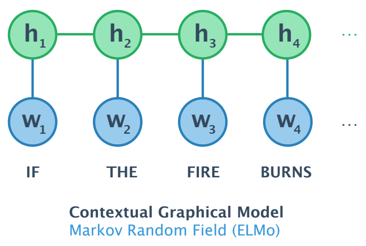


Figure 2.3 – ELMo as a Markov Random Field, specifically a Linear Chain Conditional Random Field.

Furthermore, the MRF formulation paves the way for the explicit capture of a priori structure in language models. Similar to what was proposed in [Dai et al., 2016a], the graph structure of the input data can be used as the conditional independence structure of the graphical model itself, as showcased in Fig. 2.4. If structural information is available, this restricts the range of coupling between each random variable. Since the model is undirected, no assumptions are made on how variables generate one another, but it provides a strong bias for a model to identify groups of dependent variables. Additionally, works like [Raychev et al., 2014] have noted that the undirectedness of a graphical model helps when the exact order (or direction of edges) is not always well defined a priori, reason why they leverage a Markov model rather than their directed counterparts, Bayesian networks. If a priori information is available about the structure of the data, it can be leveraged by the model to strengthen its inductive bias.

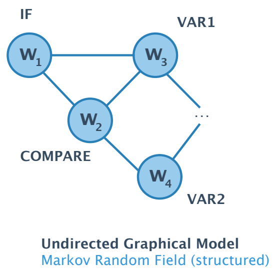


Figure 2.4 – Markov Random Field of a structured language.

Shying away from claiming a universal grammar, some descriptive syntactic structures can nevertheless be extracted from free-form text. These annotations have been developed over years through formal grammars and rich heuristics, like with parse trees or dependency trees, through rule-based matching like for Named Entity Recognition, or through statistical learning like in the case of Part-of-Speech (PoS) taggers. However, in many of these cases, the produced information is not unique. Given a sequence of words, multiple syntactical and semantic parses can be grammatically correct, and considered semantically correct in the absence of relevant global context. This is a major difficulty in the automatic understanding and treatment of text, as it makes it difficult to extract structural properties and additional token-level features and relationships, that could be of great help to augment current representations.

Source code however does not have this issue, at least with respect to its syntactical parsing. For this reason, it seems reasonable to leverage available information in attempts at characterizing the statistical properties of source code, an idea which has been leveraged since the early probabilistic models of source code [Gulwani and Jojic, 2007, Kremenek et al., 2007].

2.2 On the representation of source code

2.2.1 Idiosyncrasies of source code

While source code is deterministic in its syntactic parsing, this is only true at a given level of representation. Indeed, one of its specificities is the ability to live at different levels of abstraction. The programmer writes code similarly to the way they would write natural language: character by character, in a free-form textual environment. Some more visual, interactive or abstract environments has begun to arise but the majority of written code does not live in this plane. This textual representation is structured by formatting rules, which are more or less enforced by languages, but before any processing there is little difference between code and natural language, modulo the vocabulary.

The free-form expressions can however be passed through a set of specialized tools, unlike what is available for natural language. Only through this pipeline does source code exist in its different representations. While each language possesses its own arsenal of tools to process the programmer's input, with some even being shared at different levels of abstraction (*e.g.* through tools like LLVM compiler infrastructure [1]), each performs different passes to produce representations at different layers of abstraction.

These representations can range from cosmetic changes, like variable name obfuscation or constant replacement, operations that still live in the realm of text-based operations, all the way down to the production of low-level machine code. Indeed, like natural language which only takes meaning when it is processed - read or heard - source code's meaning is to express a computation, which is only realized when it is executed. However, as work on natural language has shown, the statistical properties of text are a powerful vehicle for meaning, even without direct interpretation. Continuing the analogy, we can assume that source code as written by the programmer encapsulates intent, and such intent can be captured without execution.

Like with syntax, semantic information is generally speaking better defined for code thanks to the formal rules that govern its usage. Semantic annotations are not always available however, depending on the context or the language's properties. For example, some languages can provide things like a full data-flow map, strong type information or variable scoping at compile time. Other languages however allow behaviours like dynamic linking or lazy evaluation that make it impossible to know the full context before execution, and sometimes even during execution, leading to undefined behaviour. Again, this motivates the scope at which this work will operate, as we ideally want to remain as independent from the language's specific constructs as possible. This may drop some richness from the input signal, but should allow for a more flexible formulation when extended across tasks or opened to multiple languages.

Remaining high up on the ladder of abstraction [2] also allows us to stay closer to the downstream applications which we envision as being more user-oriented. Some works focus on reasoning about program properties, in which case it makes sense to drop down to lower-level signals, but we frame the problems at hand as inherently high-level.

2.2.2 The structure of code

Concretely, we identify four main representations of source code that could be relevant to our application, summarized in Fig. 2.5.

[1] https://llvm.org/

[2] http://worrydream.com/LadderOfAbstraction/

Raw text.

Before being pre-processed through any language-specific pipeline, source code lives as raw text. It is unstructured beyond the linting rules of the language, some being strongly enforced like in Python where white-space provides structure to the code, others like Java using braces to indicate syntactic context. This representation is used in several prior works, notably because this representation is close to raw text and it retains useful information such as programmers' comments. These include [Bavishi et al., 2018, Tufano et al., 2018, Pu et al., 2016, Hellendoorn et al., 2018, Bhoopchand et al., 2016, Iyer et al., 2016, Allamanis et al., 2014].

Abstract Syntax Tree.

An Abstract Syntax Tree (AST) captures the essential of the code's structure, relegating much of the noisiness of the programmer's input down to node features, and removes some proportion of syntactic redundancies like punctuation or composite keywords. Each node is a specific token provided by the lexicon of the language, and edges are created between them based on the parse rules defined by the language's grammar. ASTs usually live at a relatively high abstraction level, with many languages sharing many commonalities in their parse tree structures and elements: every language needs to represent some fundamental blocks like arithmetic operations or function calls. Since the parse is mostly syntactic, the specific language implementation should have limited effect on the end structure of the tree.

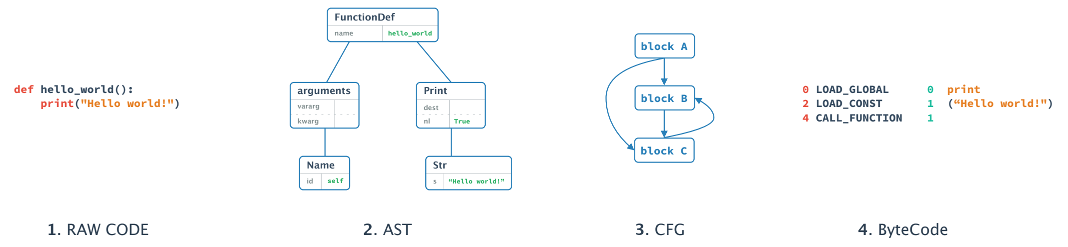
Overall, the AST presents an interesting middle-of-the-road representation in that it is close enough to the programmer's input that the specificities of their input can be recovered (*e.g.* variable names, string literals, ...) but they do not impact the underlying structure of the code too strongly. This trade-off has been leveraged in works like [Allamanis et al., 2018b, Maddison and Tarlow, 2014, Raychev et al., 2014, Yin et al., 2018b, White et al., 2015, Hu et al., 2018, Neamtiu et al., 2005]

Control-Flow Graph.

The Control-Flow Graph (CFG) is a high-level representation which shows all paths that may be traversed throughout program execution [Allen, 1970]. In this regard it incorporates operations like explicit branching for conditionals, loops, jumps, ... Nodes are defined as "blocks" of code, logical sections of contiguous context, between which there is no ambiguous edge. As soon as branching occurs, this split is added as an edge. Unfortunately, the CFG is not available for all languages. In PYTHON for example it is difficult to obtain because of its dynamic behaviour: at compile-time there is no way to know where an edge will lead, as this information is resolved at run-time. This representation is used in works such as [Tufano et al., 2018, Ben-Nun et al., 2018], with others leveraging similar abstraction levels in variable-flow graphs [Allamanis et al., 2018b] or program dependence graphs [Gabel and Su, 2010].

Machine code.

At the lower end of the abstraction spectrum, machine code represents a low-level representation. It usually sits just above executable code, with the exception of languages run inside specific Virtual Machines, such as Java bytecode. This intermediate representation is ridden of much of the noise introduced by the textual form input by the programmer: it synthesizes the written code in terms of atomic operations that can later be run on a dedicated processor. Bytecode contains much more semantic information, as provided by the compiler, but it is also far removed from the programmer's input. This makes it suitable for low-level prediction tasks, like loop-invariance [Si et al., 2018], static analysis [Koc et al., 2017], low-level API-modelling [Nguyen et al., 2016] or thread coarsening parameter prediction [Ben-Nun et al., 2018], but is not particularly suitable for user-interfacing applications.



```
def hello_world():
    print("Hello world!")
```

```
0 LOAD_GLOBAL      0  print
2 LOAD_CONST       1  (“Hello world!")
4 CALL_FUNCTION    1
```

Figure 2.5 – Representations of source code.

Given the considerations made explicit above, we decide to leverage the AST as our base representation. It provides a good balance in that it provides commonalities across languages, a structured representation in between semantically rich but syntactically poor machine code and flat raw text. This means both semantic, machine-specific tasks can be tackled without renouncing the ambition of building tools directly applicable to the programmer's coding interface. Its structure is readily available and can easily be represented as a graph, in the form of a tree.

2.3 The regularities of structured code representations

Much like natural language, code presents several modes of regularities, due to its repetitive and hence predictable nature [Hindle et al., 2012]. Unlike natural language however, code has a deterministic structural representation in the form of its AST. As such, the underlying graph representation of source code could be leveraged as an additional source of information to learn the idiosyncrasies of code.

One notable way of identifying regularities in a graph is by means of counting network motifs – sub-graph structures of size k that re-occur within a given network. The recurring presence of specific network motifs in a graph reflects its main functional properties, by ways of identifying the fundamental building blocks of the network.

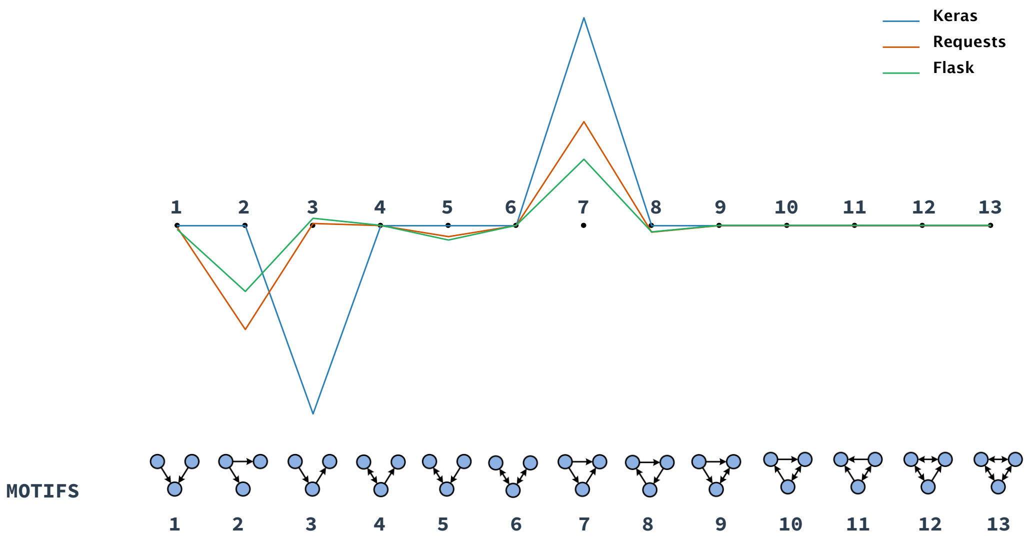


Figure 2.6 – Distribution of z-scores for 3-motifs.

Producing the exhaustive count of motifs in a graph is computationally challenging, therefore most of the analysis on large graphs are limited to 3-motifs. Fig. 2.6 shows the distribution of z-scores for 3-motifs performed on 3 popular PYTHON projects: `keras`, `requests`, and `flask` (respectively a Deep Learning library, an HTTP client, and a Web framework). The choice of those 3 projects was driven by a diversity criterion, to provide results across a wide spectrum of application domains. All three are also used in downstream tasks detailed in Chapter 4.

The z-score measures how many standard deviations away each motif count is with respect to an ensemble of randomly shuffled graphs with the same degree sequence. Therefore, it is worth noting that each project shows a certain degree of regularity (*e.g.* the common peak on the feed-forward loop - motif # 7). Conversely, some of the motifs are not present at all, in sheer contrast with the results usually obtained in networks describing a natural phenomenon. Although further analysis is needed to extend regularities to colored graphs – considering each motif with typed AST nodes – this initial analysis hint at strong regularity properties in structured representations of source code, akin to those found in the textual representation, shown by [Hindle et al., 2012].

Last but not the least, the different z-score profiles obtained from the analysis of `keras` vs. `requests` and `flask` is evidence that a rule-based approach to extract the constituent components of an AST would be a daunting (and probably sub-optimal) strategy. This is a strong motivation for a learning-based approach.

2.4 Contextualized representation learning on graphs

2.4.1 Deep Learning on Graphs

As described in Section 2.1.1, a Markov Random Field formulation opens up the possibility for structure to be explicitly leveraged. However, several subtleties must be introduced to leverage this formalism in the case of graphs, even conceptually.

First, while the graphical model itself is undirected, current models of context do encode directionality. The degenerate case of Markov Chains encodes a backwards direction in the context: the likelihood of the current token depends on its history. Clearly, this is not sufficient to capture rich context, as dependencies cannot be assumed to be always observable in a word's history, even in trivial cases like adjectives applying to nouns before them or time indications changing the tense of a verb.

Bi-directional language models (BiLMs) [Peters et al., 2017] were introduced to deal with this issue, forming forwards and backwards dependencies in context through a pair of encoders that operate in either direction. ELMo [Peters et al., 2018a] directly leverages this architecture, composing the results from both encoders but also from their hidden states - though unlike previous approaches, both directions share their parameterizations. Other models can present even more complex relational structures: the General Purpose Transformer (GPT) [Radford et al., 2018] uses a bilinear encoder (backwards and forwards contexts) but only a forwards context for the decoder. This directionality has no place in graphs, where neighborhoods of a given node usually show large variability in size and connectivity, with exceptions like balanced trees, lattices or fully-connected graphs, and are unordered. Overall, graphs often show complex topological structures with little spatial locality. This limits the out-of-the box applicability of contextual representations taken from NLP.

All the formulations described until now depend on a linear context, be it through a linear Markov chain (Eq. 2.1) or through a linear definition of the cliques in a MRF (Eq. 2.2). It is linear in the sense that there is always a direct path between elements of a context. Again, this assumption does not hold in the case of graphs, where connectivity patterns can be arbitrarily formed. This linearity is also related to the importance of ordering in most formulations: the order of the context has an impact on the final representation. Here again, we must present a permutation-invariant formulation of a neighborhood, the analog of context on graphs. Graphs do not incorporate the notion of node ordering, nor can we define a clear fixed reference point for pseudo-coordinate systems. Note however that the notion of distance can still be defined through the notion of hops. This can be useful, as models like the log-bilinear LM [Mnih and Teh, 2012] leverage distance-dependent likelihoods.

For several years, recurrent neural LMs overtook their linear competitors thanks to their ability to incorporate contexts of arbitrary length [Józefowicz et al., 2016, Merity et al., 2018, Melis et al., 2018]. Yet in practice, this advantage struggles to hold its weight, with issues like vanishing or exploding gradients [Hochreiter et al., 2001] hindering the ability of RNN-based models to hold up to the now dominant self-attentive feed-forward architectures [Peters et al., 2018b]. These new architectures returned to fixed contexts, defined as constant-sized windows in either direction. With the strong influence that the input sequence length has on the computational footprint of these models, they are usually restricted in size. This means that dependencies outside the context window can't be modeled, but with input lengths growing up to 512 words in recent work like BERT [Devlin et al., 2018], it can be assumed that most local dependencies are captured. In comparison, LSTMs have been shown to capture up to 200 words of context on average [Khandelwal et al., 2018]. It still remains to be seen how to retain global context however, for example throughout a paragraph or an entire file. Hybrid approaches like the TRANSFORMER-XL [Dai et al., 2019] do try to bridge the gap by adding recurrence, but a method dealing with graphs would ideally be capable of handling neighborhoods of arbitrary sizes.

A natural formulation to deal with these many of the aforementioned subtleties can be found in the scope of machine learning on graphs. Early signal processing on non-euclidean domains [Bronstein et al., 2017], graph kernels [Yanardag and Vishwanathan, 2015] and spectral formulations [Bruna et al., 2013, Defferrard et al., 2016] paved the way for a wave of neural approaches. Graph Neural Networks (GNNs) [Gori et al., 2005, Scarselli et al., 2009] provide a powerful tool for machine learning on graphs, thanks to their ability to recursively incorporate information from neighboring nodes in the network [Battaglia et al., 2018], naturally capturing the graph structure simultaneously with the nodes' features. They are able to learn vector representations of nodes and graphs in an end-to-end fashion, encoding structural and feature information in the embedding space.

Under this model, GNNs have achieved state-of-the-art performance across a variety of tasks, such as node classification [Kipf and Welling, 2016, Hamilton et al., 2017], link prediction [Zhang and Chen, 2018, Schlichtkrull et al., 2018], graph clustering [Defferrard et al., 2016, Ying et al., 2018b] or graph classification [Ying et al., 2018b, Dai et al., 2016b, Duvenaud et al., 2015]. These tasks occur in domains where the graph structure is ubiquitous, such as social networks [Backstrom and Leskovec, 2011], content graphs [Ying et al., 2018a], biology [Agrawal et al., 2018], and chemoinformatics [Duvenaud et al., 2015, Jin et al., 2017, Zitnik et al., 2018].

As discussed in Section 2.2, a graph structure would intuitively serve as a natural support for the representation of source code. This idea, along with insights on how to actually deal with this structure in the context of language models motivates the treatment of source code graphs with GNNs.

While this work is not the first to recognize the potential upsides of this framing, the application of GNNs to structured representation of code is still in its infancy. Several works leveraged structured graphical models for probabilistic models of source code, usually through parse trees [Maddison and Tarlow, 2014, Bielik et al., 2016]. Unlike previous works where hand-crafted features were used as node features [Raychev et al., 2014] or as explicit semantic edges [Allamanis et al., 2018b], this work leverages existing syntactic relationships between the different elements to enhance the predictive capabilities of the model. Other work like [Alon et al., 2018] does also leverage the graph structure, but linearizes the graph by running random walks on the code graph. This is reminiscent of early node embedding methods like NODE2VEC [Grover and Leskovec, 2016], which were effective in capturing relatedness in graph neighborhoods but whose applications where limited due to their expensive computation process.

As detailed in Section 2.2, we decide against incorporating these types of semantic edges. While they are shown to be useful in the applications showcased by [Allamanis et al., 2018b], these additions are difficult to obtain consistently across languages. For example, a “variable last used/written here” edge is not available in Python but can be extracted from C#, similarly to type information or other data-flow information. This restricts the usage of the method to particular subsets of languages for which rich semantic information is available at compile-time. Additionally, it requires that the entire graph be processed each time new information should be added to the graph, which incur massive pre-processing costs, possibly rendering the application to downstream tasks impossible.

2.4.2 CODESAGE: A SAmple and AGgregate framework for contextual token representations

Methodology

Inspired by the work of [Hamilton et al., 2017], we set out to leverage the graph structure of code to produce low-dimensional embeddings of tokens with a GNN, namely the GRAPHSAGE approach. Its ability to handle unseen graphs - *i.e.* work in an *inductive* setting - is appealing as good generalization to unseen nodes and graphs is essential in this application.

The key idea of this method is to learn aggregation functions rather than individual node embeddings, enabling the learned aggregators to run on any unseen graph whose node features are aligned with the original. Another key contribution is the ability to efficiently batch training graphs, which allows for efficient computation even on very large graphs, a setting which is conceivably common when dealing with source code. This is done through the construction of computation graphs, where neighbours are sampled to create a subgraph of the neighborhood of the considered node. These computation graphs can then be batched for efficient learning.

We adopt the GRAPHSAGE approach to learn node embeddings of tokens in the code graph. More formally, we define the graph $\mathcal{G} = (V, E)$ as a representation of a piece of source-code, specifically its AST form. This representation is dependent on the particular experiment, as detailed in Section 4.1.2, but in this case we simply generate the AST for a set of PYTHON files. We define the set of input features $\{\mathbf{x}_v, \forall v \in V\}$, which in our experiments are defined as a one-hot encoding of the token type provided by the AST. The rest of the forward propagation is defined in Alg. 1, taken from the GRAPHSAGE formulation.

Algorithm 1: GRAPHSAGE embedding generation (i.e., forward propagation) algorithm (from [Hamilton et al., 2017].

Input : Graph $\mathcal{G} = (V, E)$; input features $\{\mathbf{x}_v, \forall v \in V\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \to 2^V$

Output: Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

```
1 $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$
2 for $k = 1...K$ do
3     for $v \in \mathcal{V}$ do
4         $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$
5         $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
6     end
7     $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
8 end
9 $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$
```

The main motivation behind this approach was to eventually tweak the two main contributors to the learning process in GRAPHSAGE, adapting them to the setting of learning structured representations of source code: the neighborhood function $\mathcal{N}$, and the aggregation scheme AGGREGATE.

On the choice of a sampling procedure.

The sampling procedure is the main artifact that defines a neighbourhood in GNNs. The neighborhood function $\mathcal{N}$ also defines the message-passing pathways, the edges on which information is propagated between nodes. In this sense it provides much of the structural information for the model. In the case of the AST, these edges are purely syntactic, representing the structure of source code in this parsed form. As previously discussed, concurrent approaches define different edges on which the sampling can occur, like the semantic edges proposed by [Allamanis et al., 2018b]. While we elect not to leverage this specific scheme because these annotation are not widely available, even in a popular language like PYTHON, we hypothesize that other custom sampling methodologies that could strengthen the inductive bias learned by the model.

For example, we know that certain keywords in a language already encode specific relationships. Conditionals have a well defined structure and hierarchy, iteration operators define a inherent structure through loops, ... None of these require additional information from the processing side, meaning they can be leveraged "for free" provided expert knowledge in designing the system, as a deterministic subset of the CFG. In a sense, this can be seen as providing the model with a pre-defined knowledge of the language's grammar or syntax in hopes that it can be informative for computing meaningful representations.

While most research has been focused on more computationally efficient sampling [Chen et al., 2018], the aforementioned direction can be defined as exploring semantically relevant samplers. Similar to relevance sampling in work like [Ying et al., 2018a], where random walks are run to find better neighbors to sample and learn from, we could extend the procedure to leverage biased walkers guided by prior knowledge of Programming Language principles, like partial evaluation or semantic edges, this time defined probabilistically instead of based on rules that are not always available.

On the choice of an aggregation scheme.

The specific syntactic constructions of code could also be reflected in the choice or design of an aggregator. The way in which information from the neighborhood is combined depends not only on the structure of the neighborhood but also on the labels and features of the surrounding nodes.

We also envision specifying particular aggregators for different edges or neighboring node labels. This insight probably motivated [Allamanis et al., 2018b] to choose *Gated-Graph Neural Networks* [Li et al., 2016], a flavor of GNNs which supports typed edges. Again, we do not have access to the same semantic edges, but we envision different aggregation schemes based on the considered context. For instance, aggregating the conditional in an `if`-expression shouldn't necessarily follow the same rules as a list of elements in a tuple. While in the first case each branch represents vastly different contexts, the second can be readily averaged without too much information loss. The same can be said for specific operators that can vastly change the meaning of an expression, such as logical operators. Shying away from actually executing these logical operators, as this would draw us closer to differentiable computing machines [Siegelmann and Sontag, 1992, Graves et al., 2016], we still lack ways of representing contrastive operators like negation: `if (A)` and `if (~A)` would have very similar representations in a classic aggregation scheme, as both tokens are equally important to the context.

Downstream Use

Another strength of the GRAPHSAGE formulation is its ability to function as an encoder, generating features in an unsupervised fashion, but also as a supervised learner. In line with the literature's approach to the considered tasks, we mainly focus on the use of GRAPHSAGE as a context-generator, able to provide features for downstream tasks. In this unsupervised setting, the training objective looks to produce embeddings $\mathbf{z}_u$ for a given node u that are similar for neighbouring nodes, and conversely dissimilar for disjoint nodes. The loss function defined on a considered graph $\mathcal{G}$ is defined as a standard negative log-likelihood:

$$J_{\mathcal{G}}(\mathbf{z}_u) = -\log\left(\,\sigma(\,\mathbf{z}_u^\top \mathbf{z}_v\,)\,\right) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log\left(\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n})\right), \tag{2.4}$$

where $P_n(v)$ is a negative sampling distribution, with Q the number of negative samples taken from it. σ is a standard non-linearity.

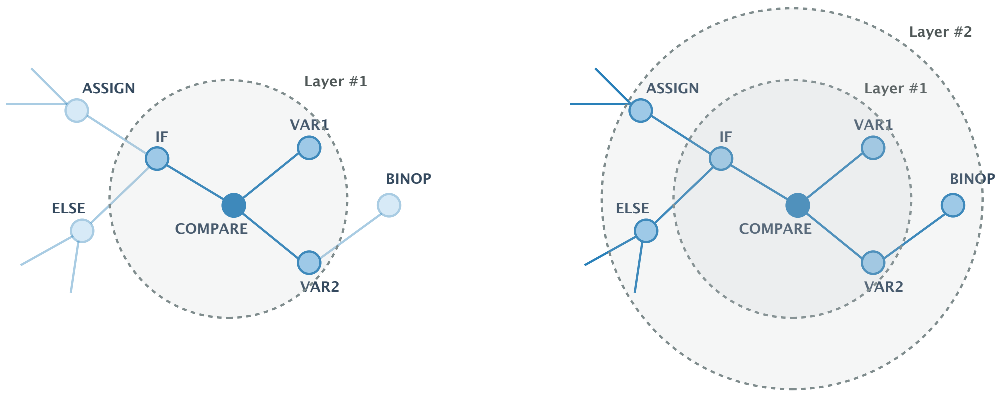


Figure 2.7 – Progressively aggregating from local structures.

Results

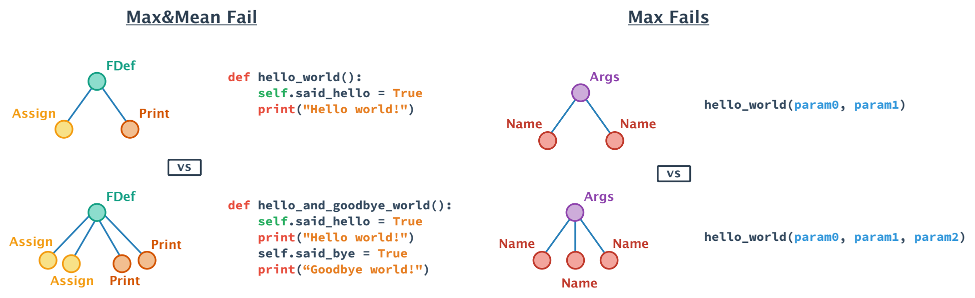
We first ran an experiment in which we learned token embeddings in an unsupervised fashion on large code graphs. In a downstream task, we attempted to predict missing subtokens from variable names through a GRU cell that predicted sub-tokens sequentially. While the contextual embeddings produced some improvements on the purely sequential model, it did so under conditions too restrictive to be useful in latter applications. For this reason, we decide to move forward with the methodology detailed in Chapter 3, not without isolating some of the key limitations of the initial method.

Shortcomings

One of GRAPHSAGE's main strengths turned out to be its downfall in our application. As opposed to previous matrix factorization approaches, GRAPHSAGE leverages node features in order to learn the embedding function. In our experiments, we found that this feature information was a large factor in obtaining a baseline result on the several tasks of learning on graphs. The addition of structure does provide a clear boost in performance, but the feature information is usually already extremely rich. In the case of code however, it is difficult to find rich node-level feature information. This seems to hinder GRAPHSAGE's ability to produce meaningful embeddings, despite successful applications of this framework with pure structural information as features (*e.g.* node degrees).

Another strength of the GRAPHSAGE framework is that it produces embeddings z_v for every node v in the graph, thanks to the fact that it learns aggregators rather than optimizing for the embeddings directly. This means it is possible to compute embeddings for unseen nodes in unseen graphs, in a sense predicting "out-of-vocabulary" to paraphrase NLP terminology. In our experiments however, the embeddings failed to capture enough context-related similarities due to huge discrepencies in the neighborhoods.

In [Xu et al., 2019], the discriminative power of different aggregators is discussed. They prove that if the aggregation function is injective, GNNs can fully discriminate the rooted sub-tree structures of the computation graph. In consequence, the authors highlight some interesting failure cases for common aggregation functions like MEAN or MAX. These can be relevant to characterize another shortcoming of GRAPHSAGE for our approach, as illustrated in Fig 2.8, which shows cases where the model should be able to produce different embeddings but doesn't.



```
def hello_world():
    self.said_hello = True
    print("Hello world!")
```

```
def hello_and_goodbye_world():
    self.said_hello = True
    print("Hello world!")
    self.said_bye = True
    print(“Goodbye world!")
```

```
hello_world(param0, param1)
```

```
hello_world(param0, param1, param2)
```

Figure 2.8 – Example Failure Modes for common Aggregation schemes

On the other side of the spectrum, Graph Convolutional Netowrks (GCNs) usually maintain an embedding look-up table [Kipf and Welling, 2016], which is updated with the new representations at each iteration, averaged across all instances of a given class. This setting is similar to a co-occurrence model, the WORD2VEC-like embedding process, which has been shown to be unable to capture a diversity of contexts. The usage subtleties (*e.g.* polysemy) are averaged out to obtain a representation vector. The distributional hypothesis did not seem to hold strongly enough to guarantee disentangled representations of the AST tokens when averaged. Finally, the limited vocabulary provided by the AST means that a lot of different meanings would be averaged together, also leading to uninformative representations. This limited our experiments with this alternative formulation, despite the fact that GCNs are better suited to the capture of purely structural information than GRAPHSAGE.

The graphs generated by the AST connect re-occurring "identifiable" nodes, unlike social or content graphs for example, where each instance in the graph is a unique user or product, whose features differentiate them but whose "nature" is the same. The observation of multiple occurrences of the same instance in different contexts is a hallmark of NLP methods, where elements of a vocabulary come together in multiple instances to form a corpus. This motivated a more direct inspiration from NLP advances than a persistence to use graph-based methods.

Finally, we also identified some weaknesses in handling highly imbalanced node degree distributions. In GRAPHSAGE, a fixed number of nodes are sampled from the neighborhood. If not properly tuned, this number of sampled nodes can hinder the ability to properly represent the actual node distribution in the vicinity of a given node. If a node has a high node degree, then several of its neighbours will be missing, at random. This is usually countered by sampling multiple times to cover the entire neighborhood. On the other hand, if the node has few neighbours, their importance can be inflated, occurring multiple times in a batch. This property is enviable from a random-walk perspective, as if there are few nodes they are often used by the messages passed through the GNN. However they can also be an issue in trying to uniformly sample from the node distribution.

While we still provide the code[3] to run and generate initial results using the framework presented here, these insights pushed a reformulation of the learning architecture, better suited to some of the specificities of source code. We discuss this second, more successful, approach in Chapter 3.

[3]https://github.com/dtsbourg/codesage

3 Towards deep structured representations of source code

3.1 Generalized Language Models

3.1.1 Attention is all you need

The domination of recurrent or convolution architectures in natural language tasks, and in sequence processing tasks in general, is threatened by a novel type of architecture, the TRANSFORMER, proposed by [Vaswani et al., 2017]. Only using feed-forward and attention-based [Bahdanau et al., 2015] blocks operating over a fixed sized context, this architecture was able to set new state of the art results on existing benchmarks. This caused a monumental shift in the NLP community, which hurried to adapt this attentive architecture in more complex ways.

In recurrent architectures, dealing with context requires sequentially stepping through the entire necessary context for each word, which makes the process extremely inefficient and difficult to train due to inherent optimization issues [Hochreiter et al., 2001]. In contrast, the TRANSFORMER performs a constant number of steps to aggregate the context. It is able to propagate information from multiple sources in parallel thanks to the attention process, which learns to model the relationship between the current word and every other word in the context. Firstly, this makes the model very efficient to optimize, and given its feed-forward structure it does not suffer form the same optimization quirks as RNNs. Secondly, the ability to explicitly refer to other parts of the context can be extremely useful, for example in translation where words can be aligned non-linearly between source and destination language. Finally, the attentive process can model multiple concurrent relationships through the use of multiple attention heads, which each learn different links.[1]

The current state of the art architectures are directly built upon TRANSFORMER blocks, with each variant proposing specific ways of composing different these elements. The General

[1]For a more detailed introduction the TRANSFORMER model, we please refer the reader to the excellent "Annotated Transformer" notebook provided by Alexander Rush from Harvard's NLP group: http://nlp.seas.harvard.edu/2018/04/03/attention.html.

Purpose Transformer (GPT) [Radford et al., 2018] introduced better ways to train in a semi-supervised fashion while promoting multi-task abilities. These capabilities are now at the center of a race between Google's BERT [Devlin et al., 2018], Microsoft's MT-DNN [Liu et al., 2019] and recently Baidu's ALICE. While none have surpassed human scores on the GLUE benchmark [Wang et al., 2018a], they outperform all existing sequence modelling architectures by a large margin.

Delegating the the information flow in the model to the attentive process is one of the key contributions of the TRANSFORMER. While it retains the popular encoder-decoder setting, the model leverages attention in multiple stages of the architecture. First, the decoder can attend the output of the encoder, capturing context from the entire sequence similarly to SEQ2SEQ models [Wu et al., 2016].

In the encoder itself, self-attention layers propagate information from the previous layer, where, again, each position can attend to all other positions in the previous layer. Finally, the decoder uses the attention as mask to hide context to the right of the token that is currently being decoded.

Specifically, the Transformer architecture uses a *scaled dot-product attention* mechanism, a flavor of soft attention. [2] It is more efficient when dealing with large sequences of input, compared to the traditional feed-forward/softmax attention mechanism [Bahdanau et al., 2015]. It also adds the scaling factor to better deal with large inputs, compared to basic dot-product attention [Vaswani et al., 2017].

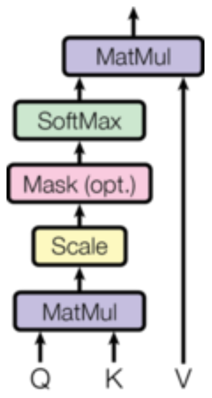


Figure 3.1 – Self-Attention in the Transformer ([Vaswani et al., 2017])

In this setting, attention is described as a mapping between a query Q and a set of key-value pairs (K, V). The output is then the weighted sum of the input, where the weights are computed as some measure of similarity, here a scaled dot-product, between the query and the corresponding key.

Essentially, (Q, K, V) are all projections of the embeddings $\mathbf{h}_i$ for token i, and the attention

[2]As opposed to hard-attention, which selects a discrete subset of the input instead of learning weights.

mechanism tries to encode the similarities of these transformations to learn relationships between the different elements of the input. These relationships can be captured in several different spaces thanks to the use of K parallel attention heads. The output of each head is then combined to give the final, contextualized representation $\mathbf{z}_i$ of a given token i.

3.1.2 The Transformer: a graph neural network perspective

In [Duvenaud et al., 2015], a general formulation of GNNs was proposed, formulating GNNs in a message-passing form, where “messages” containing information about neighbours are passed along edges which communicate this information to the relevant node, which then decides how to update its own internal representation based on the messages it receives.

Formally, let $\mathbf{h}_i^{(l)}$ be the representation of node i in layer l of the GNN. The message m_{ij} sent between two neighboring nodes i, j, where the neighborhood definition is left open as simply a relational component r_{ij}, is defined by Eq. 3.1. The inbound messages to a node i are then composed by an aggregator in Eq. 3.2. The node representation $\mathbf{h}_i$ is then updated through Eq. 3.3, using both its previous representation and this aggregated message from Eq. 3.2.

The full per-layer update of a GNN model can hence be performed through three key computations:

$$m_{ij}^{(l+1)} = \textsc{Msg}\left(\mathbf{h}_i^{(l)}, \mathbf{h}_j^{(l)}, r_{ij}\right) \tag{3.1}$$

$$M_i^{(l+1)} = \textsc{Agg}\left(\left\{ m_{ij}^{(l+1)} \mid v_j \in \mathcal{N}_{v_i} \right\}\right) \tag{3.2}$$

$$\mathbf{h}_i^{(l+1)} = \textsc{Update}\left(M_i^{(l+1)}, \mathbf{h}_i^{(l)}\right) \tag{3.3}$$

The final node representations are given by $\mathbf{z}_i = g(\mathbf{h}_i^{(L)})$ after L updates, with g an arbitrary post-processing function, often identity or a simple linear transformation if the representations are to be generated at a node-level, or some aggregation step for graph-level embeddings.

If we consider the TRANSFORMER architecture, the attentive process through which information is propagated from word to word can be seen as a form of GNN. Indeed, self-attention computes weighted messages between words in a sequence, following Eq. 3.1. In the case of [Radford et al., 2018], these connections r_{ij} are directional, with the context being passed on from previous words, given that they mainly leverage the decoder part of the TRANSFORMER, which is sequential. However, [Devlin et al., 2018] proposed a bi-directional formulation, hence the name Bidirectional Encoder Representations from Transformers (BERT). In this case, each word is “visible” to every other, emulating a fully-connected graph on which to pass messages. While the attentive propagation mechanism is order-agnostic, natural language sentences do naturally encode ordering. To solve this, the authors propose the addition of a positional encoding, which is summed to the input representation and serves as a pseudo-coordinate system for the model to latch onto.

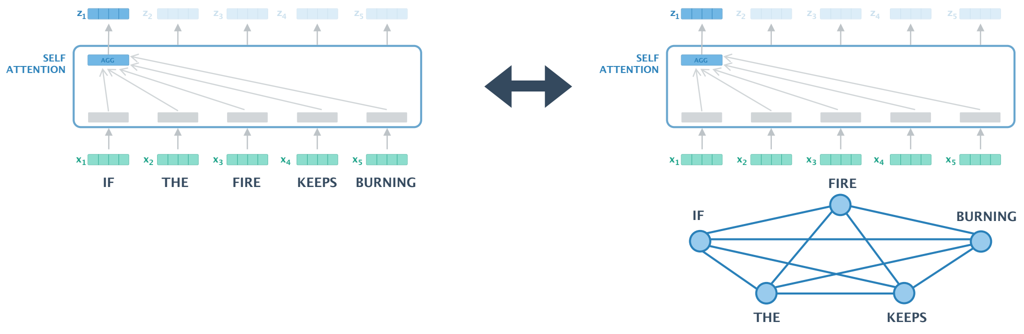


Figure 3.2 – Self-attention in the TRANSFORMER as a fully-connected aggregation scheme.

The aggregation (Eq. 3.2) and update steps (Eq. 3.3) are then standard update steps, where again the graph is fully connection so the neighborhood $\mathcal{N}_{v_i}$ represents the entire context graph.

3.2 MARGARET: Masked Attention foR Graph RepresentAtions with the Encoder from Transformers

3.2.1 Motivation

Recognizing the kindred spirit between the TRANSFORMER's architecture and the neighbourhood propagation and aggregation process of a GNN, we propose a generalization of the TRANSFORMER block that allows learning on arbitrarily structured structured input, namely graphs. These models' are able to effectively compose rich contextual representations at both a local and a global level, producing highly contextualized and semantically-rich representations. The proposed formulation is flexible enough to retain its ability to learn on sequences by defining the adequate input structure, but can now be leveraged to operate on feature-rich nodes with well-defined relational components.

The main setting of this thesis, learning representations of source code, fits in between both concerns. It provides a well-defined structured representation to limit the applicability of frequentist approaches but not enough node features to be treated as a classical graph-based learning problem. This hybrid formulation provides the perfect test bed to showcase the flexibility of our formulation.

As we have detailed throughout this work, source code provides readily available syntactic information that we would like to leverage. Incorporating such priors has multiple benefits. It allows the model to directly leverage information that would otherwise be hidden. Provided that the representation is correctly chosen, this should facilitate the learning process, or at

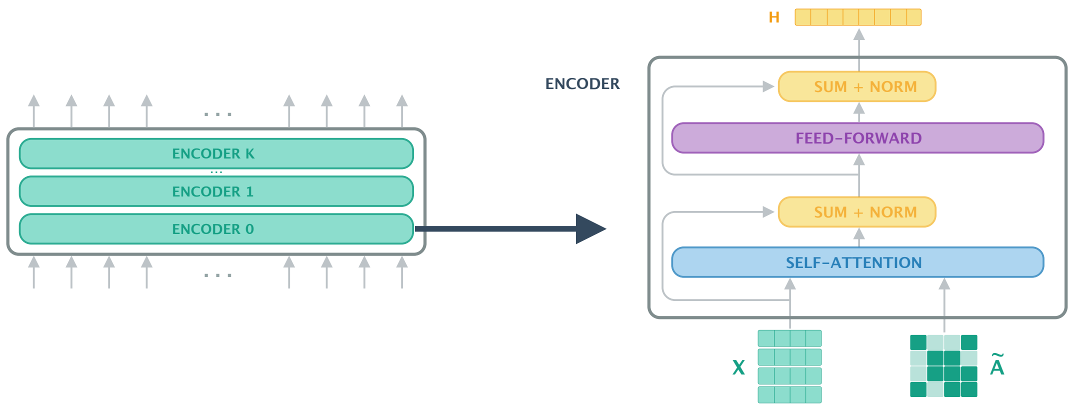


Figure 3.3 – The proposed encoder architecture.

least help make it faster as the model doesn't have to jointly learn syntax and semantics. As we detail in Section 3.2.4, this is a promising direction for the integration of pre-existing knowledge about the task at hand, in line with new synergies between data-driven, parameterized approaches and manually-designed models, constrained by heuristic models of reality.

Unifying the TRANSFORMER model with the general GNN formulation extends its operation to graph processing at large. Natural language processing generally relies on the co-occurrence of a restricted set of tokens, *i.e.* a vocabulary, in many different constructions. Exposing the underlying structure, where available, opens the application of these successful methods to capture re-occurring patterns on structured inputs. Graphs defined by a restricted set of nodes which interact multiple times through a given topology can be mined for functional patterns. Examples of this setting include anything from molecular graphs in chemistry or biology to content graphs, with structured representations of source-code representing a perfect intersection between free-form co-occurrence inputs and structured representations.

3.2.2 Formulation

Let $\mathscr{G}_{AST} = (V, E)$ denote a graph representing a code snippet C_s, where $V = \{v_0, \dots, v_N\}$ the set of AST nodes, N being the number of nodes, and $E = \{(v_i, v_j)\}$ the set of AST edges connecting them. This graph can be represented as its adjacency matrix $A \in \mathbb{R}^{N \times N}$. Here we assumed A to be unweighted such that $A_{ij} = \delta_{r_{ij}}$, which is 1 if (v_i, v_j) share an edge and 0 otherwise. A snippet is loosely defined to be a subset of a program, containing at most N tokens, valid in the sense that is it syntactically correct. This includes, among others, function definitions, a line of code, a logical block, ...

The goal is to find a vector representation $\mathbf{z}_i$ for each node $v \in V$. Here, we want to constrain the model to leverage the provided structural information. Instead of a fully-connected graph, we start by restricting messaging pathways to syntactically existing edges, for example those provided by the AST.

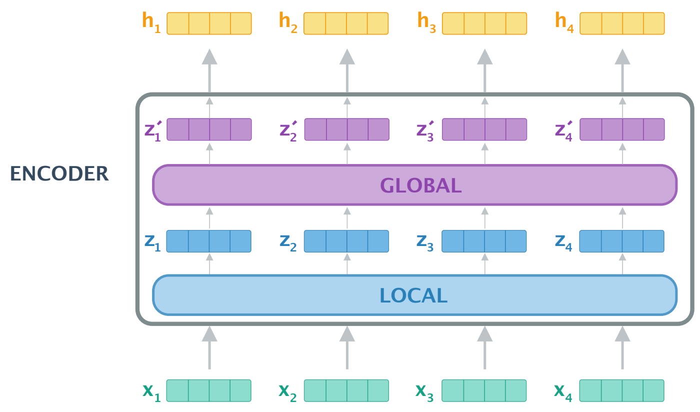


Figure 3.4 – Global and Local aggregation steps.

In this setting, the local information propagation scheme is unchanged: messages between nodes are computed on the edges r_{ij} that connect them. For each node, these messages are then aggregated from the local neighborhood to form a composite message. The choice of the exact aggregator is discussed in Section 3.2.3, but essentially any neighborhood aggregation scheme can be leveraged.

In the message-passing framework, the update step (Eq. 3.3) is also local: it incorporates messages only from its neighbours. To incorporate longer-ranging dependencies, the model must be updated sequentially. At layer k, the aggregated message in Eq. 3.2 then contains information from the k-hop neighborhood of node v_i. However this information is propagated only through the neighbours $v_j \in \mathcal{N}_{v_i}$, which themselves have been updated with the information from their own k-hop neighbourhoods. This makes it difficult to model long-ranging dependencies, as information is diluted through the layers.

In the TRANSFORMER model however, the update step is global. This is a consequence of the fact that for a fully-connected graph, local and global updates are equivalent. In restricting the message-passing pathways, we restrict the model's receptive field. The first option is to instead provide local updates to the node representations based on the aggregated representation from Eq. 3.2. However this reduces to the same GRAPHSAGE/GCN-based formulation presented in Section 2.4, which we have shown to present substantial shortcomings. One notable limitation is the inability to effectively incorporate long-ranging dependencies.

By sequentially aggregating information, we force the model to propagate information from k layers away through k aggregation steps. With the combinatorial explosion of the number of nodes, this can drown information from more than a few hops away, restricting the ability for the fully-local models to leverage far-removed node's features (similar to gradient vanishing in deep recurrent models) and their contextual information (because of excessive averaging at each hop). This entire model is also based on the assumption that nodes close together in the graph should be informative for each other, but this assumption is limiting, notably in the case of source code where dependencies can be long-ranging.

To counter this, we propose to preserve the global update step, making this model conceptually closer to the non-local neural network formulation [Wang et al., 2018b]. One fundamental difference lies in the fact that we learn the global aggregation layer instead of using a non-parametric node similarity function.

This hybrid formulation allows not only to compose local information but also to integrate global information, where the contextual representations of each node is preserved: the local structures are preserved, and the global features' propagation can easily be learned. Both are then composed to obtain deep contextualized representations from structured inputs. This is conceptually related to work like Sentence-state LSTMs, where parallel global and local updates are performed to compute sentence embeddings [Zhang et al., 2018].

Algorithm 2: MARGARET encoder algorithm.

```
Input  : Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth $K$; weight matrices
         $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity $\sigma$; differentiable aggregator functions
         $\textsc{aggregate}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \to 2^{\mathcal{V}}$; INIT the embedding
         initializer; OUT the final output layer; FFN is a dense Feed-Forward NN.
Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$
1  $\mathbf{h}_v^0 \leftarrow \textsc{Init}(\mathbf{x}_v), \forall v \in \mathcal{V}$
2  for $k = 1...K$ do
       ▷ Local update
3      for $v \in V$ do
4          $\mathbf{h}^k_{\mathcal{N}(v)} \leftarrow \textsc{Aggregate}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$
5          $\mathbf{h}_v^k \leftarrow \textsc{LayerNorm}\left(\mathbf{h}_v^{k-1} + \sigma\left(W^k \cdot \mathbf{h}^k_{\mathcal{N}(v)}\right)\right)$
6      end
       ▷ Global update
7      $H_V^k \leftarrow \textsc{Concat}(\mathbf{h}_v^k), \forall v \in \mathcal{V}$
8      $H_V^k \leftarrow \textsc{LayerNorm}(H_V^k + \text{FFN}(H_V^k))$
9  end
10 $\mathbf{z}_v \leftarrow \textsc{Out}(\mathbf{h}_v), \forall v \in \mathcal{V}$
```

Together, these steps form an encoder, of which multiple layers can be sequentially stacked to obtain deeper and deeper representations. Note that the modularity of this architecture also allows to experiment with alternating global and local update steps. For instance, several local blocks could be successively applied in order to grow the receptive field large enough, before being passed to a global aggregation step. This reduces closer to a graph classification model for a GNN, though more subtleties can be added. The presence of skip-connections for example is uncommon in GCNs, though it has been hypothesized to produce better results [Velickovic et al., 2018].

In the current formulation, the INIT function is a random initializer, but if rich feature information is available about the nodes, a simple look-up table can be used to produce the initial embedding vector.

3.2.3 On the choice of a local aggregation scheme

Graph Convolution (GCN)-based aggregator

In a Graph Convolutional Network, the representation of a node u at layer k is obtained from the normalized sum of the features of all its neighbours:

$$\textsc{Aggregate}_k(u) = \sigma\left(\sum_{v \in \mathcal{N}(u)} \frac{1}{c_{uv}} W^k \cdot \mathbf{h}_u^k\right) \tag{3.4}$$

where c_{ij} is the normalization constant, usually taken as $c_{uv} = \sqrt{|\mathcal{N}(u)|}\sqrt{|\mathcal{N}(v)|}$ for GCNs [Kipf and Welling, 2016], or simply $c_{uv} = |\mathcal{N}(u)|$ in a framework like GraphSAGE [Hamilton et al., 2017]. σ is the UPDATE operation from Eq. 3.3, usually a RELU activation function. Finally, W^k is a learned weight matrix which linearly composes the features from neighbours.

Graph Attention (GAT)-based aggregator

As highlighted in follow-up studies [Luong et al., 2015], it appears that one of the most powerful mechanisms introduced by the TRANSFORMER is its ability to selectively compose information. Where a GCN-based approach would indiscriminately average the neighbourhoods information, attention-based aggregation schemes weight each neighbour differently. This enable fine-grained contextual information to propagate and influence the final representation.

Attention is a mechanism that has been proposed for precisely this purpose [Bahdanau et al., 2015]. It finds an extension to graphs through Graph Attention Networks (GAT), which only preserve attention weights on existing edges rather than on the entire set of nodes [Velickovic et al., 2018]. Instead of a constant normalization weight c_{uv}, GATs use a small neural network ϕ to predict which neighbours are the most influential. The softmaxed output of this prediction is used as a weight for the aggregation. The local update step then takes the following form, at layer k for node u:

$$\textsc{Aggregate}_k(u) = \sigma\left(\sum_{v \in \mathcal{N}(u)} \alpha_{uv}^k \cdot W^k \cdot \mathbf{h}_u^k\right) \tag{3.5}$$

where the weights $\alpha_{uv}^{(l)}$ are computed from the output of the predictor ϕ:

$$\alpha_{uv}^{k} = \textsc{Softmax}\left(\phi(u,v)\right) \tag{3.6}$$

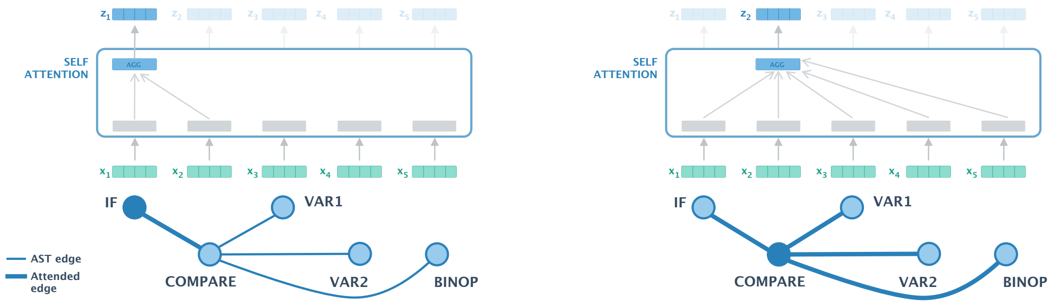


Figure 3.5 – Neighbourhood aggregation scheme.

Masked Attention-based aggregator

In contrast with the additive attention of GATs, the original Transformer model uses a dot-product architecture. Both are still soft-attention mechanisms, with the latter requiring less parameters. We modify this attention scheme by masking the activation vectors with the normalized adjacency matrix. In essence, this limits attention to pass messages along the pre-defined structure that we have provided, for example through the AST in the case of source code.

Using the notation from Section 3.1, let (q,k,v) be the query, key, value decomposition of the input to the attention layer at depth k, for node u. The aggregation function is then defined as:

$$\textsc{Aggregate}_k(u) = \sum_{v \in \mathcal{N}(u)}^{N} \textsc{Softmax}\left(\frac{\mathbf{q}_u^k \cdot \mathbf{k}_v^k}{\sqrt{d_k}}\right) \cdot \mathbf{v}_u^k \tag{3.7}$$

or, in matrix form for all nodes:

$$Z = \textsc{Softmax}\left(\frac{\tilde{A} \odot (Q \cdot K)}{\sqrt{d_k}}\right) \cdot V \tag{3.8}$$

,
where $\tilde{A} = A + I$, which adds self-loops in order to enable self-attention.

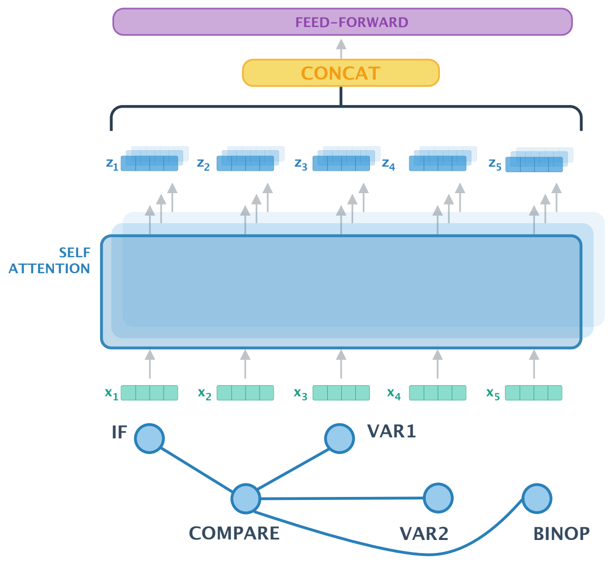


Figure 3.6 – Multi-headed attention aggregation.

Semantic aggregation schemes

The definition of an aggregation function is quite flexible. As the neighbourhood of a given node u has no natural ordering, the aggregation should operate over an unordered set of values. Beyond that, the desired properties are up to the designer. While some aggregators provide more or less expressive power, as shown in [Xu et al., 2019], other properties could be considered in the choice or design of novel aggregation schemes.

The desiderata and options for more specific aggregators undertaken in Section 2.4 still holds in the current scheme, given the potential equivalence in the local neighborhood aggregation scheme. The added bonus here comes from the flexibility of the architecture, which can reduce to the simple sequence learning problem that has shown great results so far. It can also leverage additional information, such as semantic annotations or regular parse trees for text, as discussed in Section 3.2.4.

In the end, we decide to stick with the masked attention formulation, as it was the most straightforward extension to prove the conceptual feasibility of our idea. In Chapter 4 we show that this choice leads to quality results, which can surely be compared to the other aggregation schemes presented here in future work.

3.2.4 Syntax-aware language modelling

The advent of large-scale attentive models signals that in general, syntax can be learned from the input signal. By attending to syntactically relevant structures, the network would concurrently learn the syntax of the language it was modeling. Attention has even been observed to exhibit distributional properties in that the different heads can concurrently learn to attend to different types of semantic and syntactic structures. Even without attention, [Gulordava et al., 2018] showed that RNN-based LMs are able to capture long-distance agreements, exhibiting deeper grammatical competence than previously thought. Several works have demonstrated that deep RNNs capture some form of syntactic hierarchy, *e.g.* [Blevins et al., 2018, Gulordava et al., 2018]. More recently, the advent of TRANSFORMER-based architectures has shown even more efficient encoding of sentence structures [Tenney et al., 2019] and of other properties like subject-verb agreement [Goldberg, 2019]. In general, better probes into the models and the produced embeddings are being developed to understand exactly what is captured in the learning process [Conneau et al., 2018].

Despite a definite improvement in models' ability to capture semantically and syntactically relevant structures, there are still some glaring shortcomings. [Belinkov et al., 2017] found that the use of attention reduces the decoder's ability to learn the syntax of the target language, while syntactically relevant structure is captured in the source language, as shown by [Shi et al., 2016]. While they highlight the fact that Recurrent Language Models can capture number agreement, [Linzen et al., 2016] also note that this intuition fails when structural and sequential information are conflicting. The way in which these models fail has also been shown to be very different than that of humans' linguistic understanding patterns [Linzen and Leonard, 2018], despite some similarities in reaction to some grammatical stimuli, such as noun-phrase re-ordering [Futrell and Levy, 2019].

To address these issues, the routine arsenal of machine learning methodology has been deployed. More complex, deeper models have been proposed: [Radford et al., 2018] has around 110M parameters, BERT's largest version has around 350M [Devlin et al., 2018], with all these being blown out of the water by GPT-2's whopping 1.5B parameters [Radford et al., 2019]. They are also trained on more challenging tasks, which require increasing levels of semantic and syntactic understanding [Wang et al., 2018a, McCann et al., 2018]. Finally, the importance of the quality of the input data is primordial, with large, clean datasets often offering substantial improvements without little architectural changes. [Radford et al., 2019]

As an alternative, recent work has started to argue that adding explicit syntactic knowledge can benefit the learning process. The ability to ensure syntactic correctness was pioneered in translation by [Yamada and Knight, 2001], the field through which this resurgence began [Eriguchi et al., 2016, Aharoni and Goldberg, 2017, Li et al., 2017, Eriguchi et al., 2017]. In some ways related to our approach, [Bastings et al., 2017] propose leveraging a GCN as a pooling operation on dependency trees, generating more contextual embeddings for translation applications. GCNs were also used over semantic annotations like predicate-argument structures by [Marcheggiani et al., 2018] or dependency parses [Vashishth et al., 2018]. In terms of formal grammars, the first foray into logical reasoning with GNN was offered by [Wang et al., 2017], which attacked the problem of premise selection in theorem proving, leveraging graph-based representations of mathematical formulas.

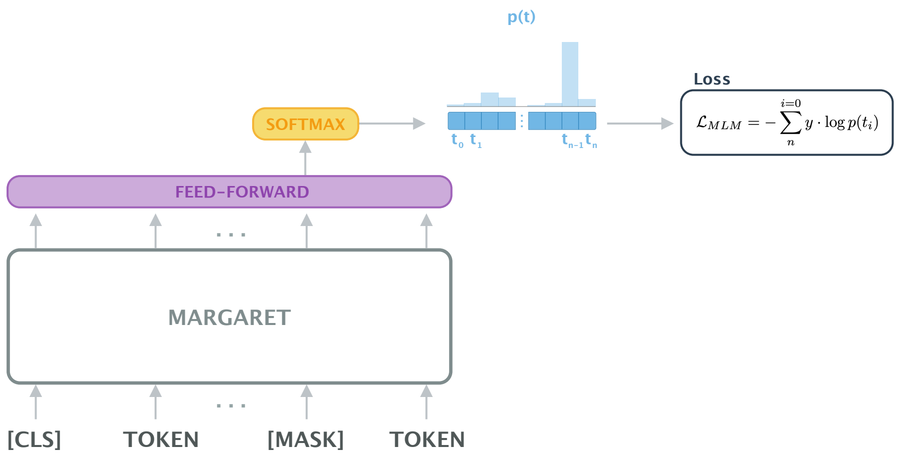

p(t)
Loss
$\mathcal{L}_{MLM} = -\sum_{n}^{i=0} y \cdot \log p(t_i)$
SOFTMAX
t_0 t_1 t_{n-1} t_n
FEED-FORWARD
MARGARET
[CLS]
TOKEN
[MASK]
TOKEN

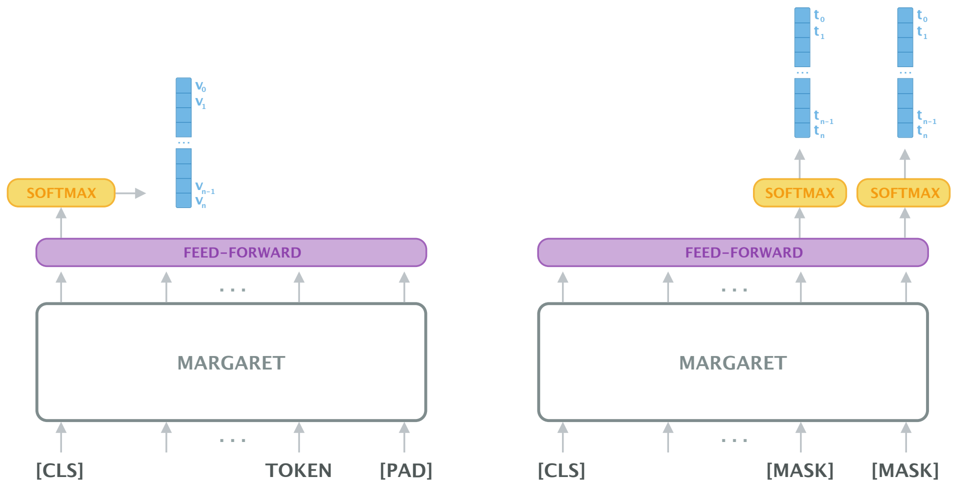
Figure 3.7 – The Masked Language Model task.

The specific structure of code and the availability of syntactic information through structured parses like the AST make it a prime domain for the addition of apriori knowledge into already powerful, state of the art NLP methods. This work shows that augmenting these architectures with structure provides a strong learning base for the model to capture semantic and syntactic constructs, embedding strong inductive biases into the model.

3.3 The masked language model objective

In Section 2.4, we describe the unsupervised loss that GraphSAGE uses to compute similar embeddings for similar nodes. Here, we differ from this approach, taking from a method *en vogue* in the NLP community, by learning embeddings through a semi-supervised loss. Previous architectures, and their losses, were inherently directional. Traditionally, semi-supervised prediction tasks were hence run sequentially: the goal is to predict the next word given the past context, and repeat for each word across the sequence. This can be seen as a remnant on the Markov Chain assumption for LMs (see Section 2.1). It is pervasive even in latter architectures like [Peters et al., 2018a, Radford et al., 2018], which implement left-to-right prediction tasks.

Instead, [Devlin et al., 2018] proposed a Masked Language Model (MLM) task, where a bidirectional prediction task is obtained by masking some percentage of tokens at random, then asking the model to predict those masked tokens. This task is often referred to a *Cloze task* [Taylor, 1953], originally designed to estimate the “readability” of a piece of text by randomly deleting one or several “units” (here a word) and measuring the difficulty of filling in the blank. This objective is similar to denoising auto-encoders, except for the fact that only the masked part of the input is to be reconstructed.



(a) Graph-classification tasks.

(b) Node classification tasks.

Figure 3.8 – Multi-task abilities.

We extend this line of thought to our setting on graphs by randomly masking some percentage of nodes. We do not ask the model to reconstruct the local connectivity patterns, only the node label itself. Nevertheless, it could be extended to a full reconstruction, which might further aid the model to capture structure (see Section 3.6). In a sense, the MLM task as defined on graphs is a node classification task, a well defined learning objective for GNNs. In this instance, the label is supposed to be generated from the training nodes' labels, information which is available by definition since we are working with graphs with re-occurring nodes. Given that training data exists in abundance, it is an attractive option for training our model.

Instead of feeding the model with a sentence, we define an analog by using code subgraphs that are sampled from the ASTs generated from our source code corpus. This abstraction can be seen as a snippet of code, a small block, usually spanning 1-5 lines, which we found to be a reasonable granularity level to provide the model with and obtain meaningful representations. Concretely, the input to the model is then a list of token identifiers, hashed according to the vocabulary drawn from the tokens extracted from the training set, and the adjacency matrix representing its structure, as illustrated in Fig. 3.3.

3.4 Supervised learning

The proposed architecture is also suitable for use as a supervised learner. In the previous section, we presented a semi-supervised formulation, in which node labels are retrieved from their identity. However, other node classification tasks are natively supported by the model, by simple extension of the semi-supervised setting to a fully supervised one. In case the labels change dimensionality, the feed-forward output layer can be replaced to adequately fit the new task, and the loss can easily adapted, as illustrated in Fig. 3.8b, by changing the OUT function used in Alg. 2.

The model also provides an extension to an equivalent of graph classification. Inspired by BERT, we append a special classification token dubbed `[CLS]`. It is fully connected in the context of the subgraph, operating as a graph pooling operation in parallel with the rest of the aggregation steps described in the model architecture. In the output layer, this pooled representation is passed to a feed-forward classification layer, which then attributes the label. Again, the only additional step is the definition of an appropriately-sized output layer, which matches the number of labels in the multi-class setting, before the definition of an adequate loss, usually a cross-entropy objective.

Note that another way to formulate this problem is to consider the input graph as the sampled computation graph, with the graph label being the label that we want to compute for the anchor node from which this computation graph is sampled.

The original semi-supervised task was defined in such a way that data was abundant. However in the purely supervised setting, annotations are either expensive to obtain or much rarer, making it difficult to accurately learn a predictor on this restricted sample. To counter this, we introduce the notion of pre-training, where the model is first trained in a domain where data is abundant. This same model is then fine-tuned on a second related task for which data is much scarcer, with the learned weights serving as the initial state of a supervised model.

3.5 Pre-trained language models and transfer learning

Traditionally, NLP applications are constructed through supervised learning on a task-specific objective. A single model is trained and evaluated on a single task, for use in a single domain. This inherently limits the scope of a given model, and does not reflect the real-world applications of these systems which should ideally be multi-lingual, multi-task and multi-domain. If not by nature, these models should at least be adaptable to these generality constraints.

The purely supervised formulation can sometimes hinder the ability for a model to actually capture semantics of a task, instead optimizing for the controlled classification objective. This shallowness in understanding has been shown for example by demonstrating that paraphrasing a successfully classified example can lead to failure in prediction [Iyyer et al., 2018]. Supervised methods have also shown to be brittle, being very sensitive to adversarial examples [Jia and Liang, 2017] and noise [Belinkov and Bisk, 2018].

Despite downstream applications being strongly supervised, NLP has a long tradition of transferring knowledge from one context to another. Many techniques revolved around producing syntactically and semantically meaningful features that could the be leveraged for downstream applications. The most recent and popular variant of this would of course be the pre-trained word-embeddings produced by WORD2VEC [Mikolov et al., 2013], but even earlier work like Latent Semantic Analysis (LSA) [Deerwester et al., 1990] or Brown Clusters [Brown et al., 1993] can also be seen as forms of transfer learning. Skip-thought vectors [Kiros et al., 2015] even extend this idea to sentence-level features, which are then used in a downstream linear model on 8 different tasks.

However, recent advances in semi-supervised representation learning of LMs, such as BERT [Devlin et al., 2018] or GPT-2 [Radford et al., 2019], and the study of their latent properties [Belinkov et al., 2017, Zhang and Bowman, 2018], have enabled a new wave of transfer learning. These advances concern both *transductive* (training and running on the same task but with data labeled only for the source domain) and *inductive* (transferring knowledge between tasks for which labeled data is available in the target domain). We focus here on the latter as it is most relevant to our applications, but plenty of great work deserves to be highlighted in the former, with advances in domain adaptation and cross-lingual learning providing powerful tools to democratize advances in NLP, for example by enabling the use of the tools on rare languages. [Artetxe et al., 2018, Lample et al., 2018, Schwenk and Douze, 2017, Conneau et al., 2017]

Early work in multi-task learning showed that the ability for a model to transfer its knowledge depends on the hypothesis space in which it is learned: instead of learning to solve the specific task, a learner exists within an environment of related learning tasks and converges towards one of them, depending on the optimization under the training data's constraints [Caruana, 1997, Baxter, 2000]. Later, [Ando and Zhang, 2005] proposed that a semi-supervised learning process fit the gamut for transferring knowledge across tasks, offering a general enough hypothesis space. This idea was then applied specifically to language models by [Dai and Le, 2015], cementing its place as a popular tool with ELMo [Peters et al., 2018a]. The main insight of these methods is that the model used to learn features in an unsupervised fashion can be used a seed for the supervised model. In other words, the model is "pre-trained" to capture a general understanding of the domain, before becoming specialized for a downstream task [Erhan et al., 2010].

This learning mode is directly inspired by advances in Computer Vision where large architectures (*e.g.* AlexNet [Krizhevsky et al., 2012]) were pre-trained on large processed corpora of images like ImageNet [Deng et al., 2009], only to be fine-tuned for the final application. In the intermediate layers, filters of increasing complexity had already been learned, from edge-detectors to basic shapes. By training a second time but with a small learning rate, the model can make subtle updates mostly localized on the final output layer while still leveraging the powerful filters it has learned a priori.

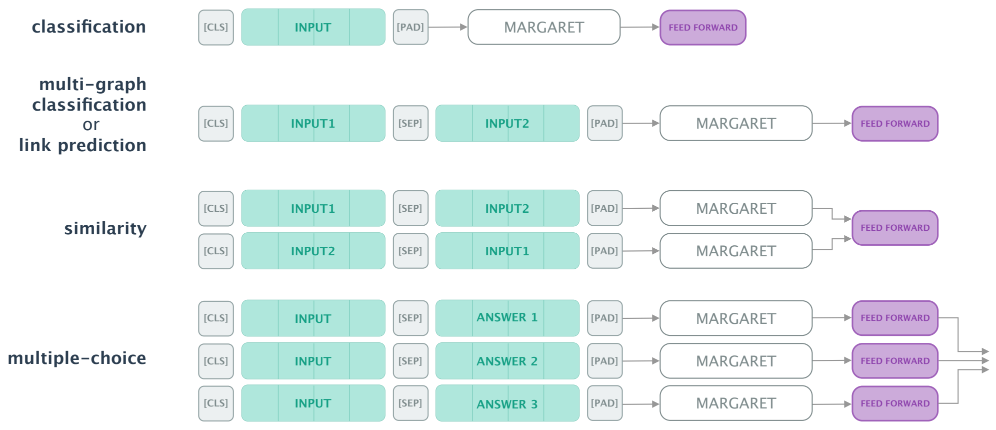


Figure 3.9 – Support for multiple tasks.

Until BERT [Devlin et al., 2018], it remained to see if this type of pre-training was possible for text. However with the advent of high-quality datasets like [Radford et al., 2019] and the powerful Transformer architecture [Vaswani et al., 2017], this "IMAGENET moment" [3] has arrived in NLP. Similar to powerful pre-trained CNN models, NLP has transitioned from learning edge-detectors to rich, high-level feature extractors. Conceptually, the field's relative success motivated our attempts are producing richer representations for structured inputs.

In the case of BERT, fine-tuning can be achieved by adding a simple linear classification layer, either after obtaining a pooled representation from the starting keyword token `[CLS]` or in a task-specific manner as shown in Figure 3.8a-3.8b. The model is then re-trained, with the parameters of BERT and the added linear layer being optimized jointly to minimize the classification loss (often a cross-entropy loss to maximize the likelihood of the desired label).

Unfortunately, this requires copying and fine-tuning the model for each separate task, which [Liu et al., 2019] solves by directly incorporating parallel task-specific output layers on top of the BERT architecture. In similar spirit, [Houlsby et al., 2019, Stickland and Murray, 2019] propose an adapter module, a small model with only a few parameters per task, reducing the computational cost of re-training the entire model. Note that additional improvements can be gained by better engineering the optimization of the fine-tuning task, as proposed by ULMFIT [Ruder and Howard, 2018].

3.6 Model extensions and future work

The versatility of the proposed model shows great promise for a variety of different extensions, some of which will be detailed in this section. These represent future research directions of interest or simply investigations into possible enhancements to the model that could not be

[3]https://thegradient.pub/nlp-imagenet/

investigated during the course of this work.

Variable length inputs.

One of the biggest limitations of the proposed model is the rigidity of the maximum size of the graph that can be manipulated. While this is common practice in other graph models, *e.g.* [Hamilton et al., 2017] restrict the number of sampled neighbours, we would like an alternative that can produce embeddings for arbitrarily sized graphs. To tackle this problem, two angles appear evident. The first leverages code's modular properties: smaller chunks are built up to form larger entities, themselves composing larger logical abstractions. This compositionality can be leveraged by the model, for example by recursively applying MARGARET with as input the embeddings produced by the previous layer, similar in spirit to [Dai et al., 2019]. The second approach would seek to choose a fixed base-layer of abstraction, but integrate global features into this representation. For example, some initial work was undertaken to embed the library structure of a package through hierarchical embeddings, proposed by [Nickel and Kiela, 2017]. The integration of information at different scales can be an issue, though inspiration can surely be taken from the way our proposed model handles this composition at a snippet-level.

Multi-graph setting.

In BERT [Devlin et al., 2018], a second pre-training task is introduced to complement the masked language model objective. In this setting, the model is asked to predict whether two sentences provided as input, spaced by a separator, are contiguous. This forces the model to reason at a longer range, maintaining global context across sentences. A similar learning artifact could be introduced here, we two subgraphs could be separated, with a classification task whose objective is to determine whether these two snippets are used in similar contexts. An clear extension would consist of a link-prediction task, such as guessing whether a second snippet is likely to follow the first input, as illustrated in Fig. 3.10.

In pre-training, it has been shown that multi-task learning provides great improvement when transferring to supervised settings, particularly when tasks are related in the hypothesis space [Caruana, 1997, Baxter, 2000]. Training the model on a more global task could enhance its generalization capabilities.

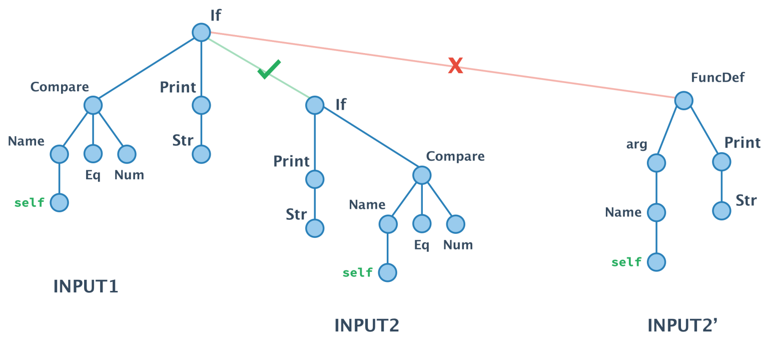


Figure 3.10 – Support for multiple graphs, as showcased in a link prediction task.

Multi-modal extension.

Since the presented model can natively support free-form text through a fully-connected input graph, we imagine an application where two or more segments are provided, in line with the multi-graph setting described above. However, both would be of different modalities. For example, `stackoverflow.com` questions are often structured as a bit of text along with a code snippet. An interesting prediction task would be to generate answers to a code-related question by dealing jointly with text and code.

Multi-lingual properties.

Here, we present results for a single language, which makes sense particularly when pre-training a model. However much work in linguistics has been done to map syntactically related languages, and even support multiple languages with one single model [Artetxe et al., 2018, Lample et al., 2018, Schwenk and Douze, 2017, Conneau et al., 2017]. The AST provides a relatively abstract formulation, whose syntactic elements could be aligned to other related languages. Additionally, we do not rely on language-specific semantic edges, meaning only syntactic structure is needed. This would open tasks like similarity to the multi-lingual setting, providing useful tools, for example to recommend from APIs across implementations and languages.

Code-specific tokenizers.

NLP has a large spectrum of pre-processing methods for its input tokens, many of which have been shown to provide decent improvements in accuracy by helping the model focus on relevant syntactic and grammatical structures, and by enhancing the model's ability to predict out-of-vocabulary elements by looking at sub-structures instead of entire words [Kudo, 2018]. Some example include [Radford et al., 2019] who leverage these insights through the use of Byte-Pair Encoding [Sennrich et al., 2016] or BERT, which pre-processes tokens by applying the WordPiece tokenizer [Wu et al., 2016] cutting words based on semantic properties (plurality, adjective modulation, tense, . . .) [4]. Given the specific patterns that programmers employ, we hypothesize that code-specific tokenizers could provide a similar performance boost. There often exist coding conventions that can be a way of specifying some particular property, from informative prefixes like `get, set, ...` which convey meaning to the use of an underscore to specify that a method is local to the class. The ability to model these specificities would pursue our desire to embed the model with a priori knowledge, guiding the learning process into regions that we know to be relevant.

[4] https://juditacs.github.io/2019/02/19/bert-tokenization-stats.html

4 Experiments

4.1 Datasets

In this section we detail the data collection, processing and generation process. All the processing code is made available through the `codegraph-fmt` tool. [1]

4.1.1 Collection

We focus our efforts on Python code. Despite being considered a more challenging task [Allamanis et al., 2018b, Alon et al., 2018] than other languages more routinely used like JAVA, JAVASCRIPT or C#, we decide to address PYTHON as it is now ubiquitous in many tasks and domains, including our own, where the libraries and model we provide here are implemented in PYTHON. It offers a large corpus of implementations across diverse topics and tasks: it is the third most popular language on `github.com` and just surpassed JAVA for the most discussed language on `stackoverflow.com`. [2] Here, we collect the source code of the top-10 trending PYTHON repositories on `github.com`, and collate the data into three different corpora representing three different scales:

- **CORPUS-SM** A single project. [3]
- **CORPUS-MID** A collection of three projects, related in topic. [4]
- **CORPUS-LG** The collection of top-10 most popular PYTHON projects on `github.com`.

We also collected a giant dataset consisting of 3 TB of raw source code data, scraped directly from `github.com`. The processing of this massive corpus requires a compute power that unfortunately was not available at the time of writing. However, the way the processing pipeline is setup, the choice of repositories is arbitrary, so the desired selection can simply

[1] `github.com/dtsbourg/codegraph-fmt`
[2] https://bit.ly/2N3x5sE
[3] In this case we selected `keras` for its intermediate size.
[4] We chose `keras`, `scikit-learn` and `pytorch`.

be created manually, cloned or added as a sub-module into the correct path specified by the documentation. The exact hashes that we used in the experiments are specified as the Hash ID in Table 4.1.

	LoC	# Snip.	# Tokens	# Unique Tok.	Avg. node deg.	Hash ID
`keras`	38,139	7,142	173,696	1,156	2.09 / 4.69	`3e6db0e`
`sk-learn`	192,663	35,228	776,365	3,581	2.07 / 4.61	`611254d`
`pytorch`	17,163	2,384	59,803	740	2.06 / 4.70	`f3a860b`
`ansible`	428,144	95,846	2,168,605	5,847	2.06 / 4.65	`0b579a0`
`requests`	5,036	699	11,508	452	2.06 / 4.72	`2820839`
`django`	121,188	22,892	337,444	3,413	2.05 / 4.71	`cf826c9`
`httpie`	3,919	612	8,886	421	2.06 / 4.65	`358342d`
`youtube-dl`	131,960	25,742	371,753	2,248	2.04 / 4.69	`794c1b6`
`flask`	7,750	804	13,086	490	2.05 / 4.64	`4f3dbb3`
`BERT`	5,928	1,967	17,805	480	2.06 / 4.60	`bee6030`
CORPUS-SM	65,225	7,142	173,696	1,146	2.09 / 4.69	
CORPUS-MID	247,965	44,754	1,009,864	3,823	2.07 / 4.67	
CORPUS-LG	951,890	193,316	3,938,951	9,769	2.06 / 4.67	

Table 4.1 – Dataset Statistics

4.1.2 Generating ASTs

The first step in processing is to generate the actual ASTs for the desired corpus of source code. `codegraph-fmt` supports the specification of file subsets, which can be useful to restrict training on a directory and testing on another, for instance train on the library and test on examples. The library scrapes the specified files and uses PYTHON's standard AST library to parse them. For convenience, the generated ASTs are saved in two separate formats: `.ast`, a binary pickle file for fast loading, and `.txt` for manual inspection.

Once the ASTs are generated, they can be manipulated to fit the task at hand. To do so, we implement custom AST walkers through PYTHON's standard module for manipulating its abstract syntax grammar. When the `ast.parse()` function is called, a set of `ast.NodeVisitor` objects traverse the AST in a BFS-way. By sub-classing these visitors either in the generic setting - for all nodes - or in token-type specific ways, we can manually extract all the relevant information from the nodes and their edges. Note however that there are some limits to the information that can be gathered. For example, due to the dynamic nature of PYTHON code, things like data-flow, return edges, or type information cannot be computed from the AST alone.

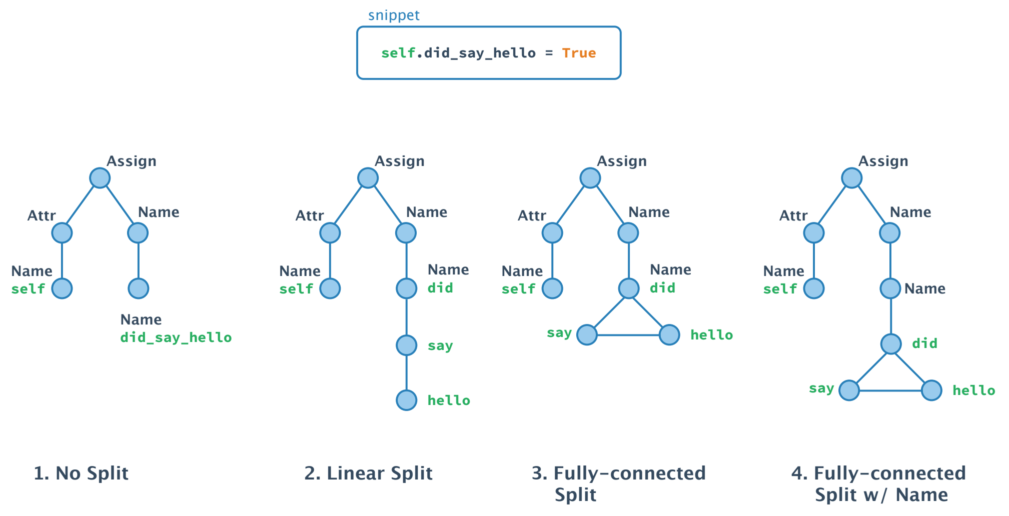


Figure 4.1 – Connectivity modes of string literals.

The information we extract matches the granularity level described in Section 2.2. Each node of the AST, roughly corresponding to a token in the source code, becomes a node in the generated graph. The graph can also be configured to include extracted literal names, usually for methods, classes and variables, when applicable. These can either be placed whole, resulting in a smaller but more difficult to predict vocabulary, or be split according to their morphology - camel-case and underscores are currently supported - to leverage their co-occurrences. When the literals are split, they can either be tacked together linearly in the graph, or as locally fully-connected components. This choice is illustrated in Fig. 4.1.

To each token can be associated a set of node features, if the experiment requires it. This can be configured to be any property: we experimented with anything from pre-trained word-embeddings, random initializations, graph-dependent features like node-degrees, identity features like the token type, constant features, ...

We experimented with several connectivity modes in the edges, from pure tree structure to attempts to densify the graph by connecting adjacent children. We found that a tree structure showed better results as it better represents the syntactic hierarchy that exists between the tokens. If the literals are split into sub-tokens, the induced subgraph can also be manipulated to either represent the succession of sub-tokens or shaped into a dense, fully-connected group of sub-tokens.

To connect multiple files, we introduced the concept of a *root node*. This virtual node serves as an anchor in each file, and can connect to a global anchor if the experiment requires a single large graph. Of course, the option to generate individual graphs per file or directory is offered by `codegraph-fmt`.

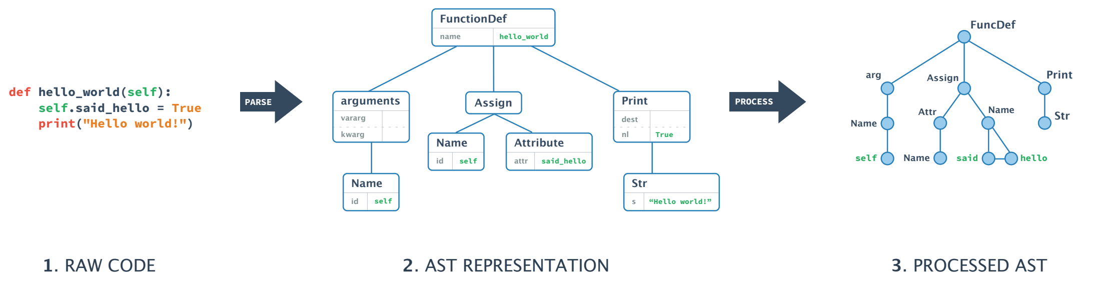
In its most generic form, `codegraph-fmt` will produce the following of files used in the downstream applications:

- `<prefix>-feats.npy` : A NumPy [Jones et al., 01] array containing, for each node, the set of specified features.
- `<prefix>-id_map.json` : A map of unique identifiers to nodes identifiers in the generated graph.
- `<prefix>-file_map.json` : A map of root nodes to the source code files they connect.
- `<prefix>-source_map.json` : A map of node identifiers to positions in the original source files (line and column indices).
- `<prefix>-G.json` : A networkx compatible graph of the generated AST.
- `<prefix>-var_map.json` : A map of extracted variable name literals to their respective node identifiers.
- `<prefix>-func_map.json` : A map of extracted method name literals to their respective node identifiers.

This format was found to be extremely versatile, easily providing all the necessary information for the set of presented experiments and the inspection of their results. The entire processing pipeline is also configurable at will, through a set of options specified in a `YAML` format, examples of which are provided in the linked repository.

4.1.3 Generating valid snippets

Where earlier experiments, aimed at testing parts of Section 2.4, focused on learning over entire source graphs, at a file or project level, the MARGARET scheme is more adapted to shorter input representations. The input to the model is a subgraph of code tokens. Akin to a sentence in natural language, a code *snippet* of length N is a valid subset of code which contains no more than N tokens. Its connectivity is arbitrary, as the adjacency matrix that represents it is fixed at a size of $N \times N$. In the degenerate case, a fully-connected ($A = \mathbb{1}_{N\times N}$) adjacency matrix reduces to the fully-connected setting found in BERT, or an identity ($A = \mathbb{I}_N$) reduces to a pure sequence-learning problem. In general we provide the adjacency matrix representing the generated AST.



```
def hello_world(self):
    self.said_hello = True
    print("Hello world!")
```

Figure 4.2 – Data processing pipeline.

Since the computational cost of MARGARET is highly dependent on the snippet size, it is crucial to find a balance between the expressiveness of large subgraphs and the efficiency of local neighborhoods. Empirically, we found that a snippet of 64 tokens usually retained $3-10$ lines of code, enough to incorporate interesting code elements from `StackOverflow`-like snippets to reasonable function definitions.

For the relevant set of experiments, we compute these snippets through an additional layer of post-processing of the ASTs generated by the method in Section 4.1.2. We propose two methods to generate snippets, both iterating through the nodes looking for valid subgraphs. Note that this procedure could be related to a biased random-walk sampling which is commonly used in GCNs to create the computation graph for a given node. The first method simply looks for sub-trees through a BFS traversal. If a valid subgraph of the correct size is found when the leaves are reached, it is added to the pool of snippets. The second method consists of higher-order ego-networks, usually of order $k = 2-4$ for the considered snippet sizes, around the considered node.

These generated snippets are saved into pairs of files for their use in the experimental pipeline:

- `<snippet_uid>-tk.txt` : A flattened representation of the tokens in a given snippet.
- `<snippet_uid>-adj.mtx` : A sparse representation of the adjacency matrix of a given snippet.

While the presented pipeline might appear costly and somewhat convoluted, it is merely the product of a desire to broaden the experimental setting. In a real-world setting, these redundant steps would be fully optimized to run efficiently, in parallel and in a single pass. This is commonly done for large datasets, for example in source code representation learning problems, where [Alon et al., 2019] pre-compute and cache 1M AST paths. Here, we describe the setting as generally as possible to apply the data generation process to the majority of experiments that were run.

Name	Range	Description
`nb_masked_tokens`	1-10	Number of tokens masked in training instance
`mask_probability`	0.15	Probability for uniform sampling of masked token
`noise_factor`	0.1	Probability of adding a random incorrect token to the training instance
`dupe_factor`	50	Number of generated training instances from each input instance
`max_seq_length`	64-128	Maximum length (resp. number of nodes) of input sequence (resp. graph)

Table 4.2 – Dataset Generation Hyperparameters

4.2 Pre-training a Language Model

4.2.1 Preparing data for a semi-supervised learning task

One of the clear advantages of the semi-supervised formulation of the Language Model setting is the ability to generate large amounts of training data without expert or even simple manual annotation. The random fraction of masked elements in the input sequence serve as training labels for the LM objective, as described in Section 3.3.

In [Devlin et al., 2018], the authors describe a set of perturbations they apply to the input data that seems to strengthen the model's ability to learn a robust LM in BERT. As a balance to the fact that during the eventual fine-tuning phase the `[MASK]` token is never seen, the authors propose to not always replace the masked word with the actual `[MASK]` token. Instead, with probability `mask_probability`, the masked word is replaced with a random word from the corpus, and with the same probability it is not replaced at all. For each snippet, the procedure can be repeated several times in order to augment the training dataset. Each of the `dupe_factor` times, a different token is sampled at random to be masked.

We replicate this behaviour, with the hyperparameter set provided in Table 4.2.

4.2.2 Experimental setting

We train multiple versions of MARGARET, the implementations of which are bootstrapped off of the implementation of BERT in Tensorflow [Abadi et al., 2015] released by [Devlin et al., 2018]. [5] We run the task on a single `Titan V (Pascal)` GPU, which we found to process btween 150 graphs/s for the largest model, and up to 1500 graphs/s in the case of MARGARET-small, a smaller, more parameter-efficient version of the model. The settings for each model are described in Table 4.3, and the individual configurations are also released along with the rest of the code as model configuration files. [6]

	MARGARET		Baseline
	SMALL	LARGE	BERT
`hidden_size`	384	768	768
`intermediate_size`	1,028	3,072	3,072
`num_attention_heads`	6	12	12
`num_hidden_layers`	3	12	12
`learning_rate`	5×10^{-5}	5×10^{-5}	1×10^{-5}

Table 4.3 – Model Hyper-parameters

[5]https://github.com/google-research/bert
[6]https://github.com/dtsbourg/magret

For all models presented here, we set a batch size of 32, a `max_seq_length = 64`. We use the Adam optimizer [Kingma and Ba, 2015] with parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$, with a rate warm-up over 10,000 steps which is then decayed linearly. A L_2 weight decay of 0.01 is also applied. We use a dropout probability of 0.1 on all layers. As in the GPT(-2) [Radford et al., 2019], we use a `gelu` [Hendrycks and Gimpel, 2016] activation function. We use the same vocabulary, training and testing examples in between baselines and try to keep comparable hyper-parameters to ensure fair comparisons.

4.2.3 Results

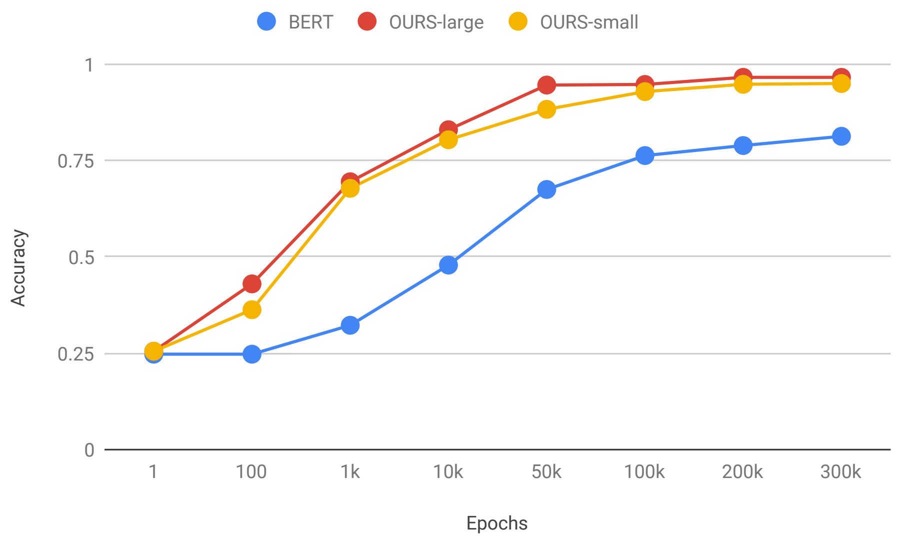


Figure 4.3 – Test accuracy on the Masked Language Model task

The results shown in Fig. 4.3 indicate that the addition of structure constitutes a strong inductive bias for the model to latch onto. MARGARET outperforms one of the strongest LM and sequence learning architectures [7] in BERT, and converges much faster, reaching very high accuracies after only 100k epochs.

In Fig 4.3, we also include results for the smaller version of MARGARET. Given the node degree distribution of the datasets shown in Table 4.1, we can estimate that for a tree of 64 nodes and an average node degree of ≈ 4, it will take 3 propagation steps for information to propagate to all neighbors ($4^3 = 64$). This insight is confirmed in Fig. 4.4, which shows that for a depth larger than 3 we only achieve marginal gains. The computational benefit is enormous however, with MARGARET-SMALL able to process an order of magnitude as many graphs as MARGARET-LARGE.

[7] At the time of writing, March 2019.

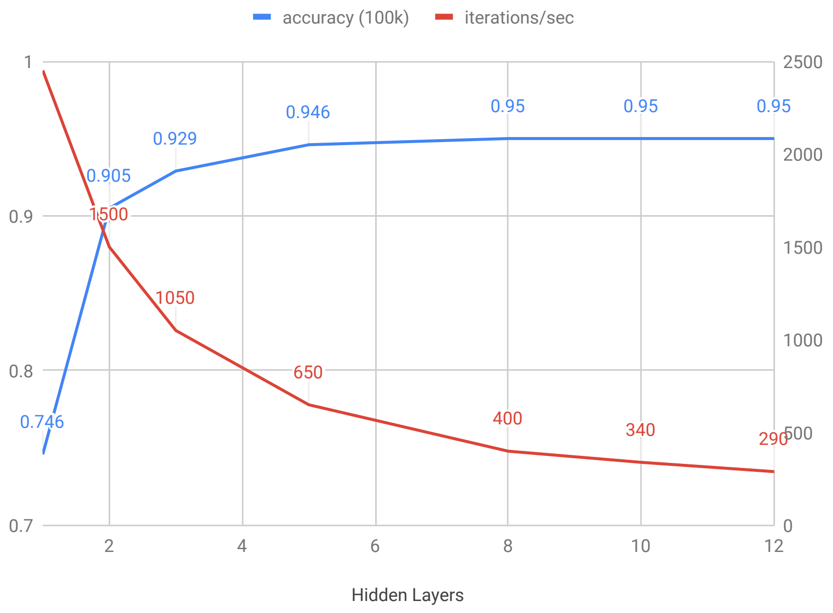


Figure 4.4 – Test accuracy after 100k iterations and computation cost with respect to model size.

4.3 Transferring to supervised tasks

As described in Section 3.5, a major advantage of powerful LMs is the ability to transfer learned knowledge from the semi-supervised setting to downstream , supervised tasks. Where pre-trained models usually served as sole feature extractors, now the same models can now be fine-tuned to perform well on subsequent tasks.

Here, we apply the fine-tuning objective to two supervised task of high relevance to the field of machine learning on code. These benchmarks have been used in several previous works, allowing a decent comparison between our solution and the current state-of-the-art, and represent a challenging evaluation of the model's syntactic and semantic abilities. Both are oriented around naming conventions in source code, which has been shown to be an important characteristic of the way developers write code [Takang et al., 1996, Liblit et al., 2006] and is essential for the human understanding and communication of source code [Allamanis et al., 2015].

4.3.1 Method Naming

The method naming task is akin to the extreme summarization task introduced in [Allamanis et al., 2016], where the model is asked to produce a short but descriptive name for a code snippet. This name should capture the internal semantics of the method but also its external usage, for example in the case of an exposed API.

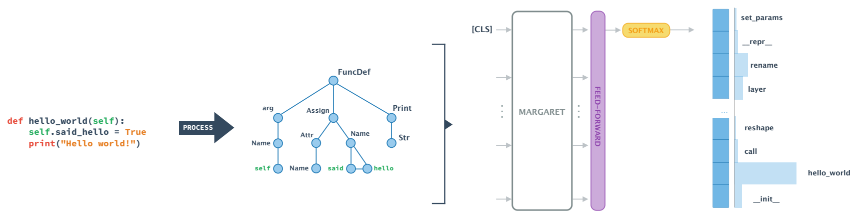


Figure 4.5 – The METHODNAMING task.

The task can be seen as a proxy for similarity in the sense that if the model is able to predict the same, or similar hopefully semantically relevant, names for two related snippets of code, this is a hint that it has captured the similarity between the two.

We pose this task as a multi-class classification problem, where a method definition is provided to the model which must assign a name by choosing from a global vocabulary, containing 400 / 2,500 / 10,000 names for CORPUS-{SM,MID,LG} respectively. Note that the representations could also be fed as the initial layer of a GRU cell which would predict at a sub-token level, in which case MARGARET would act as contextual feature extractor. This architecture was used in [Allamanis et al., 2018b] for example.

1

```
def deserialize(config, custom_objects=None):
    return deserialize_keras_object(config,
                                    module_objects=globals(),
                                    custom_objects=custom_objects,
                                    printable_module_name='regularizer')
```

```
Predictions  0. deserialize (1.0)
             1. model_from_config (0.0)
             2. from_config (0.0)
```

2

```
def __init__(self, minval=-0.05, maxval=0.05, seed=None):
      self.minval = minval
      self.maxval = maxval
      self.seed = seed
```

```
Predictions  0. __init__ (1.0)
             1. on_train_begin (0.0)
             2. preprocess_input (0.0)
```

3

```
def __call__(self, shape, dtype=None):
        return K.random_uniform(shape,
                                self.minval,
                                self.maxval,
                                dtype=dtype,
                                seed=self.seed)
```

```
Predictions  0. __call__ (0.995)
             1. truncated_normal (0.001)
             2. transform (0.0)
```

4

```
def get_config(self):
       return {
           'mean': self.mean,
           'stddev': self.stddev,
           'seed': self.seed
       }
```

```
Predictions  0. get_config (1.0)
             1. _updated_config (0.0)
             2. _preprocess_conv3d_kernel (0.0)
```

5

```
def trainable_weights(self):
   if not self.trainable:
      return []
   weights = []
   for cell in self.cells:
      if isinstance(cell, Layer):
         weights += cell.trainable_weights
   return weights
```

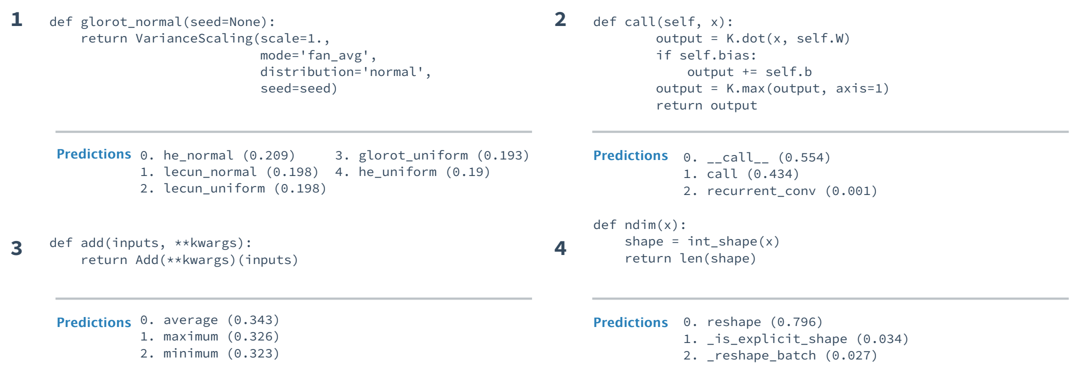
```
Predictions  0. trainable_weights (1.0)
             1. non_trainable_weights (0.0)
             2. get_weights (0.0)
```

6

```
def updates(self):
   if hasattr(self.forward_layer, 'updates'):
      return self.forward_layer.updates +
             self.backward_layer.updates
   return []
```

```
Predictions  0. updates (0.999)
             1. state_updates (0.0)
             2. get_updates_for (0.0)
```

Figure 4.6 – METHODNAMING task: samples of correct predictions on source code from `keras`.



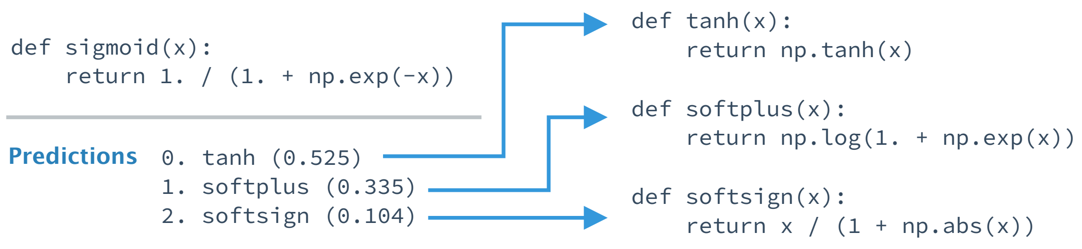
1
```
def glorot_normal(seed=None):
    return VarianceScaling(scale=1.,
                           mode='fan_avg',
                           distribution='normal',
                           seed=seed)
```
Predictions
```
0. he_normal (0.209)       3. glorot_uniform (0.193)
1. lecun_normal (0.198)    4. he_uniform (0.19)
2. lecun_uniform (0.198)
```

2
```
def call(self, x):
        output = K.dot(x, self.W)
        if self.bias:
            output += self.b
        output = K.max(output, axis=1)
        return output
```
Predictions
```
0. __call__ (0.554)
1. call (0.434)
2. recurrent_conv (0.001)
```

3
```
def add(inputs, **kwargs):
    return Add(**kwargs)(inputs)
```
Predictions
```
0. average (0.343)
1. maximum (0.326)
2. minimum (0.323)
```

4
```
def ndim(x):
    shape = int_shape(x)
    return len(shape)
```
Predictions
```
0. reshape (0.796)
1. _is_explicit_shape (0.034)
2. _reshape_batch (0.027)
```

Figure 4.7 – METHODNAMING task: samples of incorrect predictions on source code from `keras`, ranked by likelihood.



```
def sigmoid(x):
    return 1. / (1. + np.exp(-x))
```
Predictions
```
0. tanh (0.525)
1. softplus (0.335)
2. softsign (0.104)
```

```
def tanh(x):
    return np.tanh(x)

def softplus(x):
    return np.log(1. + np.exp(x))

def softsign(x):
    return x / (1 + np.abs(x))
```

Figure 4.8 – METHODNAMING task: an illustration of hybrid concerns.

	Reported	Description
[Iyer et al., 2016]	0.275	RNN+Attention on textual representation of JAVA source code. Original work is done on C#/SQL ([Alon et al., 2019] for reported).
[Allamanis et al., 2016]	0.473	CNN+Attention run on JAVA source code.
[Alon et al., 2018]	0.511	Learning a CRF on paths generated from Python AST code (Accuracy measured @7).
[Alon et al., 2019]	0.633	RNN+attention embedding of paths on the AST, run on a filtered subset of JAVA code.
Ours	0.76	Generalized TRANSFORMER model run on Python code (CORPUS-lg).

Table 4.4 – Method Naming Results - Literature.

The accuracy of such a task is difficult to define: a first, formal interpretation of the multi-class classification problem would aim to measure the accuracy of predictions. These results are summarized in Table 4.5 for all three corpora. We restrict the test set to methods that appear at least once in the sample training set, and report exact match accuracies as well as a subtoken accuracy, which awards partial accuracy when the prediction and label have common subtokens. The score is 1 is all subtokens are predicted. In all settings accuracies are reported @1.

We report several scores gathered from the literature in Table 4.4, with a quick note on the methodology behind them. Usually, these methods award different accuracy schemes to their predictions, and are often run on different languages. This is issue is discussed in more detail in Appendix A, but we report strong results which compare favorably to the literature, across datasets. We also show the large improvement over the standard BERT, which overfit to always predict the most frequent token.

	F1-Macro	F1-Weighted	Subtoken Accuracy @1
MARGARET			
CORPUS-SM	0.82	0.85	0.86
CORPUS-MID	0.68	0.76	0.81
CORPUS-LG	0.53	0.76	0.76
BERT			
CORPUS-SM	0.03	0.12	0.21

Table 4.5 – Method Naming Results

In order to showcase out results more qualitatively, we provide several selected examples, taken from the output of our model. We try to highlight both successful examples (Fig. 4.6) and some failures of the model (Fig. 4.7), which are also extremely informative to probe its behaviour.

In the case where the model is successful, we observe that it is often highly confident. When the model is less certain, it often is hesitating between several semantically correct classes. For instance in Example 1 of Fig. 4.7 (upper-left corner), it does not guess the correct class, but understands that this method is dealing with some uniform intializers, proposing several semantically relevant and valid alternatives. In this case, we hypothesize that it is able to latch onto structural similarities between the method's constructions.

This behaviour is further showcased in Fig 4.8, where the model is also incorrect but it predicts based on similar structural properties of the AST produced by predictions 1 and 2. Prediction 0 showcases the model's ability to also capture co-occurrence information: `tanh` and `sigmoid` were often used in similar contexts, possibly in many examples the model has learned from in the pre-training phase: both are activation functions commonly used throughout the `keras` library. This could explain why it sees these two methods as similar despite very different implementations.

We provide more detailed insight in Appendix B, notably showing the influence of label frequency on the predicted result.

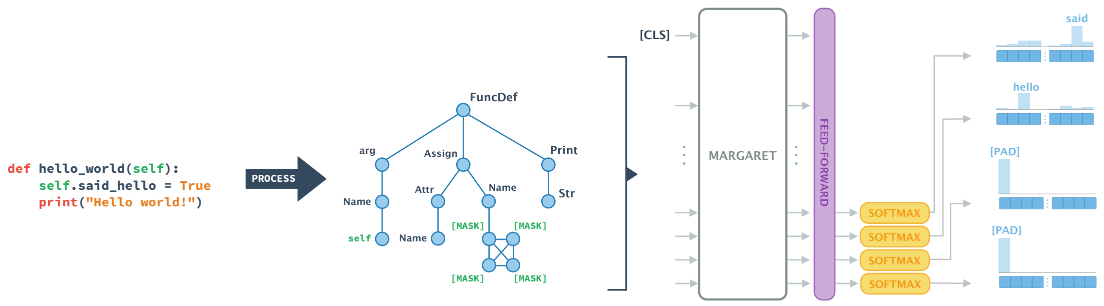


Figure 4.9 – The VarNaming task.

4.3.2 Variable Naming

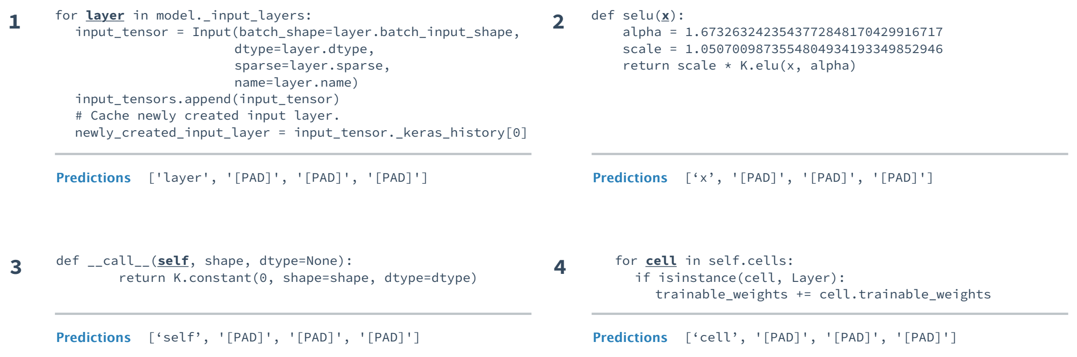
The variable naming (or VarNaming) was introduced by [Raychev et al., 2014, Allamanis et al., 2014]. The task consists of inferring the “correct”, or most likely, variable name given a context in which it occurs. Providing relevant names for variables requires some level of context-sensitivity from the model, as it must reason about where and how this token is being used to guess the correct token from a large vocabulary.

	Accuracy			
	@1	@3	@5	@7
BERT	0.3	0.43	0.48	0.52
MARGARET	**0.59**	**0.792**	**0.833**	**0.849**
[Alon et al., 2018]				
Assumed @1	0.567	-	-	-
[Allamanis et al., 2018b]				
Python	0.536	-	-	-

Table 4.6 – Variable Naming Results

To implement this task in our setting, we build upon the architecture used in the masked language model. This time however, we mask entire variable names. Since these were split into sub-tokens, as described in Section 4.1, we pad the masked variables so the masked length is always constant. We set this length to 4 tokens, which covered over 99.9% of the training set.

The model is then tasked with predicting the correct sequence of sub-tokens denominating the variable in use, complete with the padding sub-tokens if it believes the variable name is shorter than the maximum length. For this reason, we report exact match accuracy only. We also report the accuracy of predictions at several ranks in order to emulate a recommendation setting, where the developer could use the predicted names in auto-completion setting for example.



1

```
for layer in model._input_layers:
   input_tensor = Input(batch_shape=layer.batch_input_shape,
                        dtype=layer.dtype,
                        sparse=layer.sparse,
                        name=layer.name)
   input_tensors.append(input_tensor)
   # Cache newly created input layer.
   newly_created_input_layer = input_tensor._keras_history[0]
```

Predictions ['layer', '[PAD]', '[PAD]', '[PAD]']

2

```
def selu(x):
    alpha = 1.6732632423543772848170429916717
    scale = 1.0507009873554804934193349852946
    return scale * K.elu(x, alpha)
```

Predictions [‘x’, '[PAD]', '[PAD]', '[PAD]']

3

```
def __call__(self, shape, dtype=None):
        return K.constant(0, shape=shape, dtype=dtype)
```

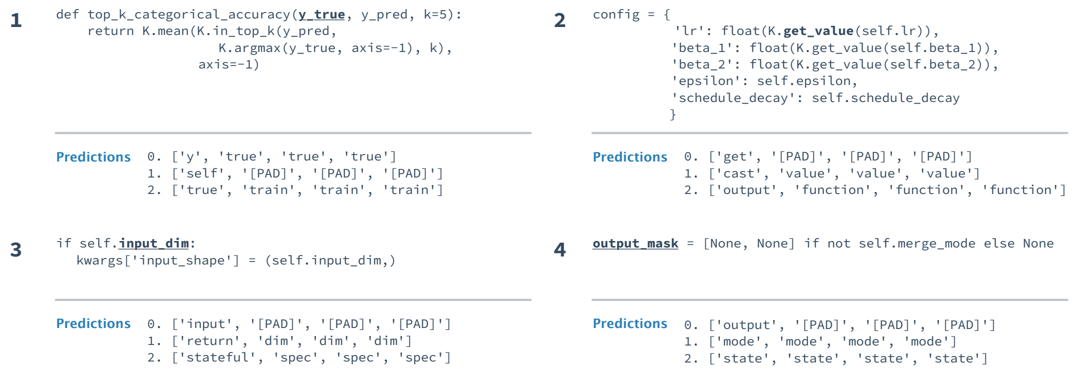
Predictions [‘self’, '[PAD]', '[PAD]', '[PAD]']

4

```
for cell in self.cells:
   if isinstance(cell, Layer):
      trainable_weights += cell.trainable_weights
```

Predictions [‘cell’, '[PAD]', '[PAD]', '[PAD]']

Figure 4.10 – VARNAMING task: samples of correct predictions on source code from `keras`.



1

```
def top_k_categorical_accuracy(y_true, y_pred, k=5):
    return K.mean(K.in_top_k(y_pred,
                     K.argmax(y_true, axis=-1), k),
                  axis=-1)
```

```
Predictions  0. ['y', 'true', 'true', 'true']
             1. ['self', '[PAD]', '[PAD]', '[PAD]']
             2. ['true', 'train', 'train', 'train']
```

2

```
config = {
          'lr': float(K.get_value(self.lr)),
          'beta_1': float(K.get_value(self.beta_1)),
          'beta_2': float(K.get_value(self.beta_2)),
          'epsilon': self.epsilon,
          'schedule_decay': self.schedule_decay
          }
```

```
Predictions  0. ['get', '[PAD]', '[PAD]', '[PAD]']
             1. ['cast', 'value', 'value', 'value']
             2. ['output', 'function', 'function', 'function']
```

3

```
if self.input_dim:
   kwargs['input_shape'] = (self.input_dim,)
```

```
Predictions  0. ['input', '[PAD]', '[PAD]', '[PAD]']
             1. ['return', 'dim', 'dim', 'dim']
             2. ['stateful', 'spec', 'spec', 'spec']
```

4

```
output_mask = [None, None] if not self.merge_mode else None
```

```
Predictions  0. ['output', '[PAD]', '[PAD]', '[PAD]']
             1. ['mode', 'mode', 'mode', 'mode']
             2. ['state', 'state', 'state', 'state']
```

Figure 4.11 – VARNAMING task: samples of incorrect predictions on source code from `keras`, ranked by likelihood.

Here again, we show strong improvements compared to the pure text-based approach. Comparing to the stock version of BERT [Devlin et al., 2018] allows us to highlight the relative gains with the addition of structure. This gain was first highlighted by the work of [Allamanis et al., 2018b], however as with the method naming task it is difficult to make an entirely fair comparison. Their method provides the model with a different structure, and the results are obtained on an entirely different experimental benchmark. We still provide these numbers as evidence that our method is competitive with state of the art results. Given its widespread use in the machine learning on code community, we believe this task should also be part of an eventual benchmark for the field (detailed in Appendix A).

In Figures 4.10-4.11, we showcase the model's prediction on the VARNAMING task, both successful and failed. The variable to predict is shown here in underlined bold. We notice that often times when the model fails to produce the correct prediction, it seems to cut the variable name short, predicting a short variable name but showing the real token as the second most likely token as predicted for the following token (*e.g.* in Examples 2,3). Note that in Example 2, we treat the class method `get_value` as a variable name: this is due to PYTHON's constraints, where variable and class attributes are not mappable directly through the parse tree. This property could have been a method, a variable, an attribute and still have the same representation in the AST. This also sets our methodology apart from the work of [Allamanis et al., 2018b] which only predicts local variables, meaning the whole scope is available.

4.4 Investigating the LMs properties

In order to investigate the structural properties of MARGARET, we design a set of controlled experiments which respectively highlight the ability for the model to model syntactic dependencies (in Section 4.4.1) and semantic constraints (in Section 4.4.2).

4.4.1 Structural properties

In this experiment, we start from a given pre-trained MARGARET model. We produce results in the regular pipeline, from pre-training to supervised fine-tuning. Then, we reiterate the fine-tuning starting from the same pre-trained model, only with randomly permuted inputs. The order of the flattened token list provided to the model as textual input is changed, but the permutations are matched in the adjacency matrix, meaning tokens and their neighbours are still aligned through the graph representation.

In Table 4.7, we report that the results are indeed equivalent when run on the METHODNAMING task, a result which is omitted for VARNAMING for the sake of brevity.

	Accuracy				MRR		
	@1	@3	@5	@7	@3	@5	@7
Standard	0.63	0.66	0.66	0.69	0.73	0.49	0.37
Random Permutations	0.628	0.65	0.67	0.68	0.72	0.478	0.36

Table 4.7 – METHODNAMING results, with and without permutations.

If the model is indeed order-invariant, the propagation of information should not be affected as it relies only on message-passing pathways between neighbours, and not by positional vicinity in the flattened representation or local co-occurrences with neighbours.

4.4.2 Token type correctness

In order to investigate further the syntactic properties of the model, we investigate grammatical correctness of its predictions, reporting a relatively small but clear improvement in favor of MARGARET. We investigate at two levels of granularity:

Token Type is a binary split between AST-provided tokens (*i.e.* language keywords) and user provided literals such as variable names, method names, class names, ...

Token Class separates the AST-provided tokens into 14 sub-classes, as defined by the Abstract Grammar provided by the PYTHON standard library. [8]

	Token Type	Token Class		
	Accuracy	Accuracy	F-1 Macro	F-1 Weighted
BERT 200k iterations	0.990	0.979	0.92	0.91
MARGARET 200k iterations	**0.997**	**0.994**	**0.96**	**0.96**

Table 4.8 – Assessing the syntactical correctness of Masked Language Model predictions.

4.4.3 Visualizing attended structures

While attention can not be considered as a robust interpretation scheme [Jain and Wallace, 2019], it certainly offers an informative view into the model's behaviour. We showcase the structures of the graph to which the model attends, for a randomly chosen example from the large corpus, in Figures 4.12-4.13. Each figure shows edges weighted as a function of the attention weights extracted from different layers of the model - here we used MARGARET-small, of depth 3 with 6 attention heads.

[8]https://docs.python.org/3/library/ast.html

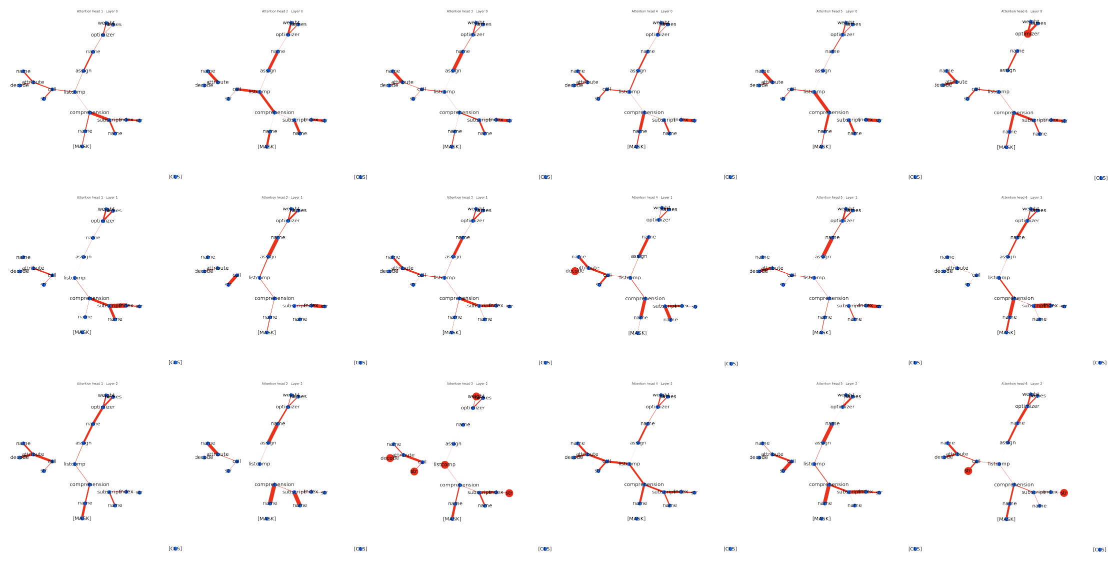


Figure 4.12 – Attention on the input graph. *(best viewed on screen)*

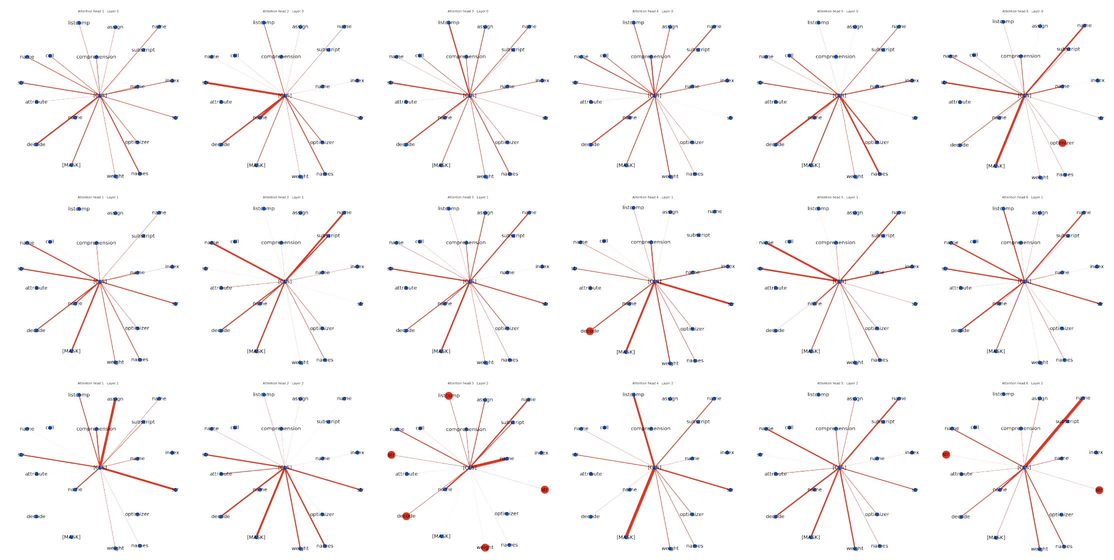


Figure 4.13 – Attention weights on the pooling token [CLS]. *(best viewed on screen)*

Of course, the analysis of attention weights is qualitative and circumstantial, so we refrain from making causal claims from these visualizations alone. However, we can observe some behaviours which can hint at interesting semantic structures highlighted by the model.

Fig. 4.12 showcases the attention weights on the original input graph, highlighting the localized part of the aggregation scheme as defined in Section 3.2. As the depth grows, we notice longer chains contiguous attended structures, *e.g.* the long chain attended by attention head #4

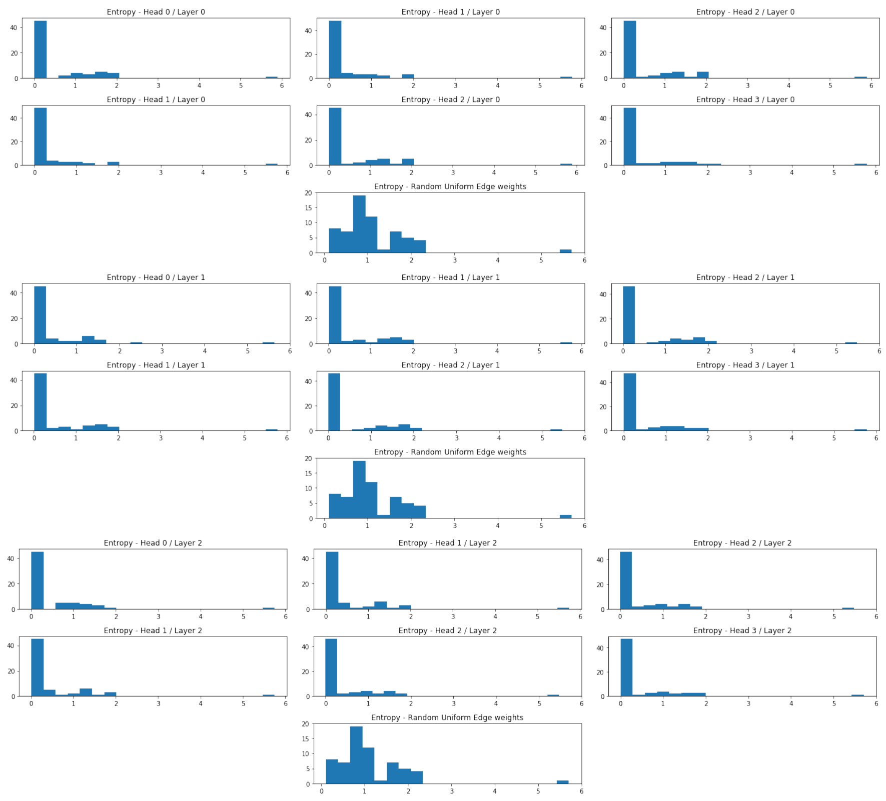


Figure 4.14 – Entropy distribution of attention weights compared to uniformly random weights.

in layer 3, showcasing the model's ability to grow its receptive field as the depth increases. We can also note that each attention head seems to focus on different relevant edges or nodes, showing that the addition of parallel attention heads allows each node to aggregate multi-modal information from its neighborhood.

In Figure 4.13, we show how the `[CLS]` token, the only node fully connected to the rest of the network, acts as a pooling operator. It selectively aggregates information from the graph, providing evidence for the global behaviour of the model. Here, each successive layer also seems to show a consideration for more abstract structures. Where the first layer looks at almost every token, often quite uniformly, in latter layers the attention heads seem to specialize to larger subgraphs, for instance heads 1 and 2 look at two different halfs of the subgraph.

Another way to showcase the use of attention in this model is to consider the entropy of edge weights as computed by the attention mechanism. This entropy can then be compared to that of randomly distributed uniform weights. If there is a difference in distributions, then the attention mechanism serves as a differentiating factor for learning the different importances of different edges. In Fig. 4.14, we notice much sharper distributions for the attention weights than uniformly random weights.

	ENZYMES					
	Ours	Freq	GCN	GraphSAGE	DiffPool	WL
Test Acc.	**0.68**	0.16	0.64	0.54	0.62	0.53

	MSRC-21		
	Ours	Freq	GCN
Test Acc.	0.90	0.05	**0.92**
@3	**1.0**	0.15	-

	MUTAG				
	Ours	Freq	GCN	DGCNN	WL
Test Acc.	0.81	0.55	0.76	**0.85**	0.80

Table 4.9 – Results for the Graph Classification dataset

4.5 Extension to pure graph tasks

In order to confirm our intuition that the generalized form of the TRANSFORMER can be used as a GNN, as detailed in Section 3.1.2, we benchmark our model on purely graph-based tasks, both in the context of graph classification (Section 4.5.1) and of node classification (Section 4.5.2). In both instances we show competitive results, on par with the state of the art.

4.5.1 Graph Classification

We start by the most straightforward extension of the model, natively supported by MARGARET's architecture. Indeed, the model readily accepts a list of node labels and the adjacency matrix that denotes the edges between them. While the model could support node features natively by replacing the random initialization of the look-up table, we focus on featureless labeled graphs to showcase the structural abilities of MARGARET.

We run the graph classification benchmark on three popular datasets taken from [Kersting et al., 2016], with results shown in Table 4.9. All these graphs are run with a `max_seq_length` value that did not incur any particular strain on the computational side (chosen to be between 54-128), while retaining almost all the graph in the proposed task.

	CORA			
	Ours	Freq	L-GCN	GCN
Test acc.	**0.83**†	0.16	**0.83**	0.81

† Label Propagation setting

Table 4.10 – Results on node classification

MSRC-21 A 21 class dataset of semantic image processing introduced by [Winn et al., 2005]. Each image is represented as a MRF which connects different super-pixels of the image. The goal is to predict a semantic label, *e.g. building, grass,* ... from one of the 21 classes.

ENZYMES A 6 class dataset of protein structures from the BRENDA database [Schomburg et al., 2004]. The goal is to predict their EC number, which is based on the chemical reaction they catalyze.

MUTAG A binary classification problem, where the goal is to predict the mutagenic effect of chemical compounds.

4.5.2 Node Classification

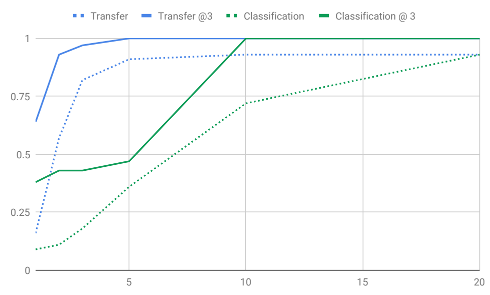
The extension to node classification is also straightforward in our model, and can be achieved in two ways. The first emulates the masked language model: the labels of one or several masked nodes is predicted in-place. The second way operates in a more node-centric way. Given a node v, we sample its computation graph C_v through adequate sampling and search procedures (usually a simple BFS or ego-network expansion until the maximum number of nodes is reached). The task then becomes graph classification where the label for node v is propagated through the special `[CLS]` token.

We showcase our method on the CORA dataset: a network of several thousand publications is connected through citation edges. The goal is to predict the correct topic, one of 7 related technical areas. As with graph classification, we could integrate node feature information by modifying the initial look-up table, but we focus on the structure aspect by providing labels for the node's neighbours. This is dubbed *label propagation.*

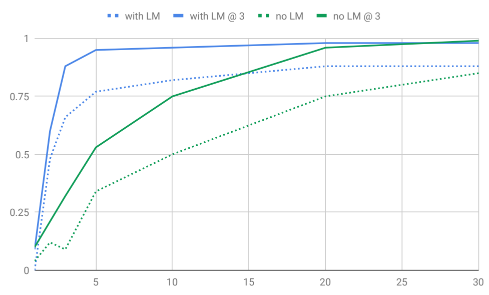
4.6 Pre-training and transfer learning on graphs

Inspired by the successes of NLP in transfer learning and pre-training models, and given the successful results of fine-tuning our model on code-related tasks, we set out to investigate the idea of transferring knowledge between a model pre-trained on a semi-supervised objective on graphs.

First, we show the benefit of pre-training our model on a node classification task (the analog



(a) Pre-training test accuracy on MSRC_21



(b) Transfer learning test accuracy on MSRC_9

of the Masked Language Model task on graphs) for downstream supervised fine-tuning. In Fig. 4.15a, we show the results on the MSRC_21 task, which we used as a benchmark in the graph classification tasks (see Section 4.5.1). The model with pre-training converges much faster than the model without, indicating that the model is leveraging some existing knowledge that is has acquired during pre-training.

The MSRC_21 also has a sister dataset, which contains different graphs under the same form, but with a different classification objective: this time, there are only nine semantic labels to choose from. This dataset is hence named MSRC_9. We show in Fig. 4.15b that again the model is able to leverage information acquired during pre-training, and is able to transfer this information efficiently to a new dataset with a different classification task.

5 Conclusion

Inspired by the extremely successful TRANSFORMER architecture from Natural Language Processing, we propose an architecture that extends to arbitrary graph-structured data. This model is able to efficiently compose local and global co-occurrence patterns to achieve deep contextual embeddings of both a graph and the individual nodes that compose it.

Learning contextual embeddings of source code structured through the Abstract Syntax Tree representation as well as classic graph tasks, we show the method's versatility. Additionally, this method enables pre-training on graphs, wherein the model learns from a semi-supervised tasks in order to more efficiently learn a related fully-supervised task downstream.

This methodology shows great promise in its ability to capture relevant patterns in source code, providing state-of-the-art results on standard methods for the field of machine learning on source code. However, these methods are only proxies to the eventual goal of modeling the semantic similarity of code fragments. As such, the method could still benefit from several improvements to handle more diverse inputs and handle multiple concurrent tasks to bring us closer to the final similarity objective.

Student Signature.

[illegible]

A Machine Learning for code: a state of affairs

A.1 On the reproducibility of ML for code

In the early phases of this thesis, an effort was made to survey the field of machine learning on source code. This work was essential in order to gain a grasp on the types of problems being tackled and the technologies researchers prefer to use. It allows to paint a picture of the tasks against which these methods are being tested and the state of the art results that are obtained. It allows the discovery of datasets on which these tasks are run, helping to understand how to handle data formats as particular as source code. However this work quickly became tedious. Somewhat unsurprisingly, the breadth of work and the multiple channels through which they are disseminated made it extremely difficult to get a clear picture of current and related work.

Aided by a main survey [Allamanis et al., 2018a] conducted of the field and a few online repositories [1], we still managed to converge to a list of around 170 pieces of work related to our setting of representation learning on source code, many of which are cited in this work. In order to compare our method rigorously and fairly, it is necessary to have available both runnable source code and the dataset that was used in the reported experiments. The first would ideally allow the comparison of our method on a dataset of our choice. The second would hopefully help compare our method on the same dataset as a previous work. Together, they would offer an entirely reproducible pipeline, helping advance the field by a fair and accurate comparison of each contribution.

Unfortunately, the resulting statistics showed a dire picture of the space. Of the 170 publications originally selected, only 7 offered a fully reproducible pipeline.

Furthermore, the field appeared fragmented in the kinds of tasks it is solving or showcasing. Some light consensus emerges around tasks like variable name prediction or method naming, which prompted us to validate our method against them (see Section 4.3.1- 4.3.2). However a wide variety of other tasks and domains are proposed, such as, among many others, variable misuse [Allamanis et al., 2018b], algorithm classification and performance prediction [Ben-

[1] https://ml4code.github.io/ - https://github.com/src-d/awesome-machine-learning-on-source-code

Nun et al., 2018], or even type inference [Hellendoorn et al., 2018].

It is rare to find intersectional tasks, which makes the properties of different code representations or methodologies difficult to compare and evaluate. There is no clear state-of-the-art or any computational methodology to probe models for common characteristics in the way they solve problems.

To add to the difficulty of comparisons, source code is an extremely heterogeneous datasource. It is inherently multi-lingual, with no clear way of evaluating the "difficulty" of running tasks on languages A vs. B. Empirically, some have noticed that certain languages are more difficult to handle, such as PYTHON, but there is no consensus on an explanation.

Each language has its own syntactic constructions, and some provide more semantic information than others (type information, library linking, data-flow edges, ...). It also has its own community of practitioners, the people who actually write the code. While this provides a strong footing for the *naturalness assumption*, it also can add noise to datasets. Public repositories have no standards for code quality, do not prevent idiosyncrasies nor restrict duplication [Lopes et al., 2017, Allamanis, 2018]. Code also evolves across versions of the standard libraries, with no guarantee of backwards compatibility.

Many tools also function at very different levels of code representations, each with its own level of abstraction, syntactic and semantic annotations. The array is large, and goes anywhere from the purely textual representation [Pu et al., 2016, Bhoopchand et al., 2016, Allamanis et al., 2014] or a semantically-augmented forms [Allamanis et al., 2018b], execution traces [Zaremba and Sutskever, 2014], the AST provided the language's standard library [Maddison and Tarlow, 2014, Raychev et al., 2014], all the way down to bytecode [Nguyen et al., 2016, Si et al., 2018] or Intermediate Representations (IR) [Ben-Nun et al., 2018].

A.2 SCUBA: Semantics of Code and Understanding BenchmArk

All of these observations place a non-trivial barrier in the study of the state of affairs in the field of machine learning on code. For this reason, we argue that the development of a benchmark suite for the robust and fair evaluation of different methods is crucial for domain advancement, similar to the so-called "IMAGENET moments" of Computer Vision and now NLP (see Section 3.5). Our ambition is not to propose an equivalent to the incredibly painstaking effort that was IMAGENET [Deng et al., 2009], but to offer a clean slate off of which best practices can be developed.

This work does not claim to solve the concerns raised here, but rather to present some desiderata arising from the pains we faced in developing new methods for machine learning on code. We hope to pursue the ideation process, including more of the community to work jointly on this problem, in order to eventually propose a fully-fledged benchmarking platform for the field at large.

A.2.1 Reproduciblity Checklist

The first step we envision would be the adoption of a set of best practices for reproducible work. Inspired by the influential Reproduciblity Checklist for Machine Learning released by Joëlle Pineau [2] and the R - Open Science equivalent, [3] we propose an adapted version specific to machine learning on code. This is initial draft should serve to resolve many of the issues that we encountered when trying to reproduce existing work, but could certainly warrant some more refinement before it can reach widespread adoption.

Data

- Is the data available? If yes, in which form?
  - ☐ Raw data.
  - ☐ Pre-processed data.
  - ☐ Output data.
- Is the pre-processing pipeline explicit?
  - ☐ What filters are applied? (*e.g.* removing low-frequency elements)
  - ☐ Which assumptions are made when generating the data? (*e.g.* snippets should be valid bits of code)
  - ☐ What transformations are applied to the original dataset?
  - ☐ What is the final representation that is passed to the model?
- Is the meta-data fully specified?
  - ☐ What is the origin of the corpus.
  - ☐ If the raw source forming the dataset is available online, are hashes or fingerprints of its version shared?
  - ☐ Is the programming language specified, including its version?
  - ☐ What are the Train / Test / Validation splits?

Code

- Is the entire pipeline available? This includes the following components:
  - ☐ Data collection.
  - ☐ Data pre-processing.
  - ☐ Main algorithm loop and architecture.
  - ☐ *(Optional)* Post-processing steps.
  - ☐ Output in a form matching that of reported results.

[2]https://www.cs.mcgill.ca/ jpineau/ReproducibilityChecklist.pdf
[3]https://ropensci.github.io/reproducibility-guide/sections/checklist/

- Is there a runnable version of the code provided? This includes the specification of:
  - ☐ The source platform and hardware specifications.
  - ☐ Dependency version information.
    or
  - ☐ A reproducible container which packages the entire project.

Model

- Is the algorithm fully specified?
  - ☐ Hyperparameter sets.
  - ☐ Computational Cost analysis.
  - ☐ Number of iterations to convergence.
  - ☐ Ablation study.
  - ☐ Pre-trained model.
- Is the evaluation task fully specified?
  - ☐ Objective
  - ☐ Metric
  - ☐ Labels

A.2.2 A standardized benchmark

Second, we would like to see a set of tasks designed to evaluate the semantic understanding of machine learning models operating on code. They are inspired in structure by the GLUE benchmark suite [Wang et al., 2018a], as well as other novel benchmarking tools that evaluate recent NLP architectures on a variety of challenging tasks to better probe their abilities. For each of the proposed tasks, a standard dataset could be proposed. While the construction of said datasets is outside the scope of this work, the endeavour is in progress at the time of writing, based off the collection of 3TB that we have collected. The goal is to cover a large spectrum of curated examples with varying level of difficulty and a broad set of semantic abilities necessary to solve them. In a first iteration, we focus only on PYTHON as it is the language of choice for this work. However, we hope to extend it in the future to other classes of languages.

The proposed tasks could find their place in three main categories:

Inference tasks. Predicting a label or property of a set of tokens from the input, akin to node classification tasks. Here, we present VarNaming as an instance of this task.

Snippet-level Evaluation. Predicting a label or property of an entire chunk of the input, such as a snippet, akin to graph classification. Here, we present MethodNaming as an instance of this task.

Similarity measures. Predicting a label for sets of inputs. These tasks can include anything from predicting the similarity of two snippets to performing link prediction to match contiguous snippets.

A.2.3 A public leader-board

Once a standardized comparison of machine learning on code has been proposed, it is in the field's best interest to open its use to all. Like many other popular challenges before it, we propose to maintain a public leader-board to more openly distribute the results of this benchmark. Authors of new architectures would be provided with easily downloadable datasets, representing the different evaluation tasks presented previously. Authors can then run this benchmark locally, though we also hope to provide a user-friendly computational environment on which the new algorithms can be benchmarked, for example through platforms like `Binder`.[4]

Each submission would be graded based on its results on the proposed tasks, providing a clear picture of the state of the art. Of course, the advancement of the field is cannot be entirely measured through scores on an artificial benchmark, no matter how well engineered. Rather we envision SCUBA as a platform to share model details and parameters more openly, provide standardized statistical insights on the differences and capabilities of proposed models, advocate for the reproducbility and open-sourcing of research material in general, and, overall, advertise the field of machine learning on source code, a promising research direction that is still in its infancy.

A.2.4 On the reproducibility of the presented work

In line with the aforementioned guidelines for reproducibility, we ensure that we are able to check the desired characteristics, both on the reproducibility checklist for machine learning and the proposed checklist for research on machine learning for source code.

[4]https://mybinder.org/

Data.

All the data that was used throughout this work is made available in a persistent way on the Stanford Digital Repository [5], which provides long term storage for scientific data. We provide access to all three corpora, each in their raw, pre-processed, and ready-to-train states.[6]

Code.

Three libraries have been developed through the course of the project. Each of these is released under the Apache 2.0 License [7]. All packages include a `README.md` which contains explicit commands for running the experiments reported in this work. Whenever applicable, the relevant configuration files are shared in their respective repositories. Finally, each repository contains several `IPython / Jupyter` notebooks which contain the code used to inspect results, probe the model, and produce the reported figures.

`codegraph-fmt` is available at https://github.com/dtsbourg/codegraph-fmt
CodeSAGE is available at https://github.com/dtsbourg/codesage
MARGARET is available at https://github.com/dtsbourg/magret

Should any questions arise from the use of this code, any link go stale or errors be found, please open an issue on the `github` repository, or contacting the author at the following address:

`contact@dtsbourg.me`

Miscellaneous.

The LaTeX source code for this report is also made available, [8] along with the raw `Sketch` files with which most of the figures were created.

[5]sdr.stanford.edu
[6]The link will be updated upon delivery of the final PURL by SDR.
[7]https://spdx.org/licenses/Apache-2.0.html
[8]https://github.com/dtsbourg/Multimodal-Representation-Learning-for-Code-Similarity

B Additional insights into task results

We provide some additional statistics about the dataset used in the experimental section, particularly regarding the distribution of labels in the different tasks.

B.1 Masked Language Model

The distribution of tokens in source code is highly skewed. The most common token, `Name`, which is a language keyword, contributes for almost a quarter of all tokens in the parsed AST. We show this distribution by using the labels used in the test set of the masked language model. Since these are uniformly sampled across the provided dataset we assume this to be an unbiased estimate of the true distribution.

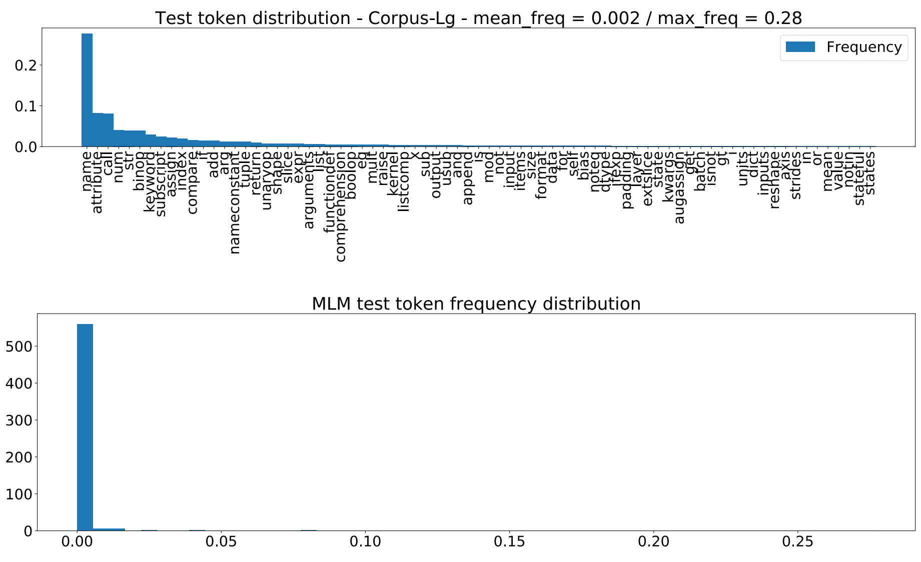


Figure B.1 – Token distributions for the masked language model.

The model shows remarkable ability to go beyond global frequency information to produce its predictions. We show little correlation between the frequency of a token and its accuracy, with the exception of tokens entirely unseen at test time. We showcase some per-token accuracies with their respective frequencies in Figure B.2.

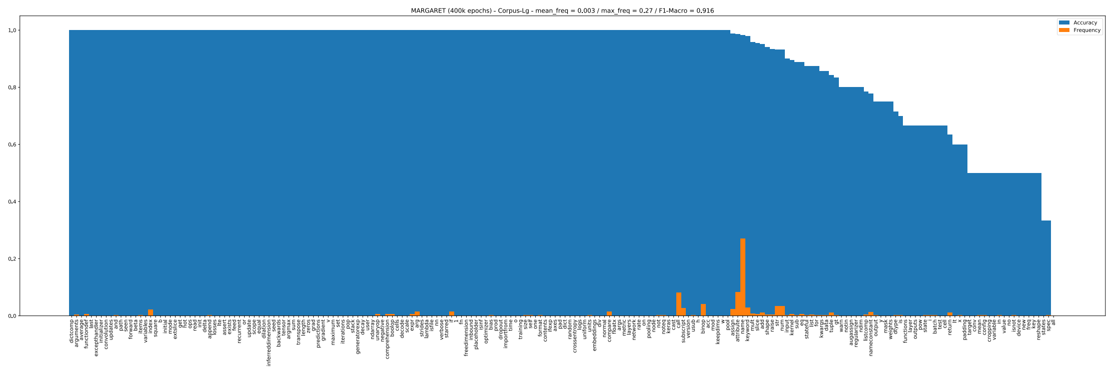


Figure B.2 – Top-150 most accurately predicted tokens and their accuracies.

Also, we show the consistency of training patterns across the different corpora. The larger the corpus, the longer the model will take to converge, but overall we achieve similar accuracy with each, as shown in Fig. B.3

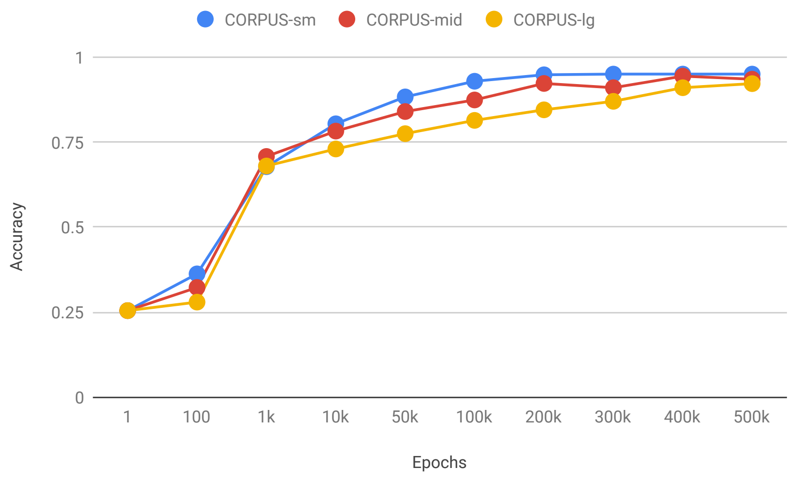


Figure B.3 – MLM Accuracy across corpora.

B.2 Method Naming

The distribution of method names has a long tail, with the `__init__` method being over-represented in the corpus compared to all other method names. This means that beyond this frequent method, the model cannot rely solely on frequency to game its predictions.

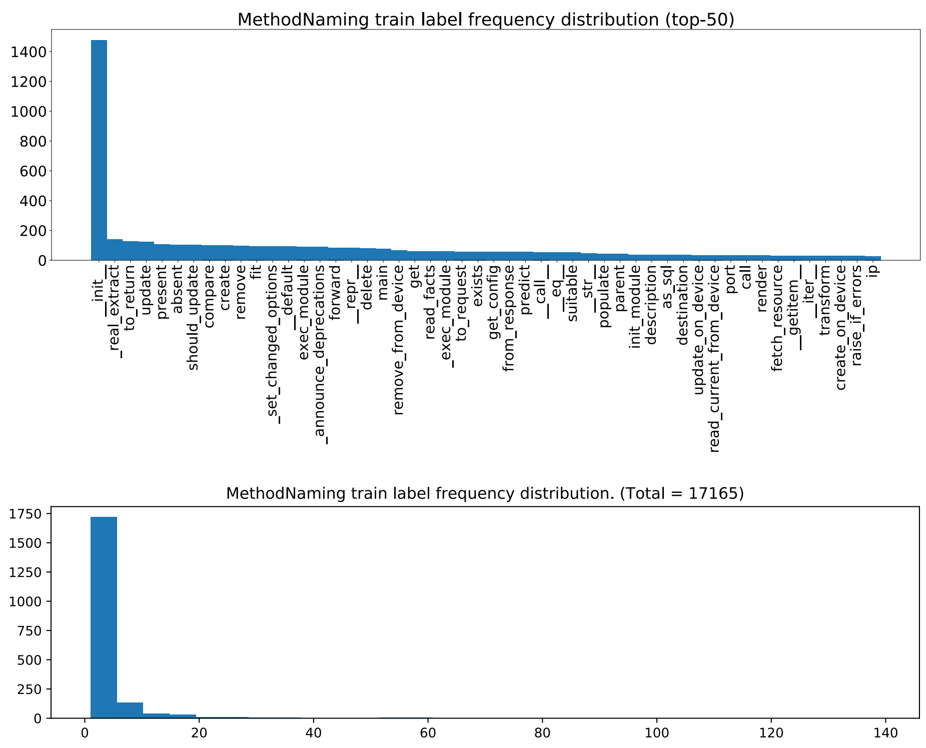


Figure B.4 – Train label distributions.

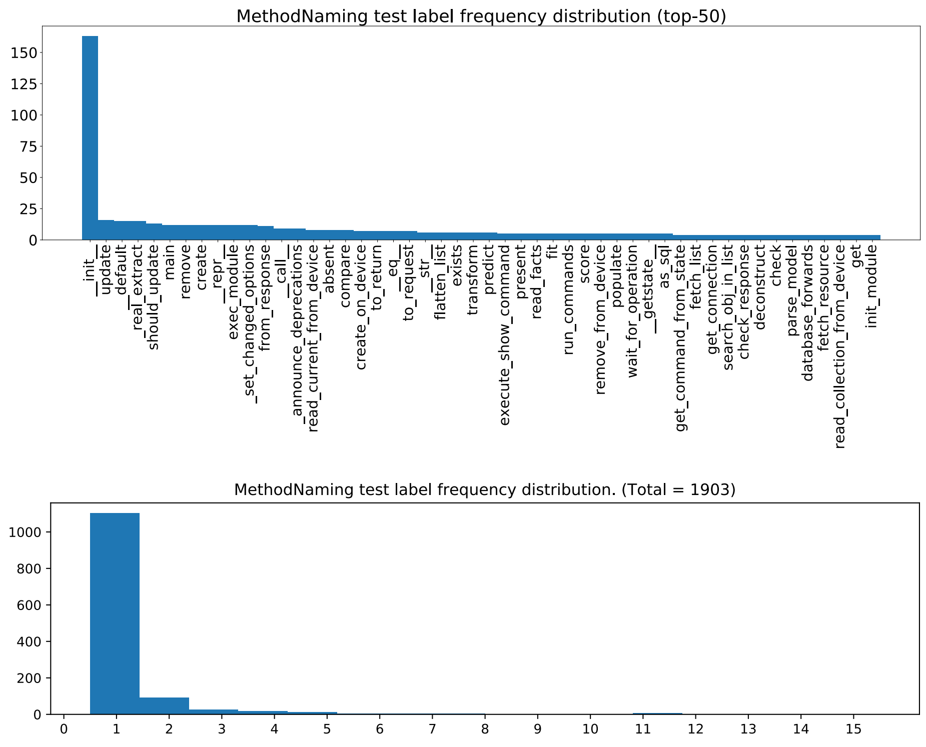


Figure B.5 – Test label distributions.

This insight is showcased further by the lack of correlation between frequency and accuracy, as shown in Figure B.6. However, this holds mostly in the case of positive examples: while the frequency does not condition the accuracy, if a method is too infrequent in the training set, the model will have trouble learning to predict it, as expected. Given the long-tailed distribution of the method names, this effect is non-negligible.

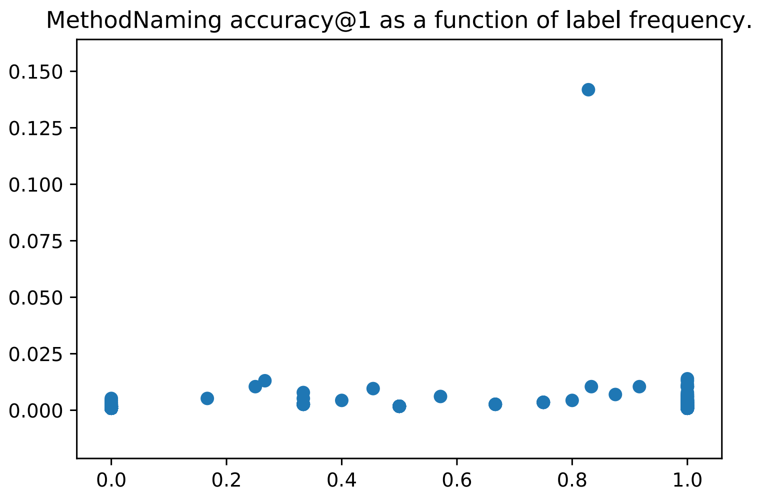


Figure B.6 – Limited correlation between frequency and accuracy.

Indeed, the model is indeed incapable of generating names for labels that are out of the training vocabulary. The accuracy @1 for the 740 methods (out of over 10,000) that only appear in the test set is zero. This effect could be mitigated by framing the problem as a generative problem instead of a multi-class classification, or by using methods that usually allow better generalization to out-of-vocabulary words. These usually rely on chunking the tokens into smaller atoms: GPT-2 leveraged this insight by using the BPE encoding process. This can also be mitigated by grouping these labels into an "unknown", catch-all category which the model can propose as a label under high uncertainty. This is used to restrict the vocabulary size in works like [Allamanis et al., 2018b]. Finally, a larger dataset to train and run on, or careful considerations in its design can also mitigate the issue. This should be addressed through datasets provided in SCUBA (see Section A).

Furthermore, removing infrequent training labels from the test set provide a decent boost in accuracy. For example, removing tokens that have occurred less than 5 times in the training set (*i.e.* have a frequency of less than 0.025%) produces a significant accuracy boost, as shown in Table B.1.

The distribution of accuracies also shows some severe splits: the model is often very confident and produces very accurate results, or is unsure and produces entirely wrong predictions for this particular token. This is shown by the skewed distribution of Fig. B.7. We hypothesize that

	F1-Macro	F1-Weighted	Subtoken Accuracy @1
MARGARET			
Corpus-Sm	0.98	0.96	0.97
Corpus-Mid	0.78	0.82	0.87
Corpus-Lg	0.68	0.81	0.85

Table B.1 – Method Naming Results (Filtered)

this is due to the skewed distributions of labels: very rare tokens are difficult to predict, as has been shown in previous work [Alon et al., 2019].

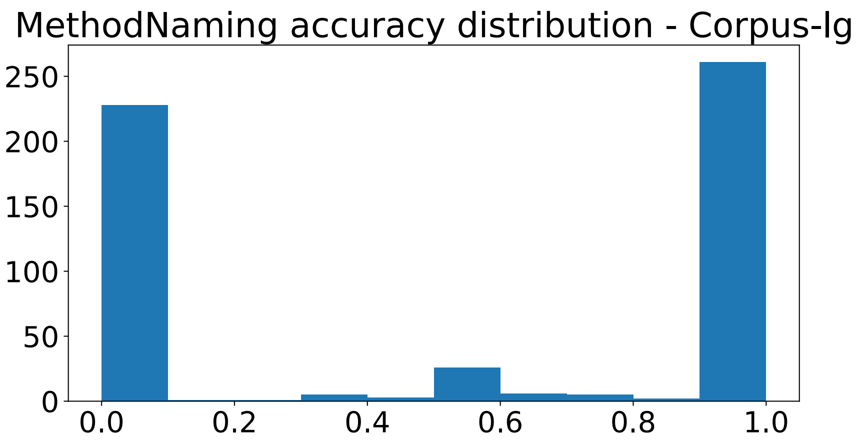


Figure B.7 – Distribution of accuracies over tokens.

Bibliography

[Abadi et al., 2015] Abadi, M. et al. (2015). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467.

[Acharya et al., 2007] Acharya, M., Xie, T., Pei, J., and Xu, J. (2007). Mining api patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 25–34, New York, NY, USA. ACM.

[Agerholm and Larsen, 1999] Agerholm, S. and Larsen, P. G. (1999). A lightweight approach to formal methods. In *Proceedings of the International Workshop on Current Trends in Applied Formal Method: Applied Formal Methods*, FM-Trends 98, pages 168–183, Berlin, Heidelberg. Springer-Verlag.

[Agrawal et al., 2018] Agrawal, M., Zitnik, M., and Leskovec, J. (2018). Large-scale analysis of disease pathways in the human interactome. In *PSB*.

[Aharoni and Goldberg, 2017] Aharoni, R. and Goldberg, Y. (2017). Towards string-to-tree neural machine translation. In *ACL*.

[Allamanis, 2018] Allamanis, M. (2018). The adverse effects of code duplication in machine learning models of code. *CoRR*, abs/1812.06469.

[Allamanis et al., 2015] Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. (2015). Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA. ACM.

[Allamanis et al., 2014] Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. A. (2014). Learning natural coding conventions. In *SIGSOFT FSE*.

[Allamanis et al., 2018a] Allamanis, M., Barr, E. T., Devanbu, P. T., and Sutton, C. A. (2018a). A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51:81:1–81:37.

[Allamanis et al., 2018b] Allamanis, M., Brockschmidt, M., and Khademi, M. (2018b). Learning to represent programs with graphs. *ICLR*.

[Allamanis et al., 2016] Allamanis, M., Peng, H., and Sutton, C. A. (2016). A convolutional attention network for extreme summarization of source code. In *ICML*.

[Allen, 1970] Allen, F. E. (1970). Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA. ACM.

[Alon et al., 2018] Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2018). A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 404–419, New York, NY, USA. ACM.

[Alon et al., 2019] Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2019). Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29.

[Ando and Zhang, 2005] Ando, R. K. and Zhang, T. (2005). A framework for learning predictive structures from multiple tasks and unlabeled data. *J. Mach. Learn. Res.*, 6:1817–1853.

[Artetxe et al., 2018] Artetxe, M., Labaka, G., Agirre, E., and Cho, K. (2018). Unsupervised neural machine translation. In *Proceedings of the Sixth International Conference on Learning Representations*.

[Backstrom and Leskovec, 2011] Backstrom, L. and Leskovec, J. (2011). Supervised random walks: predicting and recommending links in social networks. In *WSDM*.

[Bahdanau et al., 2015] Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473.

[Bastings et al., 2017] Bastings, J., Titov, I., Aziz, W., Marcheggiani, D., and Sima'an, K. (2017). Graph convolutional encoders for syntax-aware neural machine translation. In *EMNLP*.

[Battaglia et al., 2018] Battaglia, P. W. et al. (2018). Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*.

[Bavishi et al., 2018] Bavishi, R., Pradel, M., and Sen, K. (2018). Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *arXiv preprint arXiv:1809.05193*.

[Baxter, 2000] Baxter, J. (2000). A model of inductive bias learning. *J. Artif. Intell. Res.*, 12:149–198.

[Belinkov and Bisk, 2018] Belinkov, Y. and Bisk, Y. (2018). Synthetic and natural noise both break neural machine translation. In *International Conference on Learning Representations*.

[Belinkov et al., 2017] Belinkov, Y., Durrani, N., Dalvi, F., Sajjad, H., and Glass, J. R. (2017). What do neural machine translation models learn about morphology? In *ACL*.

[Ben-Nun et al., 2018] Ben-Nun, T., Jakobovits, A. S., and Hoefler, T. (2018). Neural code comprehension: A learnable representation of code semantics. In *Advances in Neural Information Processing Systems*, pages 3589–3601.

[Bengio et al., 2003] Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. (2003). A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155.

[Bhoopchand et al., 2016] Bhoopchand, A., Rocktäschel, T., Barr, E. T., and Riedel, S. (2016). Learning python code suggestion with a sparse pointer network. *CoRR*, abs/1611.08307.

[Bichsel et al., 2016] Bichsel, B., Raychev, V., Tsankov, P., and Vechev, M. (2016). Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 343–355, New York, NY, USA. ACM.

[Bielik et al., 2016] Bielik, P., Raychev, V., and Vechev, M. (2016). Phog: Probabilistic model for code. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 2933–2942. JMLR.org.

[Blevins et al., 2018] Blevins, T., Levy, O., and Zettlemoyer, L. S. (2018). Deep rnns encode soft hierarchical syntax. In *ACL*.

[Bolukbasi et al., 2016] Bolukbasi, T., Chang, K.-W., Zou, J. Y., Saligrama, V., and Kalai, A. T. (2016). Man is to computer programmer as woman is to homemaker? debiasing word embeddings. In *NIPS*.

[Bronstein et al., 2017] Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., and Vandergheynst, P. (2017). Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42.

[Brown et al., 1993] Brown, P. F., Pietra, V. J. D., Pietra, S. A. D., and Mercer, R. L. (1993). The mathematics of statistical machine translation: Parameter estimation. *Comput. Linguist.*, 19(2):263–311.

[Bruch et al., 2009] Bruch, M., Bodden, E., Monperrus, M., and Mezini, M. (2009). {IDE} 2.0: Collective intelligence in software development. In *Proceedings of the 2010 FSE/SDP Workshop on the Future of Software Engineering Research.*

[Bruna et al., 2013] Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. (2013). Spectral networks and locally connected networks on graphs. *CoRR*, abs/1312.6203.

[Calcagno and Distefano, 2011] Calcagno, C. and Distefano, D. (2011). Infer: An automatic program verifier for memory safety of c programs. In Bobaru, M., Havelund, K., Holzmann, G. J., and Joshi, R., editors, *NASA Formal Methods*, pages 459–465, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Caliskan et al., 2017] Caliskan, A., Bryson, J. J., and Narayanan, A. (2017). Semantics derived automatically from language corpora contain human-like biases. *Science*, 356(6334):183–186.

[Caruana, 1997] Caruana, R. (1997). Multitask learning. *Machine Learning*, 28(1):41–75.

[Chen et al., 2018] Chen, J., Ma, T., and Xiao, C. (2018). Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*.

[Collins, 1997] Collins, M. (1997). Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*, ACL '98/EACL '98, pages 16–23, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Collobert and Weston, 2008] Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, New York, NY, USA. ACM.

[Conneau et al., 2017] Conneau, A., Kiela, D., Schwenk, H., Barrault, L., and Bordes, A. (2017). Supervised learning of universal sentence representations from natural language inference data. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 670–680.

[Conneau et al., 2018] Conneau, A., Kruszewski, G., Lample, G., Barrault, L., and Baroni, M. (2018). What you can cram into a single
&!#* vector: Probing sentence embeddings for linguistic properties. In *ACL 2018-56th Annual Meeting of the Association for Computational Linguistics*, volume 1, pages 2126–2136. Association for Computational Linguistics.

[Dai and Le, 2015] Dai, A. M. and Le, Q. V. (2015). Semi-supervised sequence learning. In *NIPS*.

[Dai et al., 2016a] Dai, H., Dai, B., and Song, L. (2016a). Discriminative embeddings of latent variable models for structured data. In *ICML*.

[Dai et al., 2016b] Dai, H., Dai, B., and Song, L. (2016b). Discriminative embeddings of latent variable models for structured data. In *ICML*.

[Dai et al., 2019] Dai, Z., Yang, Z., Yang, Y., Cohen, W. W., Carbonell, J., Le, Q. V., and Salakhutdinov, R. (2019). Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

[Dam et al., 2016] Dam, H. K., Tran, T., and Pham, T. T. M. (2016). A deep language model for software code. In *FSE 2016: Proceedings of the Foundations Software Engineering International Symposium*, pages 1–4. [The Conference].

[Deerwester et al., 1990] Deerwester, S. C., Dumais, S. T., Landauer, T. K., Furnas, G. W., and Harshman, R. A. (1990). Indexing by latent semantic analysis. *JASIS*, 41:391–407.

[Defferrard et al., 2016] Defferrard, M., Bresson, X., and Vandergheynst, P. (2016). Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*.

[Deissenbock and Pizka, 2005] Deissenbock, F. and Pizka, M. (2005). Concise and consistent naming [software system identifier naming]. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 97–106.

[Deng et al., 2009] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

[Devlin et al., 2018] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

[Duvenaud et al., 2015] Duvenaud, D. et al. (2015). Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*.

[Erhan et al., 2010] Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., and Bengio, S. (2010). Why does unsupervised pre-training help deep learning? *J. Mach. Learn. Res.*, 11:625–660.

[Eriguchi et al., 2016] Eriguchi, A., Hashimoto, K., and Tsuruoka, Y. (2016). Character-based decoding in tree-to-sequence attention-based neural machine translation. In *WAT@COLING*.

[Eriguchi et al., 2017] Eriguchi, A., Tsuruoka, Y., and Cho, K. (2017). Learning to parse and translate improves neural machine translation. In *ACL*.

[Fellbaum, 1998] Fellbaum, C. (1998). A semantic network of english: The mother of all wordnets. *Computers and the Humanities*, 32(2):209–220.

[Firth, 1957] Firth, J. R. (1957). A synopsis of linguistic theory 1930-55. *Studies in Linguistic Analysis (special volume of the Philological Society)*, 1952-59:1–32.

[Furnas et al., 1983] Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T. (1983). Human factors and behavioral science: Statistical semantics: Analysis of the potential performance of key-word information systems. *The Bell System Technical Journal*, 62(6):1753–1806.

[Futrell and Levy, 2019] Futrell, R. and Levy, R. P. (2019). Do rnns learn human-like abstract word order preferences? *Proceedings of the Society for Computation in Linguistics (SCiL) 2019*, pages 50–59.

[Gabel and Su, 2010] Gabel, M. and Su, Z. (2010). A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 147–156, New York, NY, USA. ACM.

[Goldberg, 2019] Goldberg, Y. (2019). Assessing bert's syntactic abilities. *arXiv:1901.05287*.

[Gori et al., 2005] Gori, M., Monfardini, G., and Scarselli, F. (2005). A new model for learning in graph domains. *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, 2:729–734 vol. 2.

[Graves et al., 2016] Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., et al. (2016). Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471.

[Grover and Leskovec, 2016] Grover, A. and Leskovec, J. (2016). node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM.

[Gu et al., 2016] Gu, X., Zhang, H., Zhang, D., and Kim, S. (2016). Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM.

[Gulordava et al., 2018] Gulordava, K., Bojanowski, P., Grave, E., Linzen, T., and Baroni, M. (2018). Colorless green recurrent networks dream hierarchically. In *NAACL-HLT*.

[Gulwani and Jojic, 2007] Gulwani, S. and Jojic, N. (2007). Program verification as probabilistic inference. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 277–289, New York, NY, USA. ACM.

[Halpern et al., 2001] Halpern, J. Y., Harper, R., Immerman, N., Kolaitis, P. G., Vardi, M. Y., and Vianu, V. (2001). On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic*, 7(2):213–236.

[Hamilton et al., 2017] Hamilton, W. L., Ying, Z., and Leskovec, J. (2017). Inductive representation learning on large graphs. In *NIPS*.

[Harris, 1954] Harris, Z. S. (1954). Distributional structure. *WORD*, 10(2-3):146–162.

[Hellendoorn et al., 2018] Hellendoorn, V. J., Bird, C., Barr, E. T., and Allamanis, M. (2018). Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 152–162, New York, NY, USA. ACM.

[Hendrycks and Gimpel, 2016] Hendrycks, D. and Gimpel, K. (2016). Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *arXiv preprint arXiv:1606.08415*.

[Hindle et al., 2012] Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. (2012). On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA. IEEE Press.

[Hochreiter et al., 2001] Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., et al. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

[Houlsby et al., 2019] Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., de Laroussilhe, Q., Gesmundo, A., Attariyan, M., and Gelly, S. (2019). Parameter-efficient transfer learning for nlp. *arXiv:1902.00751*.

[Hu et al., 2018] Hu, X., Li, G., Xia, X., Lo, D., and Jin, Z. (2018). Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 200–210, New York, NY, USA. ACM.

[Hutchins et al., 1955] Hutchins, W. J., Dostert, L., and Garvin, P. (1955). The georgetown-i.b.m. experiment. In *In*, pages 124–135. John Wiley & Sons.

[Iyer et al., 2016] Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. (2016). Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083. Association for Computational Linguistics.

[Iyyer et al., 2018] Iyyer, M., Wieting, J., Gimpel, K., and Zettlemoyer, L. S. (2018). Adversarial example generation with syntactically controlled paraphrase networks. In *NAACL-HLT*.

[Jain and Wallace, 2019] Jain, S. and Wallace, B. C. (2019). Attention is not explanation. *CoRR*, abs/1902.10186.

[Jernite et al., 2015] Jernite, Y., Rush, A. M., and Sontag, D. (2015). A fast variational approach for learning markov random field language models. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 2209–2216. JMLR.org.

[Jia and Liang, 2017] Jia, R. and Liang, P. (2017). Adversarial examples for evaluating reading comprehension systems. In *EMNLP*.

[Jin et al., 2017] Jin, W., Coley, C., Barzilay, R., and Jaakkola, T. (2017). Predicting organic reaction outcomes with weisfeiler-lehman network. In *NIPS*.

[Jones et al., 01] Jones, E., Oliphant, T., Peterson, P., et al. (2001–). SciPy: Open source scientific tools for Python.

[Józefowicz et al., 2016] Józefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., and Wu, Y. (2016). Exploring the limits of language modeling. *CoRR*, abs/1602.02410.

[Kersting et al., 2016] Kersting, K., Kriege, N. M., Morris, C., Mutzel, P., and Neumann, M. (2016). Benchmark data sets for graph kernels.

[Khandelwal et al., 2018] Khandelwal, U., He, H., Qi, P., and Jurafsky, D. (2018). Sharp nearby, fuzzy far away: How neural language models use context. *arXiv preprint arXiv:1805.04623*.

[Kingma and Ba, 2015] Kingma, D. and Ba, J. (2015). Adam: a method for stochastic optimization (2014). *arXiv preprint arXiv:1412.6980*, 15.

[Kipf and Welling, 2016] Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. In *ICLR*.

[Kiros et al., 2015] Kiros, R., Zhu, Y., Salakhutdinov, R., Zemel, R. S., Torralba, A., Urtasun, R., and Fidler, S. (2015). Skip-thought vectors. In *NIPS*.

[Knuth, 1984] Knuth, D. E. (1984). Literate programming. *Comput. J.*, 27(2):97–111.

[Koc et al., 2017] Koc, U., Saadatpanah, P., Foster, J. S., and Porter, A. A. (2017). Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2017, pages 35–42, New York, NY, USA. ACM.

[Kremenek et al., 2007] Kremenek, T., Ng, A. Y., and Engler, D. R. (2007). A factor graph model for software bug finding. In *IJCAI*.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[Kudo, 2018] Kudo, T. (2018). Subword regularization: Improving neural network translation models with multiple subword candidates. *arXiv preprint arXiv:1804.10959*.

[Lample et al., 2018] Lample, G., Conneau, A., Ranzato, M., Denoyer, L., and Jégou, H. (2018). Word translation without parallel data. In *International Conference on Learning Representations*.

[Landin, 1966] Landin, P. J. (1966). The next 700 programming languages. *Commun. ACM*, 9(3):157–166.

[Levy and Goldberg, 2014a] Levy, O. and Goldberg, Y. (2014a). Linguistic regularities in sparse and explicit word representations. In *Proceedings of the Eighteenth Conference on Computational Natural Language Learning*, pages 171–180. Association for Computational Linguistics.

[Levy and Goldberg, 2014b] Levy, O. and Goldberg, Y. (2014b). Neural word embedding as implicit matrix factorization. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 2177–2185, Cambridge, MA, USA. MIT Press.

[Li et al., 2017] Li, J., Xiong, D., Tu, Z., Zhu, M., Zhang, M., and Zhou, G. (2017). Modeling source syntax for neural machine translation. In *ACL*.

[Li et al., 2016] Li, Y., Zemel, R., and Brockschmidt, M. a. (2016). Gated graph sequence neural networks. In *Proceedings of ICLR'16*.

[Liblit et al., 2006] Liblit, B., Begel, A., and Sweetser, E. (2006). Cognitive perspectives on the role of naming in computer programs. In *PPIG*.

[Linzen et al., 2016] Linzen, T., Dupoux, E., and Goldberg, Y. (2016). Assessing the ability of lstms to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics*, 4:521–535.

[Linzen and Leonard, 2018] Linzen, T. and Leonard, B. (2018). Distinct patterns of syntactic agreement errors in recurrent networks and humans. *CoRR*, abs/1807.06882.

[Liu et al., 2019] Liu, X., He, P., Chen, W., and Gao, J. (2019). Multi-task deep neural networks for natural language understanding. *CoRR*, abs/1901.11504.

[Livshits and Lam, 2005] Livshits, V. B. and Lam, M. S. (2005). Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, Berkeley, CA, USA. USENIX Association.

[Lopes et al., 2017] Lopes, C. V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., Sajnani, H., and Vitek, J. (2017). Déjàvu: A map of code duplicates on github. *Proc. ACM Program. Lang.*, 1(OOPSLA):84:1–84:28.

[Luong et al., 2015] Luong, M.-T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

[Maddison and Tarlow, 2014] Maddison, C. J. and Tarlow, D. (2014). Structured generative models of natural source code. In *ICML*.

[Marcheggiani et al., 2018] Marcheggiani, D., Bastings, J., and Titov, I. (2018). Exploiting semantics in neural machine translation with graph convolutional networks. In *NAACL-HLT*.

[Marcus et al., 1994] Marcus, M., Kim, G., Marcinkiewicz, M. A., MacIntyre, R., Bies, A., Ferguson, M., Katz, K., and Schasberger, B. (1994). The penn treebank: Annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology*, HLT '94, pages 114–119, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Markov, 1913] Markov, A. A. (1913). Essai d'une recherche statistique sur le texte du roman "Eugene Onegin" illustrant la liaison des epreuve en chain ('Example of a statistical investigation of the text of "Eugene Onegin" illustrating the dependence between samples in chain'). *Izvistia Imperatorskoi Akademii Nauk (Bulletin de l'Académie Impériale des Sciences de St.-Pétersbourg)*, 7:153–162. English translation by Morris Halle, 1956.

[McCann et al., 2018] McCann, B., Keskar, N. S., Xiong, C., and Socher, R. (2018). The natural language decathlon: Multitask learning as question answering. *arXiv preprint arXiv:1806.08730*.

[McCarthy, 1960] McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195.

[McFee and Lanckriet, 2011] McFee, B. and Lanckriet, G. (2011). Learning multi-modal similarity. *Journal of machine learning research*, 12(Feb):491–523.

[Melis et al., 2018] Melis, G., Dyer, C., and Blunsom, P. (2018). On the state of the art of evaluation in neural language models. In *International Conference on Learning Representations*.

[Merity et al., 2018] Merity, S., Keskar, N. S., and Socher, R. (2018). Regularizing and optimizing LSTM language models. In *International Conference on Learning Representations*.

[Mikolov et al., 2013] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *Proceedings of the 25th International Conference on Learning Representations*.

[Mnih and Teh, 2012] Mnih, A. and Teh, Y. W. (2012). A fast and simple algorithm for training neural probabilistic language models. In *ICML*.

[Mou et al., 2016] Mou, L., Li, G., Zhang, L., Wang, T., and Jin, Z. (2016). Convolutional neural networks over tree structures for programming language processing. In *AAAI*.

[Neamtiu et al., 2005] Neamtiu, I., Foster, J. S., and Hicks, M. (2005). Understanding source code evolution using abstract syntax tree matching. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5.

[Neelakantan et al., 2014] Neelakantan, A., Shankar, J., Passos, A., and McCallum, A. (2014). Efficient non-parametric estimation of multiple embeddings per word in vector space. In *EMNLP*.

[Neubig, 2016] Neubig, G. (2016). Survey of methods to generate natural language from source code.

[Nguyen et al., 2017] Nguyen, T. D., Nguyen, A. T., Phan, H. D., and Nguyen, T. N. (2017). Exploring api embedding for api usages and applications. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 438–449, Piscataway, NJ, USA. IEEE Press.

[Nguyen et al., 2016] Nguyen, T. T., Pham, H. V., Vu, P. M., and Nguyen, T. T. (2016). Learning api usages from bytecode: A statistical approach. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 416–427, New York, NY, USA. ACM.

[Nickel and Kiela, 2017] Nickel, M. and Kiela, D. (2017). Poincaré embeddings for learning hierarchical representations. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 6338–6347. Curran Associates, Inc.

[Oda et al., 2015] Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T., and Nakamura, S. (2015). Learning to generate pseudo-code from source code using statistical machine translation. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584, Lincoln, Nebraska, USA.

[Pennington et al., 2014] Pennington, J., Socher, R., and Manning, C. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543. Association for Computational Linguistics.

[Peters et al., 2017] Peters, M. E., Ammar, W., Bhagavatula, C., and Power, R. (2017). Semi-supervised sequence tagging with bidirectional language models. *arXiv preprint arXiv:1705.00108*.

[Peters et al., 2018a] Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. S. (2018a). Deep contextualized word representations. In *NAACL-HLT*.

[Peters et al., 2018b] Peters, M. E., Neumann, M., Zettlemoyer, L. S., and tau Yih, W. (2018b). Dissecting contextual word embeddings: Architecture and representation. In *EMNLP*.

[Piech et al., 2015] Piech, C., Huang, J., Nguyen, A., Phulsuksombati, M., Sahami, M., and Guibas, L. (2015). Learning program embeddings to propagate feedback on student code. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1093–1102, Lille, France. PMLR.

[Pu et al., 2016] Pu, Y., Narasimhan, K., Solar-Lezama, A., and Barzilay, R. (2016). sk_p: a neural program corrector for moocs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 39–40. ACM.

[Radford et al., 2018] Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training. *OpenAI*.

[Radford et al., 2019] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI*.

[Raychev et al., 2015] Raychev, V., Vechev, M., and Krause, A. (2015). Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA. ACM.

[Raychev et al., 2014] Raychev, V., Vechev, M., and Yahav, E. (2014). Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA. ACM.

[Ruder and Howard, 2018] Ruder, S. and Howard, J. (2018). Universal language model fine-tuning for text classification. In *ACL*.

[Salton et al., 1975] Salton, G., Wong, A., and Yang, C. (1975). A vector space model for automatic indexing. *Commun. ACM*, 18:613–.

[Scarselli et al., 2009] Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, 20:61–80.

[Schlichtkrull et al., 2018] Schlichtkrull, M., Kipf, T., Bloem, P., van den Berg, R., Titov, I., and Welling, M. (2018). Modeling relational data with graph convolutional networks. In *ESWC*.

[Schomburg et al., 2004] Schomburg, I., Chang, A., Ebeling, C., Gremse, M., Heldt, C., Huhn, G., and Schomburg, D. (2004). Brenda, the enzyme database: updates and major new developments. *Nucleic acids research*, 32 Database issue:D431–3.

[Schwenk and Douze, 2017] Schwenk, H. and Douze, M. (2017). Learning joint multilingual sentence representations with neural machine translation. In *Proceedings of the 2nd Workshop on Representation Learning for NLP*, pages 157–167.

[Sennrich et al., 2016] Sennrich, R., Haddow, B., and Birch, A. (2016). Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725. Association for Computational Linguistics.

[Shannon, 1950] Shannon, C. (1950). Prediction and entropy of printed english. *Bell Systems Technical Journal*.

[Shi et al., 2016] Shi, X., Padhi, I., and Knight, K. (2016). Does string-based neural mt learn source syntax? In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1526–1534. Association for Computational Linguistics.

[Si et al., 2018] Si, X., Dai, H., Raghothaman, M., Naik, M., and Song, L. (2018). Learning loop invariants for program verification. In *Advances in Neural Information Processing Systems*, pages 7762–7773.

[Siegelmann and Sontag, 1992] Siegelmann, H. T. and Sontag, E. D. (1992). On the computational power of neural nets. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT ’92, pages 440–449, New York, NY, USA. ACM.

[Stickland and Murray, 2019] Stickland, A. C. and Murray, I. (2019). Bert and pals: Projected attention layers for efficient adaptation in multi-task learning. *CoRR*, abs/1902.02671.

[Sundermeyer et al., 2012] Sundermeyer, M., Schlüter, R., and Ney, H. (2012). Lstm neural networks for language modeling. In *INTERSPEECH*.

[Suter et al., 2011] Suter, P., Köksal, A. S., and Kuncak, V. (2011). Satisfiability modulo recursive programs. In Yahav, E., editor, *Static Analysis*, pages 298–315, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Takang et al., 1996] Takang, A. A., Grubb, P. A., and Macredie, R. D. (1996). The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4:143–167.

[Taylor, 1953] Taylor, W. L. (1953). "cloze procedure": A new tool for measuring readability. *Journalism Bulletin*, 30(4):415–433.

[Tenney et al., 2019] Tenney, I., Xia, P., Chen, B., Wang, A., Poliak, A., McCoy, R. T., Kim, N., Durme, B. V., Bowman, S., Das, D., and Pavlick, E. (2019). What do you learn from context? probing for sentence structure in contextualized word representations. In *International Conference on Learning Representations*.

[Tufano et al., 2018] Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., and Poshyvanyk, D. (2018). Deep learning similarities from different representations of source code. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 542–553. IEEE.

[Turian et al., 2010] Turian, J., Ratinov, L., and Bengio, Y. (2010). Word representations: A simple and general method for semi-supervised learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pages 384–394, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Vashishth et al., 2018] Vashishth, S., Yadav, P., Bhandari, M., Rai, P., Bhattacharyya, C., and Talukdar, P. (2018). Graph convolutional networks based word embeddings. *CoRR*, abs/1809.04283.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *NIPS*.

[Vechev and Yahav, 2016] Vechev, M. and Yahav, E. (2016). Programming with "big code". *Found. Trends Program. Lang.*, 3(4):231–284.

[Velickovic et al., 2018] Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. (2018). Graph attention networks. In *International Conference on Learning Representations*.

[Wang and Cho, 2019] Wang, A. and Cho, K. (2019). Bert has a mouth, and it must speak: Bert as a markov random field language model. *arXiv pre-print*, 1902.04094.

[Wang et al., 2018a] Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. (2018a). Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

[Wang et al., 2017] Wang, M., Tang, Y., Wang, J., and Deng, J. (2017). Premise selection for theorem proving by deep graph embedding. In *NIPS*.

[Wang et al., 2016] Wang, S., Chollak, D., Movshovitz-Attias, D., and Tan, L. (2016). Bugram: Bug detection with n-gram language models. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 708–719.

[Wang et al., 2018b] Wang, X., Girshick, R., Gupta, A., and He, K. (2018b). Non-local neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7794–7803.

[Weaver, 1955] Weaver, W. (1955). Translation. *Machine translation of languages*, 14:15–23.

[White et al., 2015] White, M., Vendome, C., Linares-Vásquez, M., and Poshyvanyk, D. (2015). Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 334–345, Piscataway, NJ, USA. IEEE Press.

[Williams and Hollingsworth, 2005] Williams, C. C. and Hollingsworth, J. K. (2005). Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480.

[Winn et al., 2005] Winn, J., Criminisi, A., and Minka, T. (2005). Object categorization by learned universal visual dictionary. In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, volume 2, pages 1800–1807 Vol. 2.

[Wu et al., 2016] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

[Xu et al., 2019] Xu, K., Hu, W., Leskovec, J., and Jegelka, S. (2019). How powerful are graph neural networks? In *International Conference on Learning Representations*.

[Yamada and Knight, 2001] Yamada, K. and Knight, K. (2001). A syntax-based statistical translation model. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, ACL '01, pages 523–530, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Yanardag and Vishwanathan, 2015] Yanardag, P. and Vishwanathan, S. (2015). Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1365–1374. ACM.

[Yin et al., 2018a] Yin, P., Deng, B., Chen, E., Vasilescu, B., and Neubig, G. (2018a). Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories (MSR)*, Gothenburg, Sweden.

[Yin et al., 2018b] Yin, P., Deng, B., Chen, E., Vasilescu, B., and Neubig, G. (2018b). Learning to mine parallel natural language/source code corpora from stack overflow. In *40th International Conference on Software Engineering (ICSE) Poster Track*, Gothenberg, Sweden.

[Yin et al., 2019] Yin, P., Neubig, G., Allamanis, M., Brockschmidt, M., and Gaunt, A. L. (2019). Learning to represent edits. In *International Conference on Learning Representations (ICLR)*, New Orleans, LA, USA.

[Ying et al., 2018a] Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W., and Leskovec, J. (2018a). Graph convolutional neural networks for web-scale recommender systems. In *KDD*.

[Ying et al., 2018b] Ying, Z., You, J., Morris, C., Ren, X., Hamilton, W., and Leskovec, J. (2018b). Hierarchical graph representation learning with differentiable pooling. In *NeurIPS*.

[Zaremba and Sutskever, 2014] Zaremba, W. and Sutskever, I. (2014). Learning to execute. *arXiv preprint arXiv:1410.4615*.

[Zhang and Bowman, 2018] Zhang, K. W. and Bowman, S. R. (2018). Language modeling teaches you more syntax than translation does: Lessons learned through auxiliary task analysis. *CoRR*, abs/1809.10040.

[Zhang and Chen, 2018] Zhang, M. and Chen, Y. (2018). Link prediction based on graph neural networks. In *NIPS*.

[Zhang et al., 2018] Zhang, Y., Liu, Q., and Song, L. (2018). Sentence-state lstm for text representation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 317–327.

[Zimmermann et al., 2004] Zimmermann, T., Weisgerber, P., Diehl, S., and Zeller, A. (2004). Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA. IEEE Computer Society.

[Zitnik et al., 2018] Zitnik, M., Agrawal, M., and Leskovec, J. (2018). Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics*, 34.