

Université de Montréal

Contextual Cues for Deep Learning Models of Code

par

Disha Shrivastava

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en Discipline

September 9, 2023

Université de Montréal

Faculté des arts et des sciences

Cette thèse intitulée

Contextual Cues for Deep Learning Models of Code

présentée par

Disha Shrivastava

a été évaluée par un jury composé des personnes suivantes :

Aishwarya Agrawal

(président-rapporteur)

Hugo Larochelle

(directeur de recherche)

Daniel Tarlow

(codirecteur)

Laurent Charlin

(membre du jury)

Baishakhi Ray

(examineur externe)

Aishwarya Agrawal

(représentant du doyen de la FESP)

Sommaire

Le code source offre un domaine d’application passionnant des méthodes d’apprentissage en profondeur, englobant des tâches telles que la synthèse, la réparation et l’analyse de programmes, ainsi que des tâches à l’intersection du code et du langage naturel. Bien que les modèles d’apprentissage profond pour le code, en particulier les grands modèles de langage, aient récemment connu un succès significatif, ils peuvent avoir du mal à se généraliser à du code invisible. Cela peut conduire à des inexactitudes, en particulier lorsque vous travaillez avec des référentiels contenant des logiciels propriétaires ou du code en cours de travail.

L’objectif principal de cette thèse est d’exploiter efficacement les signaux utiles du contexte disponible afin d’améliorer les performances des modèles de code d’apprentissage profond pour une tâche donnée. En incorporant ces indices contextuels, les capacités de généralisation du modèle sont amplifiées, fournissant des informations supplémentaires non évidentes à partir de l’entrée d’origine et orientant son attention vers des détails essentiels. De plus, l’utilisation d’indices contextuels facilite l’adaptation aux nouvelles tâches et améliore les performances des tâches existantes en effectuant des prédictions plus contextuelles. Pour y parvenir, nous présentons un cadre général comprenant deux étapes : (a) l’amélioration du contexte, qui implique l’enrichissement de l’entrée avec un contexte de support obtenu grâce à l’identification et à la sélection d’indices contextuels pertinents, et (b) la prédiction à l’aide du contexte amélioré, où nous exploitons le contexte de support combiné aux entrées pour faire des prédictions précises. La thèse présente quatre articles qui proposent diverses approches pour ces étapes.

Le premier article divise le problème standard de la programmation par exemples en deux étapes : (a) trouver des programmes qui satisfont des exemples individuels (solutions par exemple) et, (b) combiner ces solutions par exemple en tirant parti de leurs états d’exécution de programme pour trouver un programme qui satisfait tous les exemples donnés.

Le deuxième article propose une approche pour sélectionner des informations ciblées à partir du fichier actuel et les utiliser pour adapter le modèle de complétion de code à un contexte local jamais vu précédemment.

Le troisième article s’appuie sur le deuxième article en tirant parti des indices contextuels de l’ensemble du répertoire de code à l’aide d’un ensemble de requêtes (*prompts*) proposées

suggérant l'emplacement et le contenu du contexte particulièrement utile à extraire du répertoire. Nous proposons un cadre pour sélectionner la requête la plus pertinente, qui est ensuite utilisée pour demander à un modèle de langage de code de générer des prédictions pour le reste de la ligne de code suivant un curseur positionné dans un fichier.

Le quatrième article prolonge le troisième article en proposant un cadre qui apprend à combiner plusieurs contextes divers à partir du répertoire. Nous montrons que la formation de modèles de langage de code plus petits de cette manière fonctionne mieux ou à égalité avec des modèles beaucoup plus grands qui n'utilisent pas le contexte du répertoire de code.

Mots-clés: Apprentissage profond, synthèse de programmes, complétion de code, apprentissage automatique pour le code, génie logiciel, recherche d'informations, grands modèles de langage.

Summary

Source code provides an exciting application area of deep learning methods, encompassing tasks like program synthesis, repair, and analysis, as well as tasks at the intersection of code and natural language. Although deep learning models for code, particularly large language models, have recently seen significant success, they can face challenges in generalizing to unseen code. This can lead to inaccuracies especially when working with repositories that contain proprietary software or work-in-progress code.

The main focus of this thesis is to effectively harness useful signals from the available context such that it can improve the performance of the deep learning models of code at the given task. By incorporating these contextual cues, the model’s generalization capabilities are amplified, providing additional insights not evident from the original input and directing its focus toward essential details. Furthermore, the use of contextual cues aids in adapting to new tasks and boosts performance on existing ones by making more context-aware predictions. To achieve this, we present a general framework comprising two stages: (a) Context Enhancement, which involves enriching the input with support context obtained through the identification and selection of relevant contextual cues, and (b) Prediction using the Enhanced Context, where we leverage the support context combined with the input to make accurate predictions. The thesis presents four articles that propose diverse approaches for these stages.

The first article breaks the standard problem of programming by examples into two stages: (a) finding programs that satisfy individual examples (per-example solutions) and, (b) combining these per-example solutions by leveraging their program execution states to find a program that satisfies all given examples.

The second article proposes an approach for selecting targeted information from the current file and using it to adapt the code completion model to an unseen, local context.

The third article builds upon the second article by leveraging contextual cues from the entire code repository using a set of prompt proposals that govern the location and content of the context that should be taken from the repository. We propose a framework to select the most relevant prompt proposal context which is then used to prompt a large language

model of code to generate predictions for the tokens in the rest of the line following the cursor in a file.

The fourth article extends the third article by proposing a framework that learns to combine multiple diverse contexts from the repository. We show that training smaller models of code this way performs better or at par with significantly larger models that are not trained with repository context.

Keywords: Deep Learning, Program Synthesis, Code Completion, Machine Learning for Code, Software Engineering, Information Retrieval, Large Language Models.

Contents

Sommaire	iii
Summary	v
List of tables	xi
List of figures	xiii
List of Acronyms and Abbreviations	xvi
Acknowledgements	xviii
Chapter 1. Introduction	1
1.1. Enhance-Predict: Our General Framework	3
1.2. Thesis Overview	4
1.3. List of Excluded Contributions	6
Chapter 2. Background	8
2.1. Uniqueness of Code	8
2.2. Models of Code	9
2.2.1. Code-Generating Models	9
2.2.2. Representational Models	10
2.2.3. Pattern Mining Models	11
2.3. Applications	11
2.3.1. Improving Programming Experience or Front-End	11
2.3.2. Improving Programming Tools or Back-End	11
2.3.3. Program Synthesis	12
2.4. Large Language Models of Code	13
2.4.1. Transformers	13
2.4.2. Pretraining	15

2.4.3. Prompting	16
Chapter 3. Prologue to the first article	18
3.1. Article Details	18
3.2. Context	18
3.3. Contributions	19
3.4. Research Impact	19
Chapter 4. Learning to Combine Per-Example Solutions for Neural Program Synthesis	21
4.1. Introduction	21
4.2. Background	23
4.2.1. PCCoder	23
4.3. Neural Per-Example Program Synthesis (N-PEPS)	25
4.3.1. Per Example Program Synthesis	25
4.3.2. Cross Aggregator	26
4.3.3. Training	28
4.3.4. Inference	29
4.4. Experiments and Results	30
4.4.1. Initial Experiment: Analysis of PE Solutions	30
4.4.2. Methods	31
4.4.3. Results	32
4.5. Related Work	35
4.6. Conclusions and Future Directions	36
Appendix for the first article	38
Chapter 5. Prologue to the second article	59
5.1. Article Details	59
5.2. Context	59
5.3. Contributions	60
5.4. Research Impact	60

Chapter 6. On-the-Fly Adaptation of Source Code Models	61
6.1. Introduction	61
6.2. Related Work	63
6.3. Methodology	64
6.3.1. Line-level Maintenance	64
6.3.2. Adaptation	64
6.4. Experiments and Results	65
6.4.1. Experimental Details	65
6.4.2. Evaluation Setup	66
6.4.3. Results	66
6.5. Conclusions	69
Appendix for the second article	70
Chapter 7. Prologue to the third article	75
7.1. Article Details	75
7.2. Context	75
7.3. Contributions	76
7.4. Research Impact	76
Chapter 8. Repository-Level Prompt Generation for Large Language Models of Code	77
8.1. Introduction	77
8.2. Repo-Level Prompt Generator (RLPG)	80
8.2.1. Repo-Level Prompt Proposals	80
8.2.2. Prompt Proposal Classifier (PPC)	82
8.2.3. Prompt Composer	83
8.3. Experiments and Results	84
8.3.1. Dataset Creation	84
8.3.2. Experimental Details	85
8.3.3. Results	87
8.4. Related Work	90

8.5. Discussion	92
Appendix for the third article	93
Chapter 9. Prologue to the fourth article.....	107
9.1. Article Details	107
9.2. Context.....	107
9.3. Contributions.....	108
9.4. Research Impact	108
Chapter 10. RepoFusion: Training Code Models to Understand Your Repository	109
10.1. Introduction	109
10.2. Training with Repository Context.....	111
10.2.1. Fusion-in-Decoder	111
10.2.2. Repository Contexts	111
10.2.3. RepoFusion	112
10.3. Experiments and Results.....	114
10.3.1. Dataset Creation	114
10.3.2. Experimental Details	115
10.3.3. Results	117
10.4. Related Work.....	121
10.5. Discussion	121
Appendix for the fourth article	123
Chapter 11. Conclusion	127
11.1. Summary of Contributions.....	127
11.2. Broad Applicability of Our Framework.....	129
11.3. Going Forward.....	131
11.3.1. Modeling the Code Ecosystem	131
11.3.2. Modeling User Interactions	131
References	133

List of tables

1	Summary of how the articles presented in the thesis fit within our general framework. PE, CA, RLPG, and TSSA denote Per-Example, Cross-Aggregator, Repo-Level Prompt Generator, Targeted Support Set Adaptation.....	5
4.1	Success ratio of GPS, ind and tot for different values of k for test programs of length 4.....	31
4.2	Results for E2: Success Ratio with standard error for all models.....	33
4.3	Performance of variants of K for E1.....	35
4.4	Aggregator data statistics for E1 and E2.....	45
4.5	Hyperparameter values for training the CA. lr= learning rate, lrs = learning rate scheduler, o=optimizer.....	47
4.6	Hyperparameter values for Inference. PT = PEPS timeout, GT = 5 - (5 * PT).	48
4.7	Variation in success ratio for runs across the same machine (run1, run2, run3), different machines (M1, M2, M3) and different test splits (split-1, split-2) for E1	48
4.8	Success Ratio with standard error for key variants for E2.....	49
4.9	Success Ratio with standard error for intent generalization experiments	50
6.1	Performance on hole target prediction on test data in terms of token cross-entropy, MRR@10 and Recall@10. We also report 95% confidence intervals for each entry. We highlight the best performing models (in terms of mean) for each column....	67
6.2	Comparison of cross-entropy on prediction of identifiers and literals for TSSA-16 vs. a non-adaptation model.	67
6.3	Corpus Statistics for 1% split of the dataset. M indicates numbers in millions....	71
6.4	Notation for terms occurring in Table 6.5	72
6.5	Best hyperparameter values for all our settings.	72
6.6	Description of Java token-types given by Python’s Java-parser into broad token categories for ease of visualization.	72

8.1	Statistics of our dataset.	84
8.2	Performance of the oracle relative to Codex.	87
8.3	Success Rate (SR) of different methods on the test data when averaged across all holes.	88
8.4	Edit distance based performance evaluation.	89
8.5	Success Rate (SR) with code-cushman-001.	90
8.6	Selecting files for a prompt source.	95
8.7	List of our proposed repo-level prompt proposals.	97
8.8	Hole-wise and Repo-wise Success Rate (SR) of different methods on the test data.	100
8.9	Success Rate (SR) when taking different versions of post lines.	102
8.10	Success Rate (SR) of different compositions of the prompt proposals on the test set.	103
8.11	Success Rate (SR) of Codex and oracle over the test set when the total context length = 2048.	103
8.12	Percentage truncation when using different contexts in the prompt.	104
8.13	Success Rate of different methods on training data.	104
8.14	Success Rate of different methods on validation data.	105
8.15	Success Rate of different methods on test data.	105
10.1	Statistics of Stack-Repo.	115
10.2	Completion success rate on the test set for different methods.	118
10.3	Comparison with StarCoderBase on a test set subset.	118
10.4	Completion success rate when initialized from a pretrained vs finetuned model.	120
10.5	Completion success rate with repeating different types of PPCs multiple times.	125
10.6	Completion success rate with and without appending surrounding context.	126
10.7	Completion success rate on the test set for pretrained CodeT5.	126

List of figures

1	This figure taken from Vaswani et al. (2017a) represents the architecture of a Transformer encoder-decoder model.	14
4.1	Figure explaining the idea of N-PEPS: (<i>Left</i>) Illustrating the two stages of N-PEPS with an example; (<i>Right</i>) Synthesizing line 2 of p_g using contributions from CA and GPS, with details of how query, keys, values and relation scores are formed. White box shows an example of obtaining a PE state embedding.	22
4.2	Figure explaining the idea of PCCoder: (<i>Left</i>) Sample program along with two IO examples that forms the program state at $t = 0$; (<i>Right</i>) Block Diagram explaining the training of PCCoder at line 2 of the program.	24
4.3	Results for E1: (<i>Top</i>) Success Ratio with standard error. The top row in the table corresponds to GPS; (<i>Bottom</i>) Success Ratio vs. time taken.	32
4.4	Visualization of attention scores for N-PEPS+ \mathcal{U}	34
4.5	Variation of success ratio with PEPS Timeout (top) and α (below) for N-PEPS.	34
4.6	Distribution of training programs: (<i>Left</i>) For E1; (<i>Right</i>) For E2.	44
4.7	Function-wise breakdown of failing cases for GPS and our N-PEPS model on E1	51
4.8	Fraction of perfect PE solutions with length of test programs for our best model for E2	51
4.9	Variation of fraction of operator overlap with length of test programs for our best model for E2: (<i>Left</i>) $\alpha < 1.0$ (CA + GPS); (<i>Right</i>) $\alpha = 1.0$ (CA alone)	53
4.10	New operators discovered = 1, Total cases = 2169.	54
4.11	New operators discovered = 2, Total cases = 1155.	55
4.12	New operators discovered = 3, Total cases = 395.	56
4.13	New operators discovered = 4, Total cases = 119.	56
4.14	New operators discovered = 5, Total cases = 15.	56

6.1	Block diagram illustrating our approach for a sample file. To predict hole target <code>StandardPropertyManager</code> using <i>hole window</i> (w^h), our model learns parameter θ_k by performing k steps of gradient update using <i>support tokens</i> (t^s) and <i>support windows</i> (w^s) in its inner loop.	62
6.2	Average hole target cross-entropy for each token-type for our TSSA-16 model.	67
6.3	Improvement due to TSSA on small capacity model ($b_{low} - m_{low}$) vs. Improvement due to big model ($b_{low} - b_{high}$)	68
6.4	Variation of token cross-entropy for val data with different definition of support tokens. (Left) With fixed number of updates; (Right) With fixed number of support tokens.	69
6.5	Variation of hole target cross-entropy values with number of updates and number of support tokens for the validation data.	73
6.6	Sample cases illustrating the benefits of TSSA on low capacity model: (Left) Hole target is string literal with partial match in support tokens; (Right) Hole target is identifier with exact match in support tokens.	74
8.1	Figure explaining the idea of Repo-Level Prompt Generator: Given a list of prompt proposals and the target hole position along with the associated repository as input, the prompt proposal classifier predicts a prompt proposal. The context from the predicted prompt proposal $p = 14$, i.e., method names and bodies from the imported file (highlighted in violet) is then combined with the default Codex context or context prior to the position of the hole in the current file (highlighted in gray) to compose a prompt. Prompting Codex with the generated prompt produces a prediction for the target hole (highlighted in dark red).	78
8.2	(Left) Variation of RLPG and Fixed Prompt Proposal with #attempts (k); (Right) Mean success rates of different prompt sources when they are applicable.	88
8.3	Mean success rate on validation data based on prompt context type when they are applicable.	100
8.4	(Top) Normalized success rate of prompt sources when applicable, (Bottom) Normalized success rate of prompt context types when applicable	101
10.1	Figure explaining the idea of RepoFusion. Given multiple relevant contexts from the repository (<i>Repo Contexts</i>), RepoFusion appends the <i>Surrounding</i>	

	<i>Context</i> (highlighted in gray) to each repo context, encodes them separately, and combines them to produce a prediction of the target hole.	110
10.2	Different strategies employed for producing repo contexts (RCs) from prompt proposal contexts (PPCs): (a) T-rank: we truncate the i -th ranked PPC to yield the i -th RC. (b) T-rand: we position the truncated i -th PPC at a random position j in RepoFusion’s sequences of RCs. (c) NT-Rank: each PPC yields as many RCs as necessary to exhaust all of its tokens without truncation. (d) NT-Prior-Last: we reserve the last r RCs for the Prior PPC and fill the rest RCs as in NT-Rank.	113
10.3	Completion success rate with different approaches to producing repository contexts (RCs). (Left) Impact of RC production and ordering strategies; (Right) Impact of different RC retrieval methods.	119
10.4	Completion success rate as a function of (Left) the length of the individual repo context l ; (Middle) the number of repo contexts N ; (Right) the number of prompt proposal contexts that were used to produce the N repo contexts.	120

List of Acronyms and Abbreviations

PE	Per-Example
CA	Cross Aggregator
GPS	Global Program Synthesis
PEPS	Per-Example Program Synthesis
N-PEPS	Neural Per-Example Program Synthesis
LLM	Large Language Model
LM	Language Model
RLPG	Repo-Level Prompt Generator
PP	Prompt Proposal
PPC	Prompt Proposal Classifier

TSSA Targeted Support Set Adaptation

DL4C Deep Learning for Code

IO Input-Output

RC Repo Context

Acknowledgements

I consider myself immensely privileged to have had the opportunity to pursue my PhD at Mila. I have been lucky to be surrounded by exceptional mentors and friends who have not only guided me in understanding how to conduct quality research but also greatly contributed to my personal development as a human being.

First and foremost, I wish to express my deepest gratitude to my advisors, Hugo Larochelle and Danny Tarlow. They have truly been the most exceptional guides any student could ask for. I vividly recall the joy and pride I felt when I received Hugo’s email stating that they had decided to accept me as a PhD student. From that point on, my learning journey has been constant and rewarding. Interactions with Hugo and Danny have not only shaped me as a researcher but also endowed me with the confidence needed for effective communication and tackling difficult problems in a scientific way. I am profoundly fortunate to have had the chance to learn from two such outstanding researchers who were consistently available throughout my PhD journey. I am immensely grateful for their belief in me and their dedicated commitment to pushing me to learn the intricacies of becoming a truly independent researcher. Thanks for asking difficult questions whenever I proposed an idea that made me realize I need to think about it more concretely. When I reflect on my journey, I see a vast difference between the Disha who received that acceptance email and the Disha who is now authoring this thesis. I observe a significant evolution in her thinking process, her approach to research problems, and her execution of ideas. The majority of the credit for this transformation is owed to Hugo and Danny, their high standards for research excellence, and their constructive feedback.

I want to thank Hugo for always making sure that I am comfortable whether it be offering to pay to install an AC in my house during the pandemic or agreeing to finance my last-minute plans to visit NeurIPS even though I didn’t have a paper. I want to thank Danny for taking out extra time from his busy schedule to have one-on-one meetings with me where we discussed the art of reading papers. Both Hugo and Danny provided me with invaluable advice beyond research, such as tips on effective time management, concise writing, peer collaboration, and fruitful research career progression. They have also significantly contributed to improving my English writing skills; Hugo painstakingly added spaces before

parentheses (a lesson I hope I have now learned), and Danny advised me on the correct usage of ‘that’ versus ‘which’. They have been a tremendous source of inspiration, and I intend to continuously work towards emulating their invaluable skills.

During my internships, I have had the chance to work alongside some extraordinary researchers. I would like to extend my deepest gratitude to Yujia Li and David Choi at DeepMind for providing me with the opportunity to work in a large team. I really appreciate their support in trusting me with the ownership of an important research direction and providing me the opportunity to present and discuss my work in weekly meetings that helped me receive valuable feedback. I am also grateful to them for their diligent code reviews, their guidance on decomposing complex objectives into feasible tasks, and their lessons on efficient methods to communicate my daily progress with the team. My sincere thanks also go to Oriol Vinyals, Nando de Freitas, Junyoung Chung, and other exceptional members of the Deep Learning team for their exceedingly positive interactions. At Google Brain, in addition to Hugo and Danny, I was lucky to be mentored by Charles Sutton, who served as a constant source of support and encouragement for me. I also had the opportunity to participate in the reading group and engage with other exceptional researchers in the Learning for Code team. This not only allowed me to stay up-to-date with the latest research in the field but also provided me with a community of researchers for valuable exchange of ideas. I would also like to thank Dzmitry Bahdanau and Torsten Scholak at ServiceNow Research for granting me complete liberty to explore and pursue my own ideas.

I would like to convey my sincere appreciation to the highly talented people, not previously mentioned, with whom I have had the privilege of co-authoring papers throughout my PhD journey. I would like to thank Denis Kocetkov, Harm de Vries, Rahul Aralikkat, Edoardo Maria Ponti, Siva Reddy, Anders Søgaard, Eeshan Gunesh Dhekane, Riashat Islam, Avinash Bhat, and Jin L.C. Guo. I wish to extend special thanks to Jin for her constant appreciation of my work and for giving me a chance to mentor her students. I also want to recognize the contributions of my peers Daniel Johnson and David Bieber. Despite not having the opportunity to collaborate directly, their impressive research work and stimulating discussions have served as a constant source of inspiration for me to pursue excellence.

I have been lucky to forge friendships with some truly wonderful individuals during the course of my PhD. Thank you Gunshi, Varsha, Sandeep, and Rithesh for the cherished memories, late-night serious discussions over ice cream, and for helping to alleviate my sense of loneliness during the pandemic. My special thanks to Gunshi for being an amazing roommate and my first friend in Montreal and to Sandeep for always being available to answer all my research career-related questions. I want to thank Sarath for generously

offering his place for my initial stay in Montreal, Ankesh for his invaluable advice throughout my PhD and for letting me borrow his website template, Sherjil for being my first point of contact for every small thing during my internship at DeepMind, and Rosemary for her contagious laughter and help with the application process for the DeepMind internship. My heartfelt thanks to Ankit for understanding and empathizing with my anxieties as a fellow PhD student and for being a constant companion during our post-lab dinners, Sai and Anirudh for their continuous kindness, Breandan and David for the memorable moments we shared while organizing the AIPLANS workshop, and Sal, Melisande, and Hattie for the truly enjoyable dinner discussions we had. To my other friends - Gaurav, Max, Dishank, Sai, Shivendra, Madhu - your presence has been instrumental in preserving my sanity and bringing joy throughout my PhD journey. I want to express my deepest appreciation to Julie, my dearest neighbor and friend, for welcoming me with open arms, caring for me as a mother, and constantly baking delicious desserts for me.

Last but certainly not least, I want to convey my heartfelt gratitude to my parents for being the initial source of inspiration for my decision to pursue a PhD and for providing me with all the resources needed to accomplish my goals. They have consistently gone above and beyond in their encouragement, standing by me through the highs and lows of my life. Their unwavering support has been instrumental in allowing me to pursue my dreams wholeheartedly. Finally, I want to express my love and appreciation to my husband Sami for being incredibly motivating and understanding. Thank you for assisting me with figures, lending a patient ear to my daily musings about my research life, and ensuring that I am well-fed and well-rested, particularly during the conference deadline period.

Chapter 1

Introduction

Machine Learning systems, especially deep learning, have shown remarkable performance on a number of tasks like generation of natural language (Brown et al., 2020b; Anil et al., 2023), images (Ramesh et al., 2022b; Saharia et al., 2022), speech (Radford et al., 2022), protein structure prediction (Jumper et al., 2021), competing in games (Silver et al., 2016) and discovering novel computational algorithms (Fawzi et al., 2022; Mankowitz et al., 2023). A promising and exciting application area of deep learning methods lies in the domain of source code, which involves tasks such as synthesis, repair, and analysis of computer programs. The scope of these methods also extends to the intersection of code with natural language, encompassing tasks like explaining code and linking code with documentation to facilitate code search. The motivation for research toward developing these methods comes from three different perspectives:

- **Helping Non-Programmers:** With the advent of technology, a rapidly growing segment of the population has access to computational devices, such as smartphones and computers. However, the percentage of people programming these devices is still low as it requires technical skills both in form of knowledge in programming languages as well as in the problem domain. Program synthesis, where the aim is to generate programs or code fragments from a user’s intent, can be used as an effective tool in this scenario. The users can express themselves in a form that is natural to them, similar in spirit to the way they command a personal assistant, thereby enabling them to solve problems in an automated fashion without the need to design and implement new algorithms. For instance, GitHub’s code-assistant Copilot¹ enables users to generate programs in unfamiliar programming languages, translate between different programming languages or even summarize code, all by simply expressing their intent via natural language instructions.

¹<https://githubnext.com/projects/copilot-view/>

- **Helping Programmers:** The techniques mentioned above can help boost the productivity of software engineers by helping them divert their attention from mundane tasks to tasks which require more creative thinking. For example, code completion tools in Integrated Development Environments (IDE) save programmers the trouble of typing lengthy and repeatable code fragments without having the need to search through the documentation.
- **Advancing ML Research:** The amalgamation of (a) domain constraints, (b) rigid syntax, (c) discrete nature of code, (d) various symbolic representation forms (e.g. abstract syntax trees, data, and control flow graphs), (e) intricate structure, (f) continually evolving nature (e.g. the introduction of new features and bug fixes), and (g) a broad spectrum of downstream tasks with diverse objectives (e.g. defect localization (Pearson et al., 2017), fuzzing (She et al., 2019), and code translation (Lachaux et al., 2020)), presents unique challenges for the development of efficient ML models.

Deep Learning for Code (DL4C) has received a fair amount of attention in the last decade (Allamanis et al., 2018a), yet arguably the recent application of large-scale language modeling techniques to the domain of code holds a tremendous promise to completely revolutionize this area (Chen et al., 2021; Austin et al., 2021; Anil et al., 2023; Fried et al., 2022; Nijkamp et al., 2023b; Li et al., 2022). These techniques have significantly enhanced performance in code-related tasks such as the generation of code from natural language descriptions and code translation, leading to their integration into consumer-facing products (e.g. GitHub Copilot², Bard³, TabNine⁴). Despite their remarkable capabilities, these large language models (LLMs) might struggle to generalize to unseen code (Section 6.1 in Barke et al. (2023) and Figure 1 in Agrawal et al. (2023); Ding et al. (2022)), leading to inaccuracies especially when working with repositories that contain proprietary software or work-in-progress code.

The main emphasis of this thesis is to select relevant contextual cues from the given task and leverage them effectively in deep learning models of code. Adding contextual cues provides a way to expand the class of functions that the model can represent by introducing information that it wouldn't otherwise have access to based solely on the original input. Providing selected contextual cues also directs the model's attention toward specific information it should prioritize. Both these factors help in improving the generalization capability of the models. Leveraging contextual cues can help in adapting to unseen tasks as well as improve performance on existing tasks by making more context-aware predictions.

As an example of a specific use case, consider a scenario where a developer is working within an IDE, editing a code file nested within her organization's proprietary software repository. Suppose our objective is to assist the developer by generating code completions

²<https://github.com/features/copilot>

³<https://blog.google/technology/ai/code-with-bard>

⁴<https://www.tabnine.com/>

from her cursor’s position to the end of the line she is currently editing. Even though the deep learning model that powers the IDE might be proficient in code completion by virtue of it being trained on massive amounts of public code, the model might not be equipped to handle unfamiliar patterns found in a private, work-in-progress repository. In order to make accurate predictions, the model needs to understand both the local and global structure of the repository files, their contents, the API dependencies, and any unique coding patterns specific to the user, repository, or organization as a whole. One strategy to instill this understanding in the model is to finetune it on the code from the current repository. However, in certain scenarios, finetuning may not be possible due to black-box access to the model or privacy concerns prohibiting access to available data. Moreover, finetuning is typically computationally expensive and slow, rendering it impractical, particularly for continuously evolving codebases. Therefore, it is crucial to devise approaches that can effectively harness the prior knowledge embedded in pretrained deep learning models, as well as optimally leverage the context available from the current task. This leads us to strategies that initially identify and select relevant contextual cues, and subsequently, effectively incorporate them into the model. In the use case presented above, one valuable contextual cue that can aid in code completion could be the code present in other files in the repository, like method names from the imported files or identifiers used in files present in the same directory as the current file. Contextual cues could also be derived from signals collected from sources such as interpreters (like execution states), API documentation, or even the history of commits made by the developer.

1.1. Enhance-Predict: Our General Framework

In this section, we provide a more precise formulation of our problem setting and outline our general framework that forms the basis of all the articles discussed in the thesis. Given an *input context* X and some *context meta-information* W , our goal is to effectively harness contextual cues based on X and W , such that the *predicted target* \hat{Y} aligns closely with the *actual target* Y . The context meta-information W provides insights about the origin of the input context X . Let us revisit our earlier example of code autocompletion where we are given a position (cursor’s location) in a file and our task is to predict the rest of the line that follows the cursor. The input context X consists of the code before the cursor in the current file, while the actual contents of the tokens following the cursor form the target Y . The context meta-info W may include the repository’s directory structure where the current file resides, as well as the contents of all source code files within that repository. In this thesis, we tackle this problem in two stages: (a) *Context Enhancement*; and (b) *Prediction using the Enhanced Context*.

Context Enhancement The goal of this phase is to extract relevant signals from the input context and the context meta-information, which can enhance task performance beyond what’s possible using the input context alone. The extracted signals are referred to as the *support context* Z . This stage can be expressed as follows:

$$Z = Enhance(X, W) \tag{1.1}$$

In the above equation, *Enhance* is a model inspired by the specific domain and task we are addressing. For instance, in our code autocompletion scenario, *Enhance* could be a model that selects the most suitable code snippet from all the files in the repository where the current file resides, assisting in generating the correct completion. The selected code snippet would constitute the support context Z . Combined, X and Z make up the enhanced context.

Prediction using the Enhanced Context The *Predict* module takes in the enhanced context (X and Z) as input and gives the predicted target \hat{Y} as output. This stage can be expressed as follows:

$$\hat{Y} = Predict(X, Z) \tag{1.2}$$

In our code-completion example, *Predict* can be an LLM that takes in a concatenation of the context prior to the cursor in the current file X along with the method names from the first imported file Z as an input prompt and makes a prediction for the remaining tokens following the cursor \hat{Y} .

Without Context Enhancement When support context is not obtained, the task corresponds to taking the target context X and directly predicting the target \hat{Y} . This can be expressed as follows:

$$\hat{Y} = Q(X) \tag{1.3}$$

In our code completion example, this means that the LLM will take the context prior to the cursor in the current file as input and produce the completion.

1.2. Thesis Overview

The thesis is composed of four articles, each focusing on various methods of identifying and selecting relevant contextual cues, as well as different strategies for integrating these cues into deep learning models for code. The first article focuses on the task of programming by examples where the input-output (IO) examples given as part of the specification constitute the input context X and the next step of the program that satisfies those examples constitutes the target Y . The subsequent articles focus on the task of single-line code completion where the context prior to the cursor in the current file constitutes the input context X and the tokens after the cursor till the end of the line constitute the target Y .

Chapter 2 offers an overview of machine learning for code, discussing various code representations, employed models, and applications in this domain. This is followed by a discussion

Table 1. Summary of how the articles presented in the thesis fit within our general framework. PE, CA, RLPG, and TSSA denote Per-Example, Cross-Aggregator, Repo-Level Prompt Generator, Targeted Support Set Adaptation.

Article	Input X	Target Y	Context Meta-Info W	Support Context Z	Enhance	Predict
Chapter 4	set of given IO examples	line t of the global program	Same as X	PE solutions + execution states of each line of PE solutions + execution state of line $t - 1$ of global program	PE model (for PE solutions) + code interpreter (for execution states)	CA
Chapter 6	few tokens preceding the cursor in the current file	next token after the cursor	position of the cursor + current file	support tokens + support windows	strategies based on frequency of occurrence of tokens	TSSA
Chapter 8	all tokens preceding the cursor in the current file	tokens after the cursor till the end of line	position of the cursor + current file’s repository	context from a single prompt proposal predicted by RLPG	list of prompt proposals + RLPG	Code LLM
Chapter 10	all tokens preceding the cursor in the current file	tokens after the cursor till the end of line	position of the cursor + current file’s repository	repo contexts (e.g. contexts from all the prompt proposals)	module for obtaining the repo contexts	RepoFusion

of the basic principles behind large language models of code including the transformer model and prompting as a strategy to adapt these models for downstream tasks.

After an introductory prologue in Chapter 3, Chapter 4 presents our first article (Shrivastava et al., 2021). This article presents a novel approach to neural program synthesis by dividing the programming by examples problem into two stages: (a) obtaining the support context Z consisting of solutions that satisfy individual examples called per-example (PE) solutions, their step-wise execution states, and the execution state of the previously generated line of the global program (a program that satisfies all examples), and (b) leveraging the output from step (a) to learn a *Predict* framework called *Cross Aggregator (CA)* that generates a prediction \hat{Y} for the next step of the global program.

After an introductory prologue in Chapter 5, Chapter 6 presents our second article (Shrivastava et al., 2020). This article proposes a *Predict* module called *Targeted Support Set Adaptation (TSSA)* that is designed with the motivation of adapting the code model to the local, unseen context. The module achieves this by retrieving targeted information tokens (support tokens) from the current file, along with a few tokens preceding them (support window), and leveraging this support context to update the parameters of the code model.

As a result, the model is able to generate accurate predictions \hat{Y} for the token immediately following the cursor.

After an introductory prologue in Chapter 7, Chapter 8 presents our third article (Shrivastava et al., 2022) that builds upon Chapter 6 to leverage contextual cues from the entire repository based on a set of *prompt proposals*. These prompt proposals govern the location and content of the support context based on the repository’s structure and code in relevant files such as imports and parent class files. We introduce the *Repo-Level Prompt Generator (RLPG)* module, that is trained to select a prompt proposal conditioned on the context around the intended completion. The context derived from the selected prompt proposal is then used to prompt a code LLM (*Predict* module), generating a prediction \hat{Y} for completing the current line.

After an introductory prologue in Chapter 9, Chapter 10 presents our fourth article (Shrivastava et al., 2023) that builds upon the insights from Chapter 8. This article introduces RepoFusion, a framework (*Predict* module) that learns to combine multiple relevant contexts from the repository (referred to as repo contexts in Table 1, e.g. contexts from all prompt proposals) in order to produce a prediction \hat{Y} for completing the current line. An example of relevant contexts from the repository, referred to as *repo contexts*, includes the contexts derived from all prompt proposals used in Chapter 8, without selecting a single context using RLPG.

Chapter 11 presents a summary and broader impact of the contributions of the thesis along with interesting directions to explore in the future.

Please see Table 1 for a comprehensive summary of how the articles featured in this thesis fit within our general Enhance-Predict framework outlined in Section 1.1.

1.3. List of Excluded Contributions

During my PhD, I have had the opportunity to work on various projects focused on out-of-distribution generalization, reinforcement learning, and human-computer interaction, which, although not included in this thesis, provided valuable experiences and insights. Some of these works are outlined below.

- **Transfer Learning by Modeling a Distribution over Policies.** Disha Shrivastava*, Eeshan Gunesh Dhekane*, Riashat Islam (*ICML Workshop on Multi-Task and Lifelong Reinforcement Learning 2019*) [Shrivastava et al. (2019)]. We propose a transfer learning approach that encourages exploration in the target environment by maximizing the entropy of a distribution over policies in the source environment.
- **Minimax and Neyman–Pearson Meta-Learning for Outlier Languages.** Edoardo Maria Ponti*, Rahul Aralikkatte*, Disha Shrivastava, Siva Reddy, Anders

Søgaard (*Findings of ACL 2021*) [Ponti et al. (2021)]. We propose two training objectives (Minimax MAML and Neyman-Pearson MAML) that are better suited for robust out-of-distribution transfer in low-resource languages.

- **Approach Intelligent Writing Assistants Usability with Seven Stages of Action.** Avinash Bhat, Disha Shrivastava, Jin L.C. Guo (*CHI Workshop on Intelligent and Interactive Writing Assistants 2023*) [Bhat et al. (2023)]. Inspired by the cognitive model of Norman’s seven stages of action, we propose a framework to guide the user interaction design of LLM-based intelligent writing assistants.
- **Iterative Editing with AlphaCode.** As part of my summer internship in 2022 at *DeepMind*, I worked on improving the performance of the AlphaCode (Li et al., 2022) model for competitive programming problems by learning to iteratively edit the generated programs.

Chapter 2

Background

In this chapter, we present a brief overview of the field of Deep Learning for Code (DL4C), which encompasses research at the convergence of machine learning, programming languages, and software engineering. We start by discussing some distinctive characteristics of code that motivate the development of specialized models dedicated to its analysis and understanding. Next, based on the classification done in [Allamanis et al. \(2018a\)](#), we categorize different machine learning models of source code and briefly explain them. This is followed by an overview of some applications in this broad domain. Finally, we explore the realm of Large Language Models (LLMs) for Code, delving into key concepts such as transformers and prompting.

2.1. Uniqueness of Code

The naturalness hypothesis ([Hindle et al., 2012](#)) states that *software corpora have similar statistical properties to natural language corpora*. Even though it might seem natural to directly apply ML models for natural language to source code, there exist some important differences between the two that motivate the need for developing separate models for source code. First, the code is executable and more formal with clear syntax. This results in code being highly sensitive to minor changes (e.g. changing the indentation or swapping the types and/or order of arguments, etc.). On the other hand, natural language is more robust. A reader can often comprehend the meaning of the text with minor changes. Second, the rate of evolution of natural language is gradual, emerging through complicated social dynamics. Software corpora, on the other hand, evolve quite fast with new features being rolled and bugs getting fixed, as only a few designers control this aspect for many users. Also, neologism, e.g. new names for identifiers, is extremely common in code as compared to natural language where a person seldom uses new words to express a known concept.

2.2. Models of Code

A machine learning model of code commonly represents a probability distribution over various properties or representations of code. Machine learning provides a systematic way to handle the uncertainty (via probabilities) and assimilate different code representations (e.g. token-level representation, AST, execution traces, etc.). Below, we group these models based on their mathematical form, along with their inputs and output types. This categorization is taken from [Allamanis et al. \(2018a\)](#).

2.2.1. Code-Generating Models

Code generative models capture the generative process of code. Given an output code representation Y , and a possibly empty context X , these models learn the probability distribution $P(Y|X)$. During inference, one can sample from P to generate code. Depending on how code is represented, we may have: (a) token-level or sequence models which view code as a sequence of code tokens, subwords or sentence pieces ([Hellendoorn & Devanbu, 2017a](#); [Karampatsis & Sutton, 2019](#)); (b) syntactic models that model code at the level of abstract syntax trees (ASTs) ([Alon et al., 2019](#); [Bielik et al., 2016](#)); and (c) semantic models which model the code as a graph, generalizing sequence and tree-based models ([Li et al., 2015](#)). Code-generating models can also be used in a scoring function, assigning probabilities to fragments of code.

Let’s discuss a bit more about two sequence models which will be of interest to us in the rest of the thesis. Token-based sequence models usually view code as a sequence of elements, usually tokens, characters, or subwords, i.e. $Y = y_1, \dots, y_M$. Predicting a large sequence in a single step is intractable due to the exponential number of possible sequences (for a vocabulary with V elements, there are $|V|^N$ sequences of length N). Therefore, most sequence-based models predict sequences by generating each element one after the other, i.e. they model the probability distribution $P(y_m|y_1, \dots, y_{m-1}, X)$. Formally, we can write this as follows:

$$P(Y|X) = P(y_1, \dots, y_M|X) = \prod_{m=1}^M P(y_m|y_{m-1}, y_{m-2}, \dots, y_{m-n+1}, X) . \quad (2.1)$$

In the above equation, for practical reasons, the generated sequence so far is assumed to depend only on the previous $n - 1$ tokens. When $X = \emptyset$, we refer to such a model as a **language model**. On the other hand, if $X = S$ where S can be the representation of another piece of code (which may or may not be of the same form as Y), we refer to this model as a sequence to sequence or **seq2seq model**. The probability distribution P can be modeled by a machine learning model. Neural seq2seq models consist of encoder and decoder modules. The encoder takes as input a context representation S and the decoder

produces the output code representation Y . The information from the encoder to the decoder is transferred via hidden states, which can store the information about previous tokens apart from the $n - 1$ tokens in the current context. Note that the generated code produced from P (say, via techniques like beam search), is not always guaranteed to compile and execute. However, some external constraints can be enforced on the output to achieve that, e.g. Maddison & Tarlow (2014) generate variables declared only within each scope.

In earlier seq2seq models such as LSTM (Hochreiter & Schmidhuber, 1997) or GRU (Cho et al., 2014), at each step of generation, the decoder considers only the last hidden state of the encoder as context. However, to effectively capture long-range dependencies, the attention mechanism (introduced by Bahdanau et al. (2014) in the context of machine translation) enables the decoder to examine all hidden states of the encoder. When translating a French sentence to English, this means that for generating each word in the output English sentence, the decoder weights the input tokens based on a score that is indicative of the level of attention that should be given to them. Mostly, the scores are calculated by taking a dot product of the decoder hidden state of the previously generated English word (query) with the encoder hidden states for each word in the input French sentence (keys). Scores are normalized across input positions using a softmax activation function to produce attention weights. The attention weights are then used to compute a weighted sum over the encoder hidden states (values) to obtain a context-aware representation of the French sentence. Note that the score between queries and keys can be computed in parallel. Later, Vaswani et al. (2017a) proposed a new model architecture based entirely on the attention mechanism, called the *Transformer*. We discuss Transformers in more detail in Section 2.4.1.

2.2.2. Representational Models

To predict code properties that may be directly useful for downstream tasks, researchers have built models to learn intermediate representations of code, $f(X)$ where f is a function that transforms code into an intermediate representation (which may not be human interpretable). This representation is then used to learn a conditional probability distribution of a code property λ as $P(\lambda|f(X))$, e.g. λ can be the type of a variable. These models may be based on distributional representations where the elements being represented and their relations are encoded within a multidimensional real-valued space and the similarity between two representations can be measured within this space (Allamanis et al., 2015; Feng et al., 2020; Li et al., 2015) or based on structured prediction wherein models predict a set of structured objects (Raychev et al., 2015; Proksch et al., 2015) or a combination of both.

2.2.3. Pattern Mining Models

Pattern mining models are unsupervised machine learning models which tend to discover patterns like clusters or groupings in source code, which can then be better understood by software engineers. These models learn a probability distribution, $P(X) = \sum_z P(X|Z = z)P(Z = z)$ where Z represents a set of latent variables that represent the grouping. Applications of these models are common in the mining software repositories community and include documentation (e.g. API patterns) (Fowkes & Sutton, 2016) and summarization (Fowkes et al., 2017).

2.3. Applications

Below we provide a categorization (with loose boundaries) of prominent application areas of DL4C, based on the level of interaction with the end user.

2.3.1. Improving Programming Experience or Front-End

These applications try to improve the coding experience of an end-user. They include systems that make recommendations to assist software engineering tasks like code autocompletion in an IDE (i.e. inferring a user’s intent in order to recommend future code) and suggesting likely code reviewers for a given code change. They involve tasks like inferring coding conventions (e.g. variable naming), detecting bugs by predicting the exact span of tokens where the bug is present as well as suggesting a fix for that bug, detecting code clones, and code refactoring. In each of these applications, the machine learning models can be adapted to give personalized fixes, recommendations, and code refactoring suggestions by learning models from user-specific code edit patterns or patterns from related users who operate in the same project.

2.3.2. Improving Programming Tools or Back-End

This category includes applications that focus on developing techniques for improving general programming tools. Program analysis is an important area in this category that analyses the behavior of computer programs for a property like correctness, robustness, and safety. It includes the development of tools for program verification, formal automated proof, or learning appropriate parameterization of a static analysis tool by inferring the data-flow information from code. Another application in this category is linking code with documentation (text that captures requirements, specifications, and descriptions of code) to be used in code search engines (where a query can be in natural language).

2.3.3. Program Synthesis

The goal of a program synthesis system is to produce one or more programs that satisfy a given specification. The process of program synthesis can be described in terms of the following key constituents:

- **Specifications:** The user specifications can take multiple forms, with some of them being input-output examples, natural language statements/keywords, formal statements describing the behavior of the program (e.g. $\forall x : y = 3*x$), and demonstrations in the form of 2D/3D graphic renderings (Ellis et al., 2019b). The specification should be such that it explains the problem properly but it isn't as detailed as the program for which the specifications are to be met.
- **Domain Specific Language (DSL):** Program synthesis systems are usually characterized by a domain-specific language (DSL) which can be thought of as the language which is followed for synthesizing programs. A DSL is defined as a context-free grammar consisting of a set of terminal symbols, a set of non-terminal symbols, a start symbol and a set of production rules for expanding the non-terminal symbols. Semantically, each symbol in the DSL can be interpreted as ranging over a set of values and each production rule can be interpreted as a function. Similar to specifications, the DSL should be expressive enough to capture the problems that we wish to solve. At the same time, it should be more restrictive than full programming languages to limit the difficulty of the search.
- **Search:** The core challenge in program synthesis is to search the vast space of possible programs to find the ones that comply with the specifications. One basic example of a search procedure is the top-down enumerative tree search, where a brute force search is performed over the space of possible programs by enumerating all possible derivations in the grammar specified by the DSL and maintaining an ordered list of partial derivations in a top-down manner starting from the start symbol in the DSL as the root. In this simplest form, at depth k of the tree, we explore all expressions of the form `FUNCTION(ARG1, ARG2)`, where ARG1 and ARG2 are placeholders for expansions of non-terminals in the partial derivations until depth $k-1$ and `FUNCTION` denotes production rule. This recursive search is going to be exponential in the depth k . We can prune the search space by expanding only those branches of the search tree which produce unique outputs for the partial derivations obtained until now, thereby removing redundant branch traversals. In recent years, some neural-guided approaches (Balog et al., 2016; Zohar & Wolf, 2018) have also been proposed, where the probabilities from a neural network are used to order the nodes in the search tree.
- **Ranking:** In certain cases, there may be more than one candidate program that can meet the specifications. In these scenarios, a secondary problem is then to rank the

candidate programs based on task-dependent criteria such as favoring conciseness by scoring shorter programs higher than longer programs.

2.4. Large Language Models of Code

In this section, we provide a brief overview of Large Language Models (LLMs) along with the basic concepts needed to understand them. Large Language Models are language models (see Section 2.2.1) that contain a large number of parameters (typically of the order of millions or billions) and are trained on massive amounts of data from the Internet such as Wikipedia, Common Crawl or GitHub. LLMs are typically based on neural networks called transformers, which we describe below.

2.4.1. Transformers

Vaswani et al. (2017a) proposed a new model architecture based entirely on the attention mechanism, called the *Transformer*. An important concept in Transformers is the notion of *self-attention*, which involves calculating attention weights between the encodings of the same sequence. In self-attention, the query and keys are derived from the same sequence, as opposed to coming from different sequences. This work expresses attention as a function of three tensors, \mathbf{Q} , \mathbf{K} and \mathbf{V} , that correspond to the queries, keys and values, respectively (Equation 2.2). To capture information from different representation subspaces, the attention function is calculated on different projections of the queries, keys and values. The outputs of the application of the attention function to different projections (called heads) are concatenated and combined. This procedure is referred to as *multi-head attention* (see Equation 2.3).

$$Att(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \quad (2.2)$$

$$\begin{aligned} MultiHead(\mathbf{Q}, \mathbf{K}, \mathbf{V}) &= \text{concat}(head_1, head_2, \dots, head_\tau)W^O \\ \text{where } head_i &= Att(\mathbf{Q}W_i^Q, \mathbf{K}W_i^K, \mathbf{V}W_i^V) \end{aligned} \quad (2.3)$$

In these equations, d_k is the dimension of the key, W_i^Q, W_i^K, W_i^V are the query, key and value projection matrices for $head_i$, τ is the number of heads and W^O is the linear projection that combines the heads. The output from Equation 2.3 is fed to a positionwise fully-connected feedforward network. A residual connection (He et al., 2016) followed by layer normalization (Ba et al., 2016) is applied before and after the feedforward network. These series of operations are represented in the form of an attention block.

Figure 1, taken from Vaswani et al. (2017a), represents the transformer encoder-decoder architecture. Both the encoder and decoder modules are comprised of stacking multiple

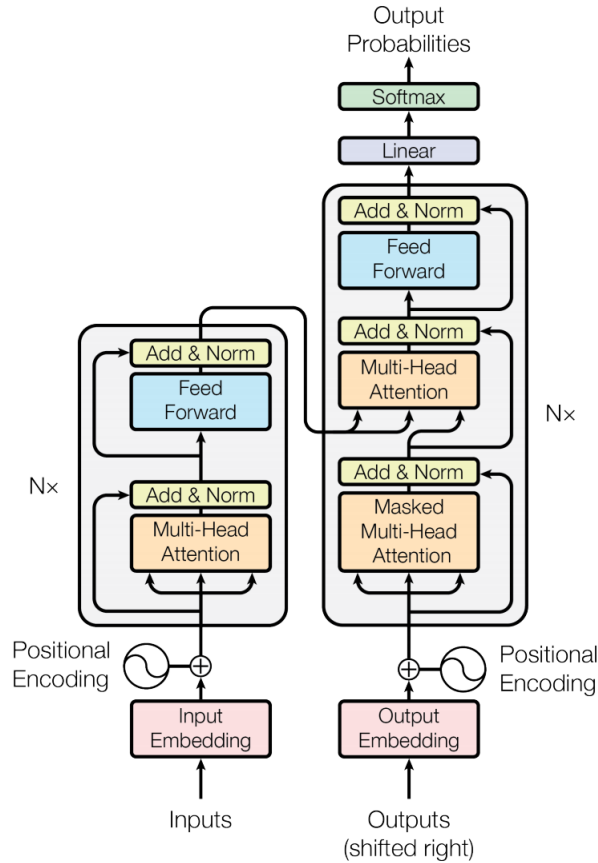


Fig. 1. This figure taken from [Vaswani et al. \(2017a\)](#) represents the architecture of a Transformer encoder-decoder model.

attention blocks. The encoder uses self-attention to establish associations between tokens of the input sequence while the decoder uses a causal self-attention that attends only to the previously generated tokens via masking the future tokens. The decoder also has cross-attention (as discussed before) where the keys and values come from the encoder whereas the query is the output from the previous attention block in the decoder. In order to model the word order in sequences of text, transformers use a fixed encoding that corresponds to a position in the sequence. These position encodings are added to the inputs of both the encoder and the decoder.

The Transformer model achieves parallelization as well as faster training and inference by attending to multiple parts of the input sequence simultaneously, eliminating the need for sequential processing. In recent years, transformers have revolutionized the field of natural language processing by demonstrated state-of-the art performance in tasks such as machine translation, text summarization and question-answering ([Brown et al., 2020a](#)). Transformers have also been used with massive success for applications involving source code such as code

completion, program synthesis, code summarization and code translation (Chen et al., 2021; Li et al., 2022; Austin et al., 2021; Nijkamp et al., 2023b).

2.4.2. Pretraining

The general idea behind pretraining is to train a model on massive amounts of data (e.g. text) to learn the statistical patterns and general knowledge present in that data. This pretraining step is typically performed on a large-scale dataset, such as a large portion of the Internet or a vast collection of books or code in all of GitHub, which provides a diverse range of syntactic and semantic contexts and helps learn a general understanding of language or code. Since transformers excel at processing intricate and extensive datasets, exhibit favorable scalability with increasing model parameters and data (Kaplan et al., 2020) and are suitable for use with modern compute hardware owing to parallelization, they have become a prominent choice for use as pretraining models. Based on the architecture and pretraining objectives, there are broadly three variants of the transformer model that have emerged over the years.

- **Decoder-only:** Decoder-only models consist of a single decoder that performs an autoregressive, left-to-right processing and generation of tokens. The pretraining objective here usually corresponds to the standard language modeling objective (see Section 2.2.1) that corresponds to predicting the probability of the next token given the previous tokens. Some examples of these models applied to code modeling are Codex (Chen et al., 2021), Codegen (Nijkamp et al., 2023b), Google’s program synthesis model (Austin et al., 2021), GPT-J (Wang & Komatsuzaki, 2021), and InCoder (Fried et al., 2022). While these models trained on autoregressive language tasks, such as code completion, excel at generating code, the left-to-right nature of these tasks does not always align with the programming process in an IDE. To address this, modifications to the pretraining approach, such as Fill-in-the-Middle (Bavarian et al., 2022), have been proposed. This method involves transforming the dataset by moving code spans from the middle of the file to the end. This allows the autoregressive models to learn to infill code, hence enabling them to better capture the non-linear nature of programming (Allal et al., 2023; Li et al., 2023).
- **Encoder-only:** Encoder-only models typically employ a single encoder that incorporates bidirectional encoding of the context, as opposed to the unidirectional (left-to-right) approach. These models use a masked language modeling objective for pretraining (first introduced in BERT (Devlin et al., 2018)), which corresponds to predicting masked tokens based on the surrounding context. Some examples of these models applied to code modeling are CodeBERT (Feng et al., 2020) and CuBERT (Kanade et al., 2020). The representations from these models serve as useful

general representations of code that can be employed for downstream tasks such as code classification, and clone detection.

- **Encoder-Decoder:** As described in Section 2.2.1, these class of seq2seq models consist of separate encoder and decoder modules. The encoder typically uses a bidirectional encoding of the context while the decoder uses this context obtained from the encoder to autoregressively decode either the next token or a series of masked tokens in order. Some examples of these models applied to code modeling are Alpha-Code (Li et al., 2022) and Code-T5 (Yue Wang, 2021). The latter in addition to the above mentioned pretraining objectives also uses a denoising sequence reconstruction objective (Lewis et al., 2019). These models perform well in conditional seq2seq generation tasks such as code translation, code summarization and generation of code given a natural language comment.

2.4.3. Prompting

Once the pretraining phase is completed, the pretrained language model can be further finetuned on specific downstream tasks. Finetuning involves training the model on a smaller dataset that is more specific to the target task. However, as mentioned in the Introduction (Chapter 1), in certain scenarios, finetuning may not be a feasible or practical approach due to various factors such as limited availability of labeled data required to achieve satisfactory performance on the target task, limited or no access to the pretrained model weights, or the computational cost associated with finetuning models that contain millions or billions of parameters, particularly in the case of pretrained large language models (LLMs). As an alternative to this pretrain-finetune paradigm, *prompting* has emerged as an effective strategy to instill the knowledge of the downstream task into the LLM. A prompt refers to an input provided to a model that allows the model to generate predictions related to the desired task. Prompting involves utilizing the knowledge encoded in the pretrained weights of an LLM to facilitate the downstream task without the need to finetune the model weights. The task can be specified in the form of a prompt, which can take the form of natural language instructions or demonstrations of desired input-output mappings using example data. Prompting has demonstrated promising results in various scenarios, including cases where only a few examples or even no examples at all (referred to as zero-shot) are available, showcasing the promise of prompting as a strategy to generalize across diverse tasks (Brown et al., 2020a).

Besides providing a mechanism to control and evaluate a language model, prompts have been shown to elicit emergent behaviour as well. Examples of this behavior include GPT-3 (Brown et al., 2020a) doing better in tasks it has never seen during training and improved reasoning capabilities with few-shot (Wei et al., 2022) and zero-shot (Kojima et al., 2022)

prompts that encourage a chain of thoughts. These factors highlight the importance of designing an effective task-specific prompt.

There has been some work toward automatic prompt generation techniques. This includes techniques that correspond to producing continuous/soft prompts where the prompt is described in the latent space of a language model (Li & Liang, 2021; Qin & Eisner, 2021; Bragg et al., 2021; Lester et al., 2021; Liu et al., 2021b) and techniques that produce discrete prompts where the prompt is a text string that can be interpreted by a human (Shin et al., 2020; Gao et al., 2021; Schick & Schütze, 2021). Please see (Liu et al., 2021a) for a comprehensive survey of different prompting techniques.

Chapter 3

Prologue to the first article

3.1. Article Details

Learning to Combine Per-Example Solutions for Neural Program Synthesis. Disha Shrivastava, Hugo Larochelle, Daniel Tarlow. This article ([Shrivastava et al., 2021](#)) was accepted for publication at the *Neural Information Processing Systems (NeurIPS) 2021*.

Personal Contribution The inspiration for the work came from FrAngel ([Shi et al., 2019](#)), a program synthesis framework that iteratively refines a program by mining fragments of Java code from partial solutions, based on a set of rules and predefined heuristics. Daniel Tarlow suggested the potential value of employing this intuition of leveraging cues from the partial solutions in tandem with neural program synthesis frameworks. Hugo Larochelle and Daniel Tarlow proposed initial investigative studies that proved fundamental in establishing that it is easier to find programs that satisfy examples partially and there is value in aggregating these partial solutions. Motivated by the efficacy of PCCoder ([Zohar & Wolf, 2018](#)), we decided to leverage the step-wise execution state of these partial programs to guide their aggregation. Disha Shrivastava was involved in writing all of the code, running the experiments, coming up with the model that aggregates the partial solutions, and writing significant parts of the paper. Hugo Larochelle and Daniel Tarlow advised on the project and were involved in the discussion of results, suggesting experiments to run, writing parts of the paper, and offering constructive critique during the drafting and revision phases of the paper.

3.2. Context

The goal of program synthesis from examples is to find a computer program that is consistent with a given set of input-output examples. Most learning-based approaches try to find a program that satisfies all examples at once, which under most settings can be hard. Our work, by contrast, tries to find this program in parts. To understand this motivation,

imagine a process wherein a programmer is asked to write a program that satisfies a set of unit test cases. They may begin by figuring out a program that satisfies a subset of unit test cases first, and later modifying the program to incorporate other corner cases. We consider an approach that breaks the problem into two stages: (a) find programs that satisfy only one example, and (b) leverage these *per-example solutions* to yield a program that satisfies all examples. For the second stage, we propose a neural architecture which we term as the *Cross Aggregator*. In terms of the broader theme of the thesis (see Section 1.1), for synthesizing one line of the global program (target Y that meets the given specifications), the support context Z is provided by the per-example solutions along with their corresponding step-wise execution states as well as the execution state of the previously generated line of the global program. Our proposed Cross Aggregator model serves as the *Predict* module. The subsequent chapter (Chapter 4) discusses this work in detail.

3.3. Contributions

The paper proposed a novel approach called *Neural Per-Example Program Synthesis* (N-PEPS) that uses neural networks to first find per-example solutions and then learns to combine the cues present in these per-example solutions to synthesize a global solution. For the latter, the paper introduces the *Cross Aggregator* neural network module based on a multi-head attention mechanism. Evaluation across programs of different lengths and under two different experimental settings revealed that when given the same time budget, the proposed technique significantly improved the success rate over PCCoder (Zohar & Wolf, 2018) (one of the leading techniques for neural program synthesis at the time) and other ablation baselines. In addition, we **open-sourced** the code, data, and trained models for the work to facilitate future research in this direction.

3.4. Research Impact

The general idea of this work is breaking down a complex problem into simpler sub-problems, learning the solutions of the sub-problems, and then automatically combining the solution of these sub-problems. We employed the concept of iterative program generation by capitalizing on the execution states of partial solutions. In the context of program synthesis, at the time, this was the only work that used neural networks to automatically learn to combine partial solutions. Another important aspect of the work was our unique experimental configuration, which involved a substantially shorter timeout of 5 seconds as compared to the previous works such as Zohar & Wolf (2018) with longer durations such as 5000 or 10000 seconds. This particular feature, coupled with our strong empirical results enhances the appeal of our framework for real-world, interactive implementations of program synthesis systems. Recently, in the context of Large Language Models (LLMs), our idea would involve

using the LLM to generate a partial solution either for each example or for each sub-problem using techniques akin to PAL ([Gao et al., 2022](#)). These partial solutions along with their associated execution traces can then serve as input to the LLM, prompting the synthesis of a complete program.

Chapter 4

Learning to Combine Per-Example Solutions for Neural Program Synthesis

4.1. Introduction

Program synthesis from examples tackles the problem of coming up with a computer program that satisfies a given set of Input-Output (IO) examples. Since the space of possible programs is large, an exhaustive search can be extremely time-consuming. Therefore, development of systems for program synthesis that can come up with a solution (program satisfying the given IO examples) within a *limited time*, such that it is practical for real-world applications, is a challenging task.

Neural-guided program synthesis systems (Balog et al., 2016; Zohar & Wolf, 2018) try to expedite the search by using a neural network conditioned on the IO examples as a learned heuristic for the search procedure. In these systems, a neural network outputs probabilities over programs or properties of programs (e.g. functions). These probabilities are then utilized to guide a search like depth-first or beam search. These systems try to find a program that satisfies all IO examples *simultaneously*, which under most of the settings can be hard. What if instead, we try to find this program in parts? To understand this motivation, imagine a process wherein a programmer is asked to write a program that satisfies a set of unit test cases. They may begin by figuring out a program that satisfies a subset of unit test cases first, and later modifying the program to incorporate other corner cases. Shi et al. (2019) uses this intuition to iteratively refine a program by mining fragments of Java code from partial solutions, based on a set of rules and predefined heuristics. Gupta et al. (2020) also uses the same intuition, but in a different application for program repair. In this work, we consider breaking the complex problem of finding a program that satisfies all N given IO examples (called the *global solution*) into N smaller, easy to solve sub-problems, where each sub-problem involves finding a program satisfying only one IO example (called *per-example solution*). The cues present in these per-example (PE) solutions are then combined

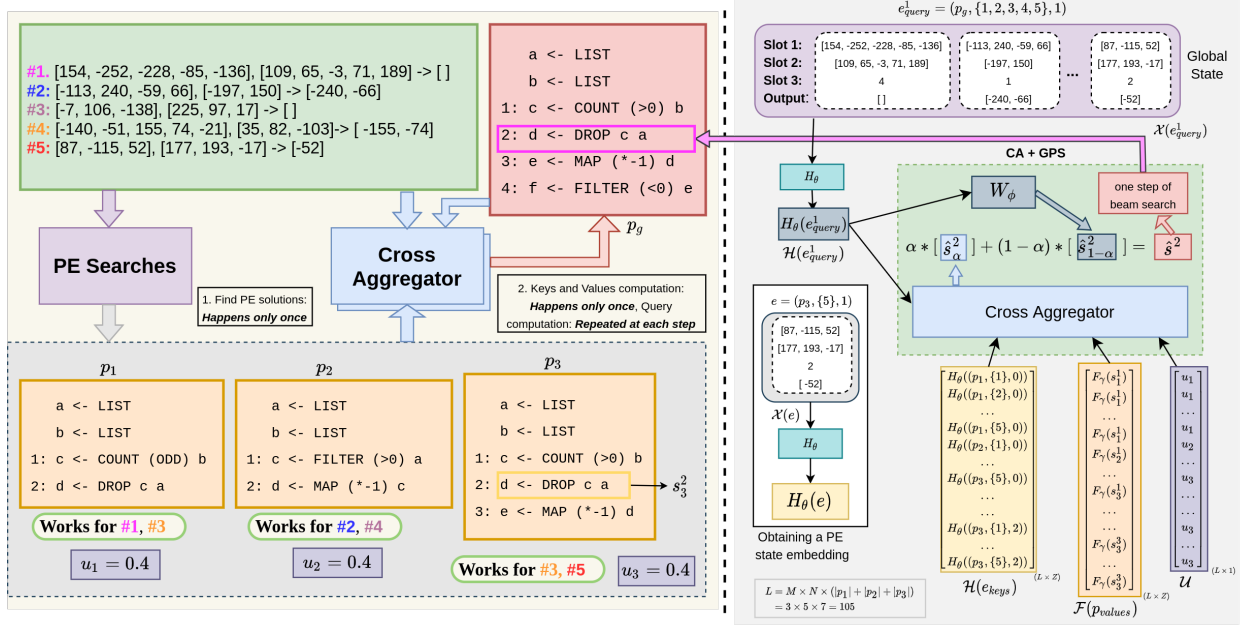


Fig. 4.1. Figure explaining the idea of N-PEPS: (Left) Illustrating the two stages of N-PEPS with an example; **(Right)** Synthesizing line 2 of p_g using contributions from CA and GPS, with details of how query, keys and values and relation scores are formed. White box shows an example of obtaining a PE state embedding.

to provide useful signals that can help guide the search for the global solution effectively. As a motivating example, consider the left part of Figure 4.1, where five IO examples are given as a specification (green box) and we need to find a global solution p_g (red box) that satisfies these five examples. The first stage of our approach consists of performing per-example searches to find a program p_i conditioned on the i -th IO example. In our example, we start from IO example #1 and find program p_1 . In addition, we also check if p_1 satisfies any other examples (#3 in figure). Iterating through the examples in this way results in a set of programs (p_1, p_2, p_3) that, taken together, in the ideal scenario, would satisfy all five IO examples. Looking closely at the discovered PE solutions, we see that they contain fragments of the global solution. This brings us to the second stage of our approach that addresses the challenge of how best to aggregate these PE solutions to produce a global solution. Towards that goal, we propose a neural network based architecture, which we refer to as *Cross Aggregator* (CA). It is designed to learn to combine the cues present in these PE solutions, in a way that helps guide the search for p_g . We model this aggregation using a multi-head cross-attention mechanism, which leverages the state of step-wise execution of the PE solutions and the synthesized global solution so far (see Section 4.3.2 for details). Our key contributions can be listed as follows:

- We consider breaking the standard program synthesis pipeline into two stages: (a) discovering PE solutions, and (b) aggregating the PE solutions such that it leads to

a global solution. We refer to our approach that uses neural networks at both these stages as *Neural Per-Example Program Synthesis* (N-PEPS).

- We propose a neural network based multi-head attention architecture called Cross Aggregator (CA) that makes use of step-wise execution information to *learn* to combine the PE cues such that it helps guide the search for the global solution.
- We demonstrate via experiments with programs of different lengths and under two different evaluation settings that when given the same time budget, our formulation shows significant improvements in success rate when compared to PCCoder (Zohar & Wolf, 2018) (one of the leading techniques for neural-guided program synthesis) and other ablation baselines.

4.2. Background

Suppose we are given a set $X = \{(x_i, y_i)\}_{i=1}^N = \{r_i\}_{i=1}^N$ of N IO examples and our task is to come up with a program p_g that satisfies these examples. The i -th IO example r_i consists of a pair of input x_i and output y_i . The program consists of T lines (excluding lines with input variable declarations), i.e. $p_g = [p_g^t]_{t=1}^T$. To be practically meaningful, we impose the constraint that p_g has to be found within a given time budget, specified by a *timeout* value. The syntax and semantics of p_g are governed by a domain-specific language (DSL). We use the DSL provided by Balog et al. (2016), which contains first-order functions (e.g. SORT, REVERSE) and higher-order functions (e.g. MAP, FILTER) that can take *lambda* functions (e.g. (*4), (<0)) as input. The inputs and outputs can be either an integer or a list of integers (see Appendix F of Balog et al. (2016) for more details about the DSL). The Predict and Collect Coder (PCCoder) (Zohar & Wolf, 2018) provides state-of-art results for this DSL and is illustrative of methods that directly solve for all available IO examples at once. We refer to these methods as Global Program Synthesis (GPS). We will be building on PCCoder to propose our per-example approach.

4.2.1. PCCoder

PCCoder synthesizes programs one line at a time, through a model based on the notion of a *program state*. The program state is a two-dimensional memory of size $N \times (\nu + 1)$ obtained during the execution of t lines (steps) of a program on a set of N inputs. This means that for each IO example r_i , there are up to ν slots for storing the input and intermediate program variables, with an additional slot for storing the output (see Appendix 4.A.2 for more details). Note that the initial state at $t = 0$ consists of only the IO examples (see left part of Figure 4.2).

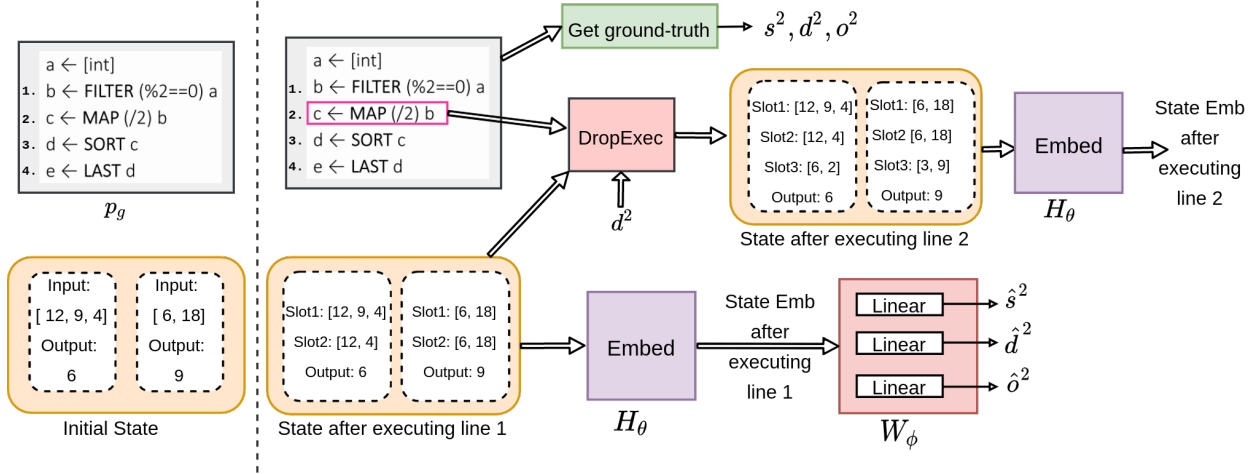


Fig. 4.2. Figure explaining the idea of PCCoder: (*Left*) Sample program along with two IO examples that forms the program state at $t = 0$; (*Right*) Block Diagram explaining the training of PCCoder at line 2 of the program.

PCCoder consists of two learnable components (i.e. neural networks), H_θ and W_ϕ , with parameters θ and ϕ . H_θ obtains the embedding of the current program state by average-pooling the representation of the $\nu + 1$ slots corresponding to individual examples (white boxes inside the state in Figure 4.2) into a vector of fixed size in \mathbb{R}^Z , where Z denotes the embedding size (see Appendix 4.A.2 for details of how these representations of slots are obtained). W_ϕ maps this *state embedding* to predictions of three quantities of interest for the next line in the program: (a) the next operator \hat{o}^t (or function e.g. MAP); (b) the next statement \hat{s}^t (operator along with its arguments e.g. MAP(/2) b); and (c) next drop vector \hat{d}^t which represents positions of variables that can be dropped from the state. The dropping is desirable as it creates slots for storing new variables, which in turn allows for synthesizing longer programs. There is a module called *DropExec* which executes a given line of the program against an example r_i and stores the resulting variable c_i in the next available slot in the state. If all ν slots in the state are filled, a variable is dropped from one of the slots using the drop vector and c_i is stored there. The updated state can then be used for predicting the next line (see right part of Figure 4.2). Next, we provide details of how training and inference is done in PCCoder.

Training: For training H_θ and W_ϕ , several instances of a specification X and the ground-truth program p_g are provided. Given an instance and line t of the program, training operates by obtaining the ground-truth values of statements (s^t), operator (o^t) and drop vector (d^t). The statement and operator values are represented as one-hot vectors of size equal to the number of statements (n_s) and number of operators (n_o), respectively in the DSL. The drop vector is a multi-hot vector of size ν with ones at positions corresponding to the variables in the program that can be dropped, i.e. variables that don't appear in subsequent lines in

the program. The step-wise loss \mathcal{L} is the sum of cross-entropy losses between the actual and predicted statement and operator, and the binary cross entropy loss between each position in the actual and predicted drop vector. The task of predicting the operator is an auxiliary task, i.e. it is used only during training and not at inference time, and is found to improve the training performance. During training, to obtain the updated state, the *DropExec* module chooses the drop-index to be a random entry from those positions in the drop vector d^t that are ones. The right part of Figure 4.2 illustrates the process of training at step 2.

Inference: Inference is done using complete anytime beam search (CAB) (Zhang, 1998) where the time for search is upper bounded by the timeout value. The CAB algorithm operates by performing different beam searches repeatedly in an outer loop. The pruning conditions of the beam search (i.e., beam size, expansion size) are weakened with each iteration of the outer loop, until a solution is found. The inner loop consists of different steps of a single beam search. At each step, the beam consists of the most promising program prefixes, with each prefix represented as a tuple of the current program state, synthesized program until now and the product of the probabilities of the statements in the synthesized program. To synthesize the next line of the program, prefixes are expanded by executing the statements in decreasing order of statement probabilities and taking the argmax of the drop vector probabilities. The statement and drop vector probabilities are obtained using the trained neural networks H_θ and W_ϕ . The search terminates if we find a candidate program prefix that satisfies all N IO examples. The corresponding program is the synthesized global solution p_g . Note that the search may fail and not discover a global solution within the specified timeout. Appendix 4.A gives details of training and inference procedures, and modules of PCCoder.

4.3. Neural Per-Example Program Synthesis (N-PEPS)

As stated in Section 4.1, in this work, we decide to break the complex problem of finding a global solution p_g that satisfies all N IO examples, into N smaller sub-problems. Each sub-problem aims to find a program p_i that will satisfy only the IO example r_i . The cues present in these PE solutions are then aggregated to help guide the search for p_g . We constrain the process of breaking and combining to fit within the specified timeout value. The distribution of total timeout between these stages is treated as a hyperparameter. In this section, we discuss our process of finding PE solutions and follow it with a description of our neural network module that learns to combine the PE solutions.

4.3.1. Per Example Program Synthesis

We refer to the general framework of finding PE solutions first and later aggregating the PE cues to find a global solution, as *Per-Example Program Synthesis* (PEPS). We call

the module that finds PE solutions as the *PE Searches* module. To train the PE Searches module, we use the PCCoder model as it is, except that it is trained to take a single IO example as input as opposed to all the examples in X . We will call this trained model as the *PE model*. We allocate a fixed value of *PEPS timeout*, which is the maximum time given to find each PE solution. The sum of PEPS timeouts across all PE solutions should be less than the total timeout, so that there is some time left for the CA module to aggregate the PE cues (i.e., $N \times \text{PEPS Timeout} < \text{Total Timeout}$). We start from the first example, and using the PE model, try to find a solution that satisfies it. Once found, we also check if this solution satisfies other examples in X . We record the fraction of IO examples satisfied by p_i , and call it the *PE solution score* u_i . If p_i satisfies all examples in X (i.e. $u_i = 1.0$), we stop and return $p_g = p_i$ as the global solution. Otherwise, we proceed to find the next PE solution (based on the order of examples given in X). Note that it is possible that for certain examples in X , we fail to find a PE solution within the PEPS timeout. Once we have our list of M PE solutions ($0 \leq M \leq N$), which ideally satisfies all N examples but may not necessarily, we proceed to aggregating them. Note that when comparison with baselines is not a requirement, we can increase speedup by finding PE solutions in parallel (see Appendix D.1 for more details).

4.3.2. Cross Aggregator

Notation: To formulate the program state, we define a basic unit called an *execution tuple* (ET). An ET $e = (p, \mathcal{S}, t)$ is a tuple consisting of a program p , a subset \mathcal{S} of example indices in X and a step number t . Executing the first t steps (lines) of a program p on every example r_i for $i \in \mathcal{S}$ yields a program state which we note as $\mathcal{X}(e)$. Like PCCoder, we pool the representation of slots of the state corresponding to each example r_i for $i \in \mathcal{S}$ to obtain a state embedding (see Section 4.2.1), hence making its size independent of the size of \mathcal{S} . To represent different combinations of programs executed against different sets of examples at different time steps, we define a list \mathbf{e} of such execution tuples, with its size denoted by L . $(p_1, \{1\}, 0)$ and $(p_3, \{2\}, 2)$ in the bottom right of Figure 4.1 are examples of such combinations. We then execute each entry in \mathbf{e} to get a list of states $\mathcal{X}(\mathbf{e})$. This is followed by embedding each entry in states $\mathcal{X}(\mathbf{e})$ using H_θ to yield a tensor of state embeddings $\mathcal{H}(\mathcal{X}(\mathbf{e})) \in \mathbb{R}^{L \times Z}$ (henceforth referred to as $\mathcal{H}(\mathbf{e})$ for simplicity). The white box towards the bottom of Figure 4.1 shows an example of obtaining a single entry of a PE state embedding.

Motivation: To explain the motivation behind CA, let’s look at Figure 4.1, which illustrates the process of synthesizing line 2 of p_g . Intuitively, at this step, we will want our aggregation mechanism to have more contribution from line 2 of p_1 and p_3 (i.e., DROP c a). A simple way of aggregating the PE solutions can be to take the sum or mean of the PE

one-hot statement vectors (these form our ablation baselines as detailed in Section 4.4.2). However, this strategy will fail for scenarios that require taking a non-trivial combination of the PE solution statements or cases where the global solution requires the generation of a new statement that is not found in the PE solutions.

In this work, we propose another way of anticipating what line of p_g comes next, that makes use of the execution information of the programs. The idea is to compare the state embedding obtained before executing line 2 of p_g with the PE state embeddings corresponding to each step of execution of the PE solutions. Then, based on the learned relevance of these state embeddings, their corresponding next PE program statements can form valuable cues for synthesizing the next line. In other words, if a particular PE program state has high relevance with the global program state at a given step, then the following PE program line is likely to be useful in synthesizing the next line of p_g . We measure this relevance by employing a cross-attention mechanism, with the query formed by the global program state embedding at step t , a key formed by the PE program state embedding at step t and the corresponding value formed by the PE program statement at $t + 1$. We take a set of such keys and values to form the key matrix \mathbf{K} and the value matrix \mathbf{V} , respectively.

Model: For synthesizing line $t + 1$ of p_g , the query \mathbf{Q} is formed from the global state embedding at step t , denoted by $\mathcal{H}(\mathbf{e}_{\text{query}}^t) \in \mathbb{R}^{1 \times Z}$, where $\mathbf{e}_{\text{query}}^t = [(p_g, \{1, 2, \dots, N\}, t)]$. The keys $\mathbf{K} \in \mathbb{R}^{L \times Z}$ are formed from the state embeddings $\mathcal{H}(\mathbf{e}_{\text{keys}})$ of the PE solutions. Let P denote the list of M discovered PE solutions, then the list of execution tuples $\mathbf{e}_{\text{keys}} = [(p_m, \{j\}, t)]$, where $p_m \in P, j \in \{1, 2, \dots, N\}, t \in \{0, 1, \dots, |p_m| - 1\}$, making $L = M \times N \times \sum_{m=1}^M |p_m|$. The corresponding PE solution statements form the values $\mathbf{V} \in \mathbb{R}^{L \times Z}$ (more details on how values are obtained is given later). In addition, we have the relation scores $\mathbf{U} \in \mathbb{R}^{L \times 1}$ obtained by taking the PE solution score u_m corresponding to p_m that is part of each ET in \mathbf{e}_{keys} . Note that entries in \mathbf{U} are dependent only on the program part in the ET, and independent of the subset of example indices and the time index.

$$Att(\mathbf{Q}, \mathbf{K}) = \frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} \quad (4.1)$$

$$RelAtt(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{U}^T + Att(\mathbf{Q}, \mathbf{K})}{2}\right)\mathbf{V} \quad (4.2)$$

$$MultiHead(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{concat}(head_1, head_2, \dots, head_\tau)W^O \quad (4.3)$$

$$\text{where } head_i = RelAtt(\mathbf{Q}W_i^Q, \mathbf{K}W_i^K, \mathbf{V}W_i^V)$$

We add position encodings (depending on the time step value of each ET) to \mathbf{Q} , \mathbf{K} and \mathbf{V} . This is followed by multiheaded relative attention between our keys, values and query as described in Equation 4.3. For each head, we perform a scaled dot-product attention (Vaswani

et al., 2017b)(Equation 4.1) and a form of relative attention¹, i.e. taking a mean of the relation scores and attention scores before normalizing with softmax and multiplying with values (Equation 4.2).

In these equations, d_k is the dimension of the key, W_i^Q, W_i^K, W_i^V are the query, key and value projection matrices, τ is the number of heads and W^O is the linear projection that combines the heads. The output from Equation 4.3 is fed to a positionwise fully-connected feedforward network. We employ a residual connection (He et al., 2016) followed by layer normalization (Ba et al., 2016) before and after the feedforward network. The resulting encoding is then linearly projected and softmax is applied to get the prediction of the statement for line $t + 1$ of p_g . We see that our model resembles one layer of the transformer encoder block (Vaswani et al., 2017b). Since the keys and query come from different sources, we refer to our model as a *cross* aggregator. Like standard transformers, we can stack multiple blocks of CA. However, since we are operating on a low timeout (5s), we opted for a simple network consisting of only one layer. Details of model parameters can be found in Appendix D.3.

Obtaining V: For a key corresponding to an ET consisting of the PE solution p_m and having step index t , the value is associated with the statement vector (one-hot vector of size $= n_s$) for step $t + 1$ of p_m . Putting together the statement vectors for all execution tuples that are part of \mathbf{e}_{keys} , we get a tensor $\mathbf{p}_{\text{values}}$ of size $L \times n_s$. Embedding each entry in this tensor using an embedding layer F_γ gives us $\mathbf{V} = \mathcal{F}(\mathbf{p}_{\text{values}})$ of size $L \times Z$. This is then fed as input to the model described above. The output from the model is then linearly projected to give the logits for the statement predictions $\in \mathbb{R}^{n_s}$ for step $t + 1$ of the global program p_g . In addition to the statement predictions, we can also obtain the operator predictions $\in \mathbb{R}^{n_o}$, starting from the operator vector (one-hot vector of size $= n_o$) and following a process similar to the statements, except that we use a different embedding and final projection layer. The right of Figure 4.1 shows an example of how a query (top) is combined with keys, values and relation scores (bottom) for our model.

4.3.3. Training

The two main components of N-PEPS, the PE Searches module and the Cross-Aggregator, are trained separately. To create samples for training the PE model, we take one data point ($X = \{r_i\}_{i=1}^N$ and p_g) from the GPS approach and create N data points out of it. Since we do not have supervision for the PE solutions, for every example r_i in X , we use p_g as a proxy for ground-truth PE solution. We believe that using p_g as proxy supervision even though not being entirely correct, forces the PE search component to avoid overfitting to a single example and hence is more likely to produce PE solutions that

¹Note that our formulation of relative attention differs from the formulation used in Shaw et al. (2018); Hellendoorn et al. (2020), where the relation scores are added either to the query or values.

generalize to examples outside the ones given as specification (see Appendix D.2 for more details).

For training the CA module, we generate data points that we call *aggregator instances*. Each aggregator instance consists of X , a list Y of tuples of PE solutions p_i and corresponding PE solution scores u_i , and global program p_g . The p_i 's and u_i 's are generated via CAB from a trained PE model (more details on how they are generated in Appendix C.2). Given X and Y as input, the objective is to learn the parameters of the CA module such that the output is the line-wise statement and operator predictions corresponding to p_g . The net loss at step t is the sum of two terms: (a) a cross entropy loss between the predicted statement \hat{s}^t (obtained from CA) and the actual statement vector s^t (obtained from p_g^t); (b) a cross entropy loss between the predicted operator \hat{o}^t and the actual operator vector o^t . Like PCCoder, the operator loss is used as an auxiliary loss to improve training. Note that for each aggregator instance, since we have X and Y to begin with, we need to compute the keys and values only once. However, the computation of query has to be done at each step of the global program execution. While training, since p_g is known, we can use teacher forcing and increase efficiency by batching, where an element in the batch corresponds of one step of execution of p_g .

4.3.4. Inference

The process of inference in PEPS is the same as in PCCoder (see Section 4.2.1), except that in addition to the contribution from GPS, we add another term that accounts for the contribution from CA. The contribution from GPS is obtained by using a *GPS model* that is trained as in standard PCCoder. The net value of the predicted statement at step t is then obtained by taking a weighted contribution from the statement predictions from the trained GPS model $\hat{s}_{1-\alpha}^t$ and the statement prediction from the trained CA module \hat{s}_α^t . For predicting the drop vector \hat{d}^t , we take contributions only from GPS. When $\alpha = 0$, our approach becomes equivalent to GPS.

$$\begin{aligned}\hat{s}^t &= \alpha * \hat{s}_\alpha^t + (1 - \alpha) * \hat{s}_{1-\alpha}^t \\ \hat{d}^t &= \hat{d}_{1-\alpha}^t\end{aligned}\tag{4.4}$$

We perform CAB until we find a global solution or we exceed the specified timeout. The right part of Figure 4.1 illustrates an example of the steps involved in synthesizing step 2 of p_g .

4.4. Experiments and Results

Following prior work (Balog et al., 2016; Zohar & Wolf, 2018)², we generate programs for training and testing, with each program consisting of five IO example pairs, i.e., $N = 5$. The data generation process ensures that there is no overlap between the training and test programs, with programs being functionally non-equivalent to programs of shorter or equivalent lengths (see Appendix 4.C.1 for more details). In the first set of experiments (henceforth referred to as **E1**), we generated 105036 training programs of length up to 4 (i.e., consisting of lengths 1, 2, 3, 4). For the second set of experiments (henceforth referred to as **E2**), we generated 189328 training programs of length up to 12. 10% of the training data was used for validation. To ensure robustness and reproducibility of results, for each method, we carry out experiments over 30 different test splits, where each split contains 500 programs of a specific length. For E1, we generate test programs of length 4, and for E2 we generate programs of lengths 5, 8, 10, 12 and 14. We learn separate GPS models and PE models for E1 and E2. All GPS results were obtained using the original PCCoder implementation¹. A notable difference in our experiments from PCCoder (Zohar & Wolf, 2018) is that we consider a short timeout of 5s (in both E1 and E2, *for all methods*), instead of 5000s and 10000s. This choice is representative of the timeout required for satisfactory user experience in program synthesis systems used in real-world interactive use-cases (such as FlashFill feature in Microsoft Excel (Gulwani, 2011)). Given a particular timeout value, we record the *Success Ratio*, which is the fraction of test samples that succeeded in finding a global solution.

4.4.1. Initial Experiment: Analysis of PE Solutions

The promise of PEPS is rooted in the assumption that it is much easier to find PE solutions than finding a global solution. In order to test this hypothesis and get an idea of the types of cues discovered by PEPS, we performed a set of analysis experiments using data from E1. Using the trained PE model to find PE solutions, we consider two variants. The first variant called **tot(k)** is similar to the strategy of finding PE solutions that we use in PEPS (Section 4.3.1), where we search for PE solutions sequentially (in the order of examples in X) until the discovered PE solutions taken together satisfy k examples in X (where $k \leq 5$). This helps us understand how much the coverage ($= k$) from a list of PE solutions can be. In the second variant called **ind(k)**, we record success by searching for PE solutions sequentially until we find an individual PE solution that satisfies k out of N examples in X . Here, the success ratio helps us assess how good a single PE solution is. In other words, can we rely solely on individual solutions or do we need to aggregate them?

²We used the implementation from PCCoder (Zohar & Wolf, 2018), at <https://github.com/amitz25/PCCoder> (MIT License) for data generation and obtaining results for PCCoder.

For the initial experiment, since no aggregation is done, we divide the timeout of 5s evenly amongst PE searches, i.e., each PE search gets $\frac{1}{5} \times 5 = 1s$ as the timeout value. For GPS, we use the trained GPS model with a timeout of 5s.

Table 4.1. Success ratio of GPS, **ind** and **tot** for different values of k for test programs of length 4.

GPS	ind(1)	ind(2)	ind(3)	ind(4)	ind(5)	tot(1)	tot(2)	tot(3)	tot(4)	tot(5)
77.0	99.2	95.4	85.4	70.4	43.2	99.2	97.6	97.0	94.8	82.4

Table 4.1 gives the results of these analysis experiments on one of test splits for programs of length 4. Note that in **tot**, we are not aggregating the cues to find a global program. Hence, the value given under **tot(5)** is not directly comparable to GPS. We make a few observations. First, the success ratio increases with decreasing value of k . Therefore, as speculated, it is easier to find solutions that satisfy examples partially. Second, we see that even though the numbers for **ind** are encouraging, they are less than the corresponding values (for same k) for **tot**. This suggests that aggregating PE solutions is better than dealing with them individually. Third, the success ratio of **tot(5)** is better than GPS. This suggests there is potential in thinking of an architecture that can learn to combine these solutions. Even for cases where we couldn't find PE solutions that satisfy all 5 examples, we can hope to make use of the rich partial cues (indicated by high success ratios) coming from **tot(k < 5)**.

4.4.2. Methods

In addition to the standard GPS baseline (PCCoder (Zohar & Wolf, 2018)), we experimented with three ablation baselines that represent simple ways of aggregating the PE solutions without making use of the program state. Hence, they help us understand the role of PE cues alone. These baselines are: (i) **Sum-PEPS**: Replacing the contribution from CA module in Equation 4.4 by a module that combines the PE solutions by taking the sum of all PE one-hot statement vectors; (ii) **Mean-PEPS**: Same as (i) except that sum is replaced by mean; (iii) **Mean-PEPS+ \mathcal{U}** : Same as (ii) except that the one-hot PE statement vectors are multiplied by their corresponding solution scores before taking the mean. To understand the benefit of aggregating with our proposed CA architecture on top of the value brought by the PE cues, we experimented with the following variations: (i) **N-PEPS**: Our neural model of PEPS described in Section 4.3.2 with \mathcal{U} being a zero tensor; (ii) **N-PEPS+ \mathcal{U}** : Same as (i) but with \mathcal{U} included. Complete details of hyperparameters for all methods can be found in Appendix 4.D.

Model	Success Ratio
PCCoder (Zohar & Wolf, 2018)	77.75 ± 0.38
Sum-PEPS	82.71 ± 0.32
Mean-PEPS	82.68 ± 0.33
Mean-PEPS+ \mathcal{U}	82.70 ± 0.32
N-PEPS	86.22 ± 0.25
N-PEPS+ \mathcal{U}	87.07 ± 0.28

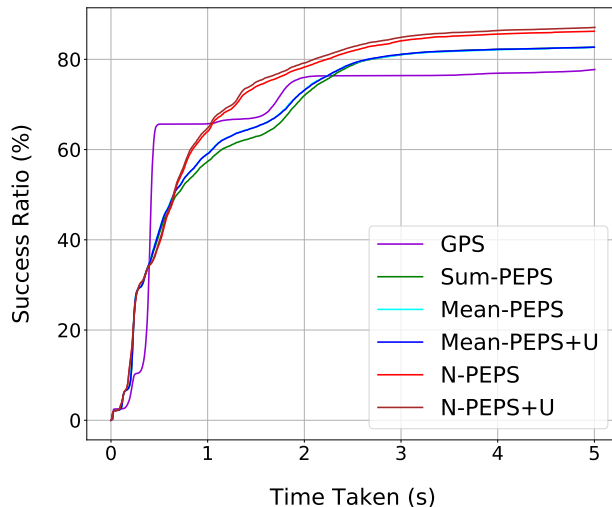


Fig. 4.3. Results for E1: (*Top*) Success Ratio with standard error. The top row in the table corresponds to GPS; (*Bottom*) Success Ratio vs. time taken.

4.4.3. Results

For each test data point, we record either a success or failure (based on whether within 5s, we find a global solution or not) and the actual time taken to find a global solution. As described in Section 4.3.1, for all PEPS methods, we start by allocating a PEPS timeout value that is less than 1s ($= \frac{1}{N} \times \text{total timeout}$). We sum the actual time (\leq PEPS timeout) taken for finding individual PE solutions. The residual times (if any) left out is added and used for aggregation and global inference. Note that PEPS timeout and α are treated as hyperparameters that are chosen using the validation set.

To provide a fair comparison across all methods, each test split is run using a single core and single CPU thread with a timeout of 5s. To account for variability across machines, we chose to run a test split on a machine chosen randomly from a collection of 7 machines of similar configuration³ (Google Cloud instances with 120GB RAM each). We report standard error across the 30 test runs.

³We additionally verify that different runs on the same machine produce similar results (Appendix 4.D.6)

Table 4.2. Results for E2: Success Ratio with standard error for all models.

Model	Length = 5	Length = 8	Length = 10	Length = 12	Length=14
PCCoder (Zohar & Wolf, 2018)	70.91 ± 0.35	44.17 ± 0.45	28.18 ± 0.33	19.69 ± 0.34	14.71 ± 0.23
Sum-PEPS	76.45 ± 0.33	43.4 ± 0.56	28.96 ± 0.27	20.94 ± 0.32	15.67 ± 0.32
Mean-PEPS	75.79 ± 0.31	44.42 ± 0.51	29.55 ± 0.29	21.45 ± 0.27	16.35 ± 0.27
Mean-PEPS+ \mathcal{U}	75.99 ± 0.32	44.49 ± 0.52	29.75 ± 0.25	21.74 ± 0.30	16.45 ± 0.33
N-PEPS	79.18 ± 0.31	47.23 ± 0.49	32.3 ± 0.34	23.34 ± 0.28	17.35 ± 0.31
N-PEPS+ \mathcal{U}	79.19 ± 0.30	46.31 ± 0.61	31.84 ± 0.36	22.71 ± 0.28	16.68 ± 0.21

Main result: The top and bottom left parts of Figure 4.3 show the success ratio and success ratio vs. time taken (average of the actual time taken) plots, respectively, for test programs of length 4 when trained on programs up to length 4 (E1). Table 4.2 shows the success ratio for test programs of lengths 5, 8, 10, 12, and 14 when trained on programs up to length 12 (E2). In both these settings, we observe that the performance of the ablation baselines is better than GPS, illustrating the promise in the quality of PE solutions. When we use our CA module to aggregate these cues instead, we see that the performance improves even further. We used the default value of $\nu = 11$ used in PCCoder, which means that for programs of length > 8 , certain variables will be dropped from the program state. Also, note that the results for test length 14 represent a case of *length generalization*. We show that in both these scenarios, our proposed method is quite advantageous⁴. In addition, we compare the performance of N-PEPS with GPS for the cases of *intent generalization*, i.e., generalization of the synthesized program to examples other than those given as part of X (Appendix 4.F.2) and when given a longer timeout of 1000s (Appendix 4.F.1). In both these settings, N-PEPS shows superior performance, highlighting its generality.

Attention visualization: Figure 4.4 shows a visualization of attention scores at $t = 2$ and $t = 3$ obtained from our best model under E1 for the example shown in Figure 4.1. This example represents a case from the test set where N-PEPS succeeds in finding a global solution, whereas other methods fail. As can be seen, the actual statement of p_g^2 is DROP c a and our model indeed gives relatively higher attention scores to p_1^2 and p_3^2 , both of which correspond to the same statement. Similarly at $t = 3$, our model gives more attention to $p_3^3 = \text{MAP} (*-1) \text{ d} = p_g^3$.

⁴See Appendix 6.G for success cases of N-PEPS & Appendix 4.F.3 and Appendix 4.F.4 for empirical analysis of synthesized programs.

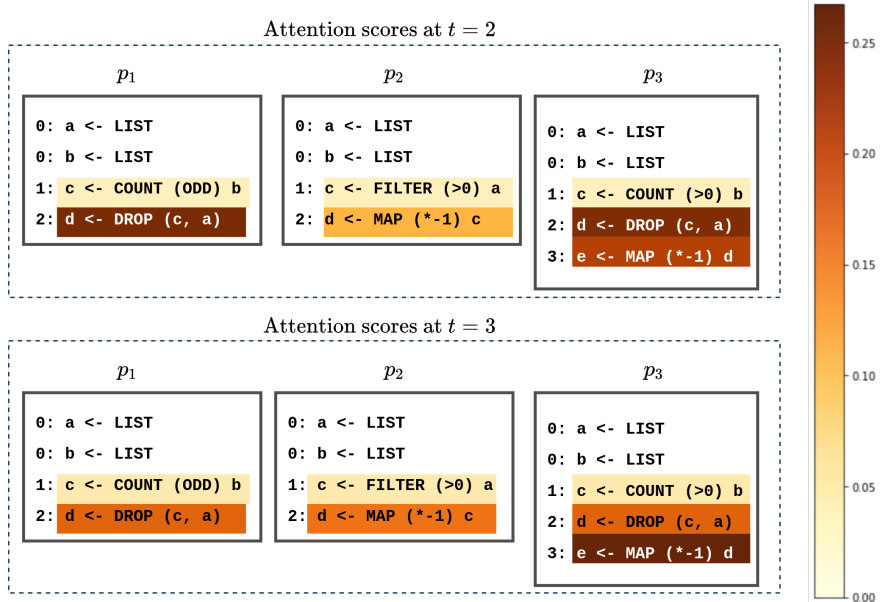


Fig. 4.4. Visualization of attention scores for N-PEPS+ \mathcal{U}

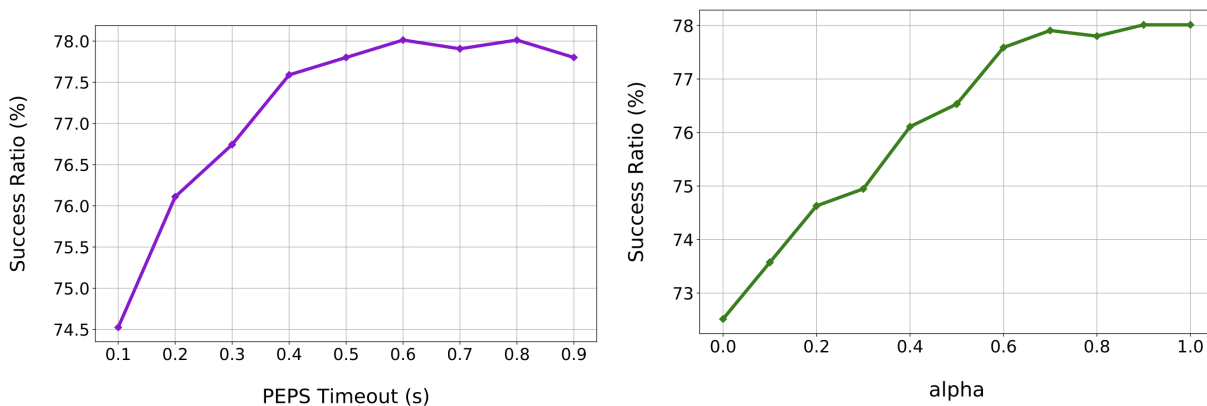


Fig. 4.5. Variation of success ratio with PEPS Timeout (top) and α (below) for N-PEPS

Variation with PEPS timeout and α : There is a non-trivial tradeoff in the division of the total timeout into the time given to find PE solutions and the time given to the CA module. A higher PEPS timeout results in better chances of discovering rich PE cues. This means that there may be less or almost no time needed for aggregation. On the other hand, if we start with a low PEPS timeout, the cues from PE solutions may not be as good, but we have more time to perform aggregation. Also, there is a question of how much contribution should be taken from CA and how much from GPS, which is determined by the value of α . Figure 4.5 analyzes this tradeoff. On left, we show the variation of success ratio with PEPS timeout and on right, we have the variation with α , for the validation set under E2. We see that the performance improves with an increase in PEPS timeout with a slight decrease

towards the end. Also, we see that generally higher values of α are better indicating that the contribution from CA is more important than the contribution from GPS.

Variants of K: In addition to obtaining $\mathcal{H}(\mathbf{e}_{\text{keys}})$ in the way described in Section 4.3.2, we tried two other ways of composing the keys by varying the set \mathcal{S} in the execution tuple against which the PE solutions are executed. In the first variant, PE solution p_m from the list P of M discovered PE solutions is executed against the global set X , i.e., $\mathbf{e}_{\text{keys}} = [(p_m, \{1, 2, \dots, N\}, t)]$ where $p_m \in P$ and $t \in \{0, 1, \dots, |p_m| - 1\}$. We denote this variant as **N-PEPS-PG** (PG = PE-global ET). In the second variant, p_m is executed against the set S_m consisting only of examples indices that p_m satisfies, i.e., $\mathbf{e}_{\text{keys}} = [(p_m, \{j\}, t)]$ where $j \in S_m$. We call this variant as **N-PEPS-PP** (PP = PE-PE ET). Table 4.3 shows the test results of these variants for E1 with and without \mathcal{U} . We see that all the variants perform better than GPS and the three ablation baselines (N-PEPS variant used in Section 4.3.2 was chosen using the validation set). We see a similar trend for E2 (see Appendix 4.E).

Table 4.3. Performance of variants of **K** for E1

Variant	Success Ratio
N-PEPS-PG	85.19 \pm 0.26
N-PEPS-PG+ \mathcal{U}	85.94 \pm 0.26
N-PEPS-PP	85.97 \pm 0.26
N-PEPS-PP+ \mathcal{U}	86.21 \pm 0.27
N-PEPS	86.22 \pm 0.25
N-PEPS+ \mathcal{U}	87.07 \pm 0.28

New operator discovery by N-PEPS: We were interested in determining that while synthesizing the global solution how often does N-PEPS rely on copying statements from the PE solutions and how often does it generate new operators. We studied this trend for the cases when $\alpha < 1.0$, i.e., contributions are taken from both CA and GPS as well as when $\alpha = 1.0$, i.e., contributions are taken only from CA (Appendix 4.G.2). This question is important as it helps us understand the generalization capabilities of CA outside the statements in the PE solutions. We found that CA alone (with $\alpha = 1.0$) is capable of generating new operators. In addition, we found that the new operators are present as part of the nearest neighbors of the PE statements, thereby pointing to an increased likelihood of these being ranked higher in the beam search and hence being present in the global solution (see Appendix 4.G.3 and 4.G.4 for details).

4.5. Related Work

There have been numerous efforts on using deep learning for program synthesis (Balog et al., 2016; Bunel et al., 2018; Devlin et al., 2017b; Kalyan et al., 2018; Devlin et al., 2017a; Lee et al., 2018; Nye et al., 2019; Odena & Sutton, 2020; Parisotto et al., 2016). However, there is less work that uses the execution of partial programs to assist in synthesis. PCCoder (Zohar & Wolf, 2018) is one such work, which we describe in Section 4.2.1. BUSTLE (Odena et al., 2021) reweighs the sub-expressions in bottom-up program synthesis

using the intermediate values obtained by execution of sub-expressions along with property signatures. REPL (Ellis et al., 2019a) executes partial programs using learned policy and value networks. Chen et al. (2018) uses a neural encoder-decoder architecture to generate program tokens conditioned on intermediate states obtained from execution of partial programs. They work with the Karel DSL (Pattis, 1994; Bunel et al., 2018) that contains loops and conditionals, an attribute missing from the DSL which we work with. Therefore, extending N-PEPS for Karel is an interesting future work. Note that all the approaches mentioned above are examples of purely GPS approaches.

Few works use solutions that satisfy examples partially, to aid in program synthesis. Initial motivation for our work comes from FrAngel (Shi et al., 2019), which is a component-wise synthesis system that relies on mining fragments of Java code that satisfy examples partially, given target program function signatures and a list of Java libraries. The mining of fragments as well as combination is done using a set of heuristics and predefined rules with no deep learning involved. Assuming that the user provides IO examples in order of increasing difficulty, Perelman et al. (2014) iteratively refines a program, with the current program satisfying the sequence of IO examples encountered till now. STUN (Alur et al., 2015) extends the CEGIS (Solar-Lezama et al., 2006) approach by providing domain-specific explicit unification operators for combining partial solutions while Alur et al. (2017) uses decision trees for the same. Recently, BESTER (Peleg & Polikarpova, 2020) and later PROBE (Barke et al., 2020) perform bottom-up enumeration of programs in a loop by enumerating all programs that satisfy IO examples partially. This is followed by heuristics-based selection of promising programs. However, as opposed to N-PEPS that automatically learns to aggregate partial solutions producing the global program in one shot, PROBE relies on using these programs to iteratively update the weights of useful productions in their probabilistic grammar using a fixed update rule. This update can be viewed similar to our ablation baselines that do not use the neural network based learned aggregation. The guided-search component of PROBE provides an interesting alternative to finding PE solutions. One way of incorporating this in our top-down setting might be to start with the CAB search as in GPS and then select promising solutions based on evaluating examples on prefixes of programs obtained during the beam search. It may be useful to then aggregate the selected solutions using a neural architecture similar to ours.

4.6. Conclusions and Future Directions

In this work, we propose N-PEPS, where the idea is to break the problem of finding a program that solves all examples into two stages: (a) finding programs that solve a single example (PE solutions) (b) aggregating the PE solutions such that it leads to a program that solves all examples. For aggregation, we propose a neural-network based multi-head attention

architecture (CA module) that utilizes the state of program execution to learn to combine the PE cues. We note that program synthesis systems in general should be deployed with caution for use in real-world applications. Blind trust on these systems can create chances for potential negative impact, as there might be cases where the generated program contains bugs, especially for unseen examples outside the specification. In the future, we want to work with programs that contain loops and conditionals ([Pattis, 1994](#); [Chen et al., 2018](#); [Bunel et al., 2018](#)). Another interesting research direction would be to explore the interaction of N-PEPS with out-of-distribution generalization settings like compositional generalization ([Lake & Baroni, 2018](#)).

Appendix for the first article

4.A. Details of PCCoder ([Zohar & Wolf, 2018](#))

We provide our version of the training and inference algorithms and description of modules used in PCCoder ([Zohar & Wolf, 2018](#)) next. Note that the terminology used in PCCoder differs from what we have used here.

4.A.1. Training and Inference Algorithms of PCCoder

Algorithm 1 Train (GPS)

Require: $p_g = [p_g^t]_{t=1}^T =$ ground-truth program with T lines
Require: $\nu = \max \#$ allowed variables = memory-size
Require: $X = \{(x_i, y_i)\}_{i=1}^N = \{r_i\}_{i=1}^N =$ set of N IO examples

- 1: $\mathcal{X}^0 = [r_i]_{i=1}^n$ \triangleright *Initial State*
- 2: **for** t in $\text{range}(T)$ **do**
- 3: \triangleright *Obtain ground truth*
- 4: $s^t, \hat{o}^t, d^t = R(p_g^t, [p_g^t]_{j \geq t}, \mathcal{H}^{t-1})$
- 5: $\mathcal{H}^{t-1} = H_\theta(\mathcal{X}^{t-1})$ \triangleright *Obtain current state embedding*
- 6: $s^t, \hat{o}^t, \hat{d}^t = W_\phi(\mathcal{H}^{t-1})$ \triangleright *Obtain predictions*
- 7: \triangleright *Calculate loss and update parameters*
- 8: $\mathcal{L} = \text{CE}(s^t, \hat{s}^t) + \text{CE}(\hat{o}^t, \hat{o}^t) + \sum_{j=1}^{\nu} \text{BCE}(d^{t,j}, \hat{d}^{t,j})$
- 9: $\theta \leftarrow \theta - \alpha * \nabla_\theta \mathcal{L}$
- 10: $\phi \leftarrow \phi - \alpha * \nabla_\phi \mathcal{L}$
- 11: \triangleright *Randomly chose an index to drop*
- 12: $d^{t'} = \text{random_choice}(d^t)$
- 13: \triangleright *Execute p_g^t to get updated state*
- 14: $\mathcal{X}^t = \text{DropExec}(p_g^t, \mathcal{X}^{t-1}, d^{t'}, \nu)$

15: **procedure** **DROPEXEC**(p, x, d', ν)

- 16: $l = \text{get_num_vars}(x)$
- 17: $N = \text{shape}(x)[0]$ \triangleright *# IO examples*
- 18: **for** i in $\text{range}(N)$ **do**
- 19: $x_i = x[i]$
- 20: \triangleright *Execute p against x_i to obtain result c_i*
- 21: $c_i = \text{Execute}(p, x_i)$
- 22: **if** $l > \nu$ **then** \triangleright *Need to drop a variable*
- 23: $x_i[d'] = c_i$
- 24: **else**
- 25: $x_i.append(c_i)$
- 26: $l = l + 1$
- 27: $\text{set_num_vars}(x, l)$
- 28: **return** x \triangleright *return the updated state*

Algorithm 2 Inference (GPS)

- 1: $\mathcal{X}^0 = [r_i]_{i=1}^n$
- 2: **while** time < timeout **do** \triangleright CAB outer loop
- 3: \triangleright *Initial Beam: (state, program, prob)*
- 4: $B = [(\mathcal{X}^0, [], 1.0)]$
- 5: $\triangleright p_g = \text{global solution}$
- 6: $p_g = \text{BeamSearch}(B)$ \triangleright CAB inner loop
- 7: **if** $p_g == \text{FAILED}$ **then**
- 8: beam_size *= 2; beam_expansion_size += 10

9: **procedure** **BEAMSEARCH**(B)

- 10: **while** beam search conditions are met **do**
- 11: $B' = []$ \triangleright *new beams*
- 12: \triangleright *For each parent node*
- 13: **for** $(b, (\mathcal{X}_b^{t-1}, p_b^{t-1}, s_b^{t-1}))$ in $\text{enum}(B)$ **do**
- 14: **if** is_solution(\mathcal{X}_b^{t-1}) **then**
- 15: **return** p_b^{t-1}
- 16: $\mathcal{H}_b^{t-1} = H_\theta(\mathcal{X}_b^{t-1})$
- 17: $\hat{s}_b^t, \hat{d}_b^t = W_\phi(\mathcal{H}_b^{t-1})$
- 18: \triangleright *sort \hat{s}_b^t by decreasing probability*
- 19: $\hat{s}_b^t = \text{sort}(\hat{s}_b^t)$
- 20: \triangleright *choose argmax of \hat{d}_b^t to drop*
- 21: $d_b^t = \text{argmax}(\hat{d}_b^t)$
- 22: \triangleright *Expand the parent node*
- 23: **for** \tilde{s}_b^t in $\hat{s}_b^t[: \text{expansion_size}]$ **do**
- 24: \triangleright *get statement id for the prob entry*
- 25: $p_b^t = \text{prob_to_stat}(\tilde{s}_b^t)$
- 26: \triangleright *get updated memory*
- 27: $\mathcal{X}_b^t = \text{DropExec}(p_b^t, \mathcal{X}_b^{t-1}, d_b^t, \nu)$
- 28: $p_b^{t-1}.append(p_b^t)$ \triangleright *updated program*
- 29: $s_b^{t-1} = s_b^{t-1} * \tilde{s}_b^t$ \triangleright *updated probability*
- 30: $B'.append((\mathcal{X}_b^t, p_b^{t-1}, s_b^{t-1}))$
- 31: \triangleright *sort beams by decreasing probability*
- 32: $B' = \text{sort}(B')[: \text{beam_size}]$
- 33: $B = B'$
- 34: **return** FAILED \triangleright *if no solution found during beam search, return Failed solution*

4.A.2. Description of Modules in PCCoder

The inputs to a program can either be an integer or an array of integers of maximum length 20. The integers can be in the range [-256, 255]. There can be a maximum of three input arguments to a program. There are 1298 statements and 38 operators in the DSL, i.e., $n_s = 1298$ and $n_o = 38$. Execution of a line in the program returns exactly one variable. Below, we describe the blocks present at different stages of PCCoder:

- **State Representation:** For each of the N IO examples, the corresponding inputs are taken and all entries are made positive by subtracting the minimum integer value under the DSL (i.e. -256) from them. For shorter inputs, NULL values up to length

$q = 20$ are padded. Then two bits indicating the type of input (list or int) are appended at the beginning of this representation. Therefore, each variable is now represented as a vector of size $q + 2 = 22$. There can be a maximum of ν input variables and one output variable (corresponding to the output of the IO example given). If there are less than ν variables, NULL values are padded to make it uniform. An account of the actual number of variables (i.e. number of filled slots) present in the state is also kept, denoted by l . The output of this stage is an array of size $N \times (\nu + 1) \times (q + 2)$. This forms the *state* \mathcal{X} .

- **State Embedding (H_θ):** The output obtained in the previous step is then passed through a series of neural network blocks to obtain a *state embedding* \mathcal{H} . An embedding layer projects each entry in the state (excluding the type bits) into an vector of size $e = 20$, giving us a tensor of size $N \times (\nu + 1) \times (q * e + 2)$. This is then passed through a linear layer of size 56 and then reshaped to obtain a tensor of size $N \times (\nu + 1) * 56$. It is then passed through a dense block to obtain a tensor of size $N \times Z$ where $Z = 256$. This pre-pooling version is what we refer to as the representation of slots in Section 3.2. An average pooling of these representations across all N examples gives a vector of size 1×256 that forms the state embedding.
- **Predicting quantities of interest in next line (W_ϕ):** The state embedding obtained above is projected into three linear heads of size 1298, 38 and ν followed by softmax, softmax and sigmoid, respectively which gets us the statement, operator and drop probabilities.
- **DropExec:** In the *DropExec* module, after a statement is executed against the variables present in the slots in the state \mathcal{X}^0 , we get new values of resulting variables. If the actual number of variables l exceeds ν , one of the existing variables is dropped based on the drop vector. If not, this new variable is simply appended to the existing variables by filling the next slot in the memory. This updated state is then passed through H_θ to get the updated state embedding \mathcal{H}^1 . This completes one step of execution of the program.

For the next steps, we repeat the last two steps mentioned above till we reach the end of the program. See Figure 2 for an illustration of the process at $t = 2$.

4.B. Sample Cases

Below we provide two sample cases where GPS fails and our N-PEPS model (for E2) succeeds in finding a global solution. For each sample case, we show the synthesized global solution on the left, the set of IO examples in the center and the discovered PE solutions along with PE solution scores in the right. We also report the actual time taken to find the solutions. Note that for the second case, even though the global ground-truth test program

is of length 8, N-PEPS discovers a global solution of shorter length.

<p>Global Solution: (Time taken to find=3.21s) a ← LIST b ← ZIPWITH (+) a a c ← TAIL b d ← TAKE c b e ← COUNT (>0) d f ← TAKE e d g ← COUNT (>0) f h ← TAKE g f i ← TAKE g h j ← HEAD i k ← TAKE j i l ← TAKE j k m ← TAKE j k n ← TAKE j k o ← REVERSE n</p>	<p>IO examples: #1. Input: [4, 5, 6, 2, 6, 2, 1, 6, 1, 4, 2, 5, 6, 3, 2, 2] <i>Output:</i> [4, 12, 10, 8] #2. Input: [3, 2, 5, 0, 3, 2, 3, 0, 4, 1, 0, 2, 3, 0, 3, 4] <i>Output:</i> [6, 0, 10, 4, 6] #3. Input: [1, 1, 4, 0, 0, 0, 0, 5, 0, 5, 3, 5] <i>Output:</i> [2, 2] #4. Input: [4, 4, 1, 4, 4, 1, 4, 2, 2, 1, 3, 4] <i>Output:</i> [4, 8, 2, 8, 8, 2, 8, 8] #5. Input: [4, 1, 1,, 3, 3, 1, 4, 0, 4, 2, 4] <i>Output:</i> [8, 2, 6, 6, 2, 2, 8]</p>	<p>PE Solutions: p₁ : Time taken to find=0.2s Satisfies #1, #4 ($u_1 = 0.2$) a ← LIST b ← ZIPWITH (+) a a c ← TAIL b d ← TAKE c b e ← REVERSE d p₂ : Time taken to find=0.34s Satisfies #2, #3, #4, #5 ($u_2 = 0.8$) a ← LIST b ← ZIPWITH (+) a a c ← HEAD b d ← TAKE c b e ← COUNT (>0) d f ← TAKE e d g ← REVERSE f</p>
--	--	--

Global Solution:

(Time taken to find=2.98s)

a ← LIST
b ← INT
c ← MAXIMUM a
d ← TAKE c a
e ← TAIL c
f ← TAKE b c
g ← ZIPWITH (+) f f
h ← MAP (+1) g
i ← TAKE e h

IO examples:

#1. Input:

[1, 0, 3, 3, 3], 35

Output:

[3, 1, 7]

#2. Input:

[6, 3, 3, 1, 2, 2, 0, 3, 8, 7], 50

Output:

[13, 7, 7]

#3. Input:

[1, 5, 6, 10, 5, 11, 7, 0, 7, 11,
10, 9, 4], 78

Output:

[3, 11, 13, 21, 11, 23, 15, 1, 15,
23]

#4. Input:

[12, 4, 11, 11, 4, 7, 12, 11, 11,
10, 5, 8, 9, 8], 166

Output:

[25, 9, 23, 23, 9, 15, 25, 23]

#5. Input:

[4, 0, 5, 5, 1, 1, 1, 1], 126

Output:

[9]

PE Solutions:

p₁ : Time taken to find=0.17s
Satisfies #1, #4, #5 ($u_1 = 0.6$)

a ← LIST
b ← INT
c ← TAIL a
d ← TAKE c a
e ← ZIPWITH (+) d d
f ← MAP (+1) e

p₂ : Time taken to find=0.37s
Satisfies #1, #5 ($u_2 = 0.4$)

a ← LIST
b ← INT
c ← TAKE b a
d ← TAIL c
e ← ACCESS d c
f ← TAKE e c
g ← ZIPWITH (+) f f
h ← MAP (+1) g

p₃ : Time taken to find=0.8s
Satisfies None ($u_3 = 0.0$)
FAILED

p₄ : Time taken to find=0.17s
Satisfies #1, #4, #5 ($u_4 = 0.6$)

a ← LIST
b ← INT
c ← TAIL a
d ← TAKE c a
e ← ZIPWITH (+) d d
f ← MAP (+1) e

p₅ : Time taken to find=0.17s
Satisfies #1, #4, #5 ($u_5 = 0.6$)

a ← LIST
b ← INT
c ← TAIL a
d ← TAKE c a
e ← ZIPWITH (+) d d
f ← MAP (+1) e

4.C. Data Generation

4.C.1. Generation of Training and Test set

Similar to the data generation process described in Balog et al. (2016); Zohar & Wolf (2018) and using the implementation from PCCoder ⁵, we generated programs for training and testing where each program consists of five input-output examples. The process starts by generating training programs iteratively starting from length 1 till the maximum length specified (4 and 12 in our case). For each length, first a program of that length is generated followed by generating corresponding IO examples which correspond to that program. This is followed by checking for functional non-equivalence of that program with all generated programs so far (i.e., programs of length less than or equal to the current length). Functional non-equivalence means that given a set of IO examples, we can't have a program of length x that satisfies the set of examples when we already have a program of length less than or equal to x in our dataset that satisfies the same set of examples. If the program is found functionally equivalent to any other programs, it is discarded, else it is added to the training set.

Once the generation of training set is complete, we proceed to generating the test set. Given a test length, we generate a program of that length followed by generating the corresponding IO example pair. In addition to checking for functional non-equivalence with all programs in the test set so far, we also test for functional non-equivalence with every program in the training set. This makes sure that there is no overlap between the training and test sets and all the programs are functionally non-equivalent to each other. We have two experimental settings: (a) **E1**: Training set = 105036 programs till length 4 and 30 test sets of 500 programs each of length = 4; (b) **E2**: Training set = 189328 programs of length up to 12 and 30 test sets of 500 programs each of lengths = 5, 8, 10, 12 and 14. In each setting, 10% of the training data was used for validation. Figure 4.6 shows the distribution of training programs with length in both the settings. There are less programs of longer lengths as there is high probability that they end up being discarded because a functionally equivalent program of shorter length was found.

⁵<https://github.com/amitz25/PCCoder> (MIT License)

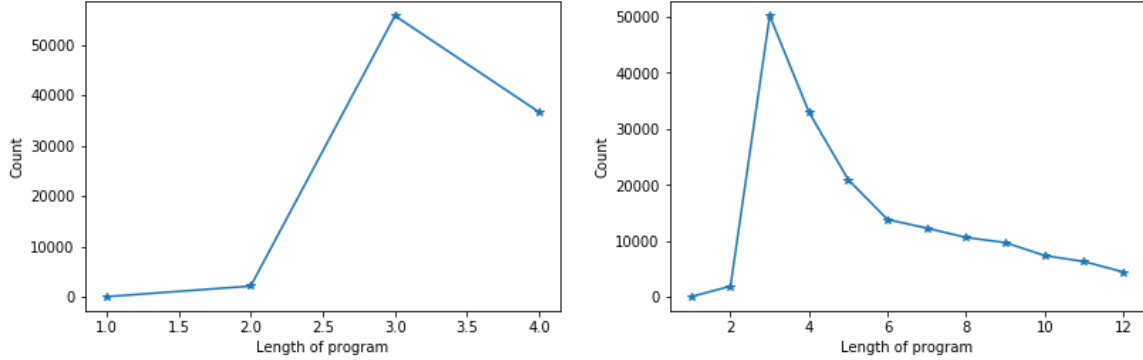


Fig. 4.6. Distribution of training programs: (*Left*) For E1; (*Right*) For E2

4.C.2. Generation of Aggregator Instances

An aggregator instance consists of the set of IO examples X , a list Y of PE solutions p_i along with the corresponding PE solution scores u_i , and the corresponding global program p_g . To create aggregator instances, for each data point (given X and p_g) in the original training dataset (generated as described in 4.C.1), we generate PE solutions and PE solution scores using the PE Searches module. For generating the PE solutions, we need to choose a value of PEPS timeout. We generated aggregator instances with PE solutions obtained using the trained PE model, in three ways: (a) one aggregator instance with a fixed PEPS timeout of 0.5s; (b) two aggregator instances with PEPS timeout randomly chosen from $[0.1s, 0.2s, \dots, 0.9s]$; (c) three aggregator instances each with PEPS timeouts of 0.4s, 0.5s and 0.6s, respectively. These options will lead to the same, twice and thrice the number of data points present in the original training set. We chose to omit a sample from being part of training data formed from aggregator instances if either (a) An aggregator instance consists of a PE solution that satisfies all examples (i.e., $u_i = 1.0$) or (b) When we fail to get any PE solution (i.e., all $u_i = 0.0$). We can then generate data which omits both (a) and (b). The datasets formed after removing these aggregator instances will be referred to as $\mathbf{D}_{0.5}$, \mathbf{D}_{rand} and $\mathbf{D}_{0.5 \pm 0.1}$ for cases (a), (b) and (c), respectively. In addition, within each aggregator instance, we can chose to discover all 5 PE solutions (*all*) or alternatively find a list of M PE solutions where $M \leq 5$ such that taken together these satisfy all examples in X (*tot*). Therefore, in total we have 12 different variations (3 PEPS timeouts \times 2 inclusion conditions \times 2 modes of discovering PE solutions) of training datasets which can be used for training the cross-aggregator. We follow a similar procedure to generate variations of corresponding validation datasets which are used to select the hyperparameters and early-stopping for training with each dataset variation. Table 4.4 gives the training and validation data statistics (note that the 2 modes for discovering PE solutions will affect the content of

Table 4.4. Aggregator data statistics for E1 and E2

Dataset	# of samples	
	E1	E2
$D_{0.5}^{train}$	16408	82102
$D_{0.5}^{val}$	2132	9492
D_{rand}^{train}	41707	176235
D_{rand}^{val}	5365	20035
$D_{0.5\pm 0.1}^{train}$	49116	248972
$D_{0.5\pm 0.1}^{val}$	6396	28734

a single aggregator instance, but the number of aggregator instances will remain the same in both cases).

4.D. Experimental details

All our implementations are based on PyTorch (Paszke et al., 2019) version 1.4.0. The training for the GPS and PE models and the CA was done using Tesla V100 (16GB) and Tesla P100 (16GB) GPUs on Google Cloud instances with 120GB RAM.

4.D.1. Parallel Execution for PEPS

In the current formulation, we find PE solutions sequentially. However, the running time can be reduced further by finding PE solution in parallel as the process of finding PE solution i is dependent only on the IO example r_i . So, instead of finding PE solutions one by one, we can find PE solutions for all examples in parallel and then check whether the PE solution p_i satisfy any other example from X apart from r_i . The total time for PEPS can then be thought of $\max(\text{time taken to find a PE solution that satisfies } r_i) + \text{time taken to aggregate}$. However, one could argue that PCCoder can also employ more threads in parallel to speed up their search. Therefore, for a fair comparison with PCCoder, we decided to find PE solutions sequentially where we evaluate both N-PEPS and PCCoder on a single CPU thread (with no parallel computations). However, when being deployed for an application where comparisons with other methods are not required, N-PEPS can significantly boost the speed up by searching for PE solutions in parallel in a way suggested above.

4.D.2. Training the GPS and PE models

For each experimental setting, we used the training set (generated as described in 4.C.1) as it is for training the GPS model. For training the PE model, we created five entries out of a single training data point such that a modified entry has a single IO example and the

corresponding program is the same across all five entries = program in the data point for GPS. Since we don't have supervision available for PE solutions, we chose p_g to serve as a proxy for ground-truth of these PE solutions. Another way of creating this supervision would have been to perform separate PE searches for each example and recording the discovered PE solution as ground-truth. However, this procedure would have required the selection of a specific PE timeout. We didn't have any good idea of how to select this value as it would have influenced the generated PE solutions, hence the supervision itself. Also, we didn't know what would have been the best supervision to use for cases where the PE search fails to find a solution. We believe that using p_g as proxy supervision even though not being entirely correct, forces the PE search component to avoid overfitting to a single example and hence is more likely to produce PE solutions that exhibit intent generalization (generalization to examples outside the ones given as specification).

The number of training points for PE model = 5 * number of training data points for GPS. The corresponding validation split was used to select hyperparameters. The selected hyperparameter values were:

- *GPS model*: learning rate = 0.001; batch size = 32 for E1 and 100 for E2.
- *PE model*: learning rate = 0.001; batch size = 100 for E1 and 256 for E2.

For both settings $\nu = 11$. This means that the state has slot for storing 7 intermediate variables, 3 slots for input variables (there can be a maximum of 3 input arguments to a program) and an additional slot for storing the output. This means that for E2, dropping will happen for programs of length greater than 8. We used Adam (Kingma & Ba, 2015a) optimizer with a learning rate scheduler that decayed the learning rate by 0.1 every 4 epochs. We used the validation set for early-stopping. Let's call the learned PE modules as $H_{\theta_{pe}}$ and $W_{\phi_{pe}}$ whereas, the corresponding GPS modules to be H_{θ_g} and W_{ϕ_g} .

4.D.3. Training the Cross Aggregator

For both E1 and E2, we train our cross aggregator (CA) module using the variants of keys mentioned in Section 3.2 and Section 4.3. For N-PEPS-PG, we use H_{θ_g} to obtain state embeddings that forms the keys, whereas for N-PEPS and N-PEPS-PP we used $H_{\theta_{pe}}$. For faster convergence, we initialize the statement and operator heads with the corresponding statement and operator linear heads from W_{ϕ_g} . We tried finetuning the parameters of H , but it didn't result in significant difference in training performance. Hence, we decided to leave the parameters of the H module unaltered during training. As mentioned in 4.C.2, we tried both *all* and *tot* ways of discovering PE solutions while training. In equations 1, 2 and 3 in Section 3.2, the projection matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_q}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$, $W^O \in \mathbb{R}^{\tau d_v \times d_{model}}$. For the multihead relative attention, we used $d_k = d_q = d_v = 64$, $\tau = 8$ and $d_{model} = 256$. A dropout value of 0.1 was used while training.

4.D.4. Details of Training Hyperparameters

We tried different values of learning rates, optimizers, learning rate schedulers, datasets and the PE discovery options. We tried three types of learning rate schedulers⁶: (a) **cosine**: `torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=10, eta_min=0)`; (b) **cosinewarm**: `torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer, T_0=10)`; (c) **reduceonplateau**: `torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min')` where `optimizer = Adam, SGD`. Below we provide the hyperparameter configuration for the best models chosen using the validation set.

Table 4.5. Hyperparameter values for training the CA. lr= learning rate, lrs = learning rate scheduler, o=optimizer

Model	Hyperparameters	
	E1	E2
N-PEPS-PP	$D_{0.5}$, <i>all</i> , lr=1e-4, o=Adam, lrs=cosine	$D_{0.5\pm 0.1}$, <i>tot</i> , lr=1e-4, o=Adam, lrs=reduceonplateau
N-PEPS-PP+ \mathcal{U}	D_{rand} , <i>all</i> , lr=1e-4, o=SGD, lrs=cosinewarm	D_{rand} , <i>all</i> , lr=1e-4, o=SGD, lrs=cosinewarm
N-PEPS-PG	D_{rand} , <i>tot</i> , lr=1e-4, o=SGD, lrs=cosine	$D_{0.5}$, <i>all</i> , lr=1e-4, o=SGD, lrs=cosine
N-PEPS-PG+ \mathcal{U}	D_{rand} , <i>all</i> , lr=1e-4, o=SGD, lrs=cosinewarm	$D_{0.5}$, <i>all</i> , lr=1e-4, o=Adam, lrs=reduceonplateau
N-PEPS	D_{rand} , <i>all</i> , lr=1e-4, o=SGD, lrs=cosine	D_{rand} , <i>all</i> , lr=3e-4, o=Adam, lrs=cosine
N-PEPS+ \mathcal{U}	$D_{0.5}$, <i>tot</i> , lr=3e-4, o=Adam, lrs=cosinewarm	D_{rand} , <i>all</i> , lr=1e-4, o=Adam, lrs=reduceonplateau

4.D.5. Details of Inference Hyperparameters

For inference we use CAB (Zhang, 1998) which consists of performing beam search iteratively, with pruning conditions of beam search (i.e., beam size, expansion size, etc.) weakened with each iteration, until a solution is found. Simialr to PCCoder (Zohar & Wolf, 2018), we start with beam size = 100, expansion size = 10 and maximum depth of beam search = number of steps = maximum program length. If the beam search fails, we double the beam size and increase the expansion size by 10, and perform beam search again with the modified parameters. The beam search terminates if we exceed the timeout. If no solution is found at the end of CAB, we mark that solution as FAILED.

We created a smaller validation split called *smallval* which consists of 5% of the samples chosen randomly from the larger validation data. The size of *smallval* is 525 samples and 946 samples for E1 and E2, respectively. We used this set to find optimal values of PEPS timeout and α for each model. Table 4.6 provides the selected hyperparameter values for all the models in both the settings. Timeout of 5s is divided between PE Searches module and the aggregation + GPS module. The time allocated to latter is denoted by GT in the table.

⁶see <https://pytorch.org/docs/stable/optim.html> for more details

For GPS, since no PE solutions are discovered, the whole timeout is allocated to the GPS inference block and no aggregation happens, i.e., $\alpha = 0.0$.

Table 4.6. Hyperparameter values for Inference. PT = PEPS timeout, GT = 5 - (5 * PT).

Model	Hyperparameters	
	E1	E2
GPS	GT=5.0s, PT=0.0s, $\alpha=0.0$	GT=5.0s, PT=0s, $\alpha=0.0$
Sum	GT=2.5s, PT=0.5s, $\alpha=0.8$	GT=0.5s, PT=0.9s, $\alpha=0.2$
Mean	GT=2.5s, PT=0.5s, $\alpha=0.8$	GT=0.5s, PT=0.9s, $\alpha=0.2$
Mean+ \mathcal{U}	GT=2.5s, PT=0.5s, $\alpha=0.9$	GT=0.5s, PT=0.9s, $\alpha=0.4$
N-PEPS-PP	GT=1.0s, PT=0.8s, $\alpha=0.8$	GT=1.5s, PT=0.7s, $\alpha=0.8$
N-PEPS-PP+ \mathcal{U}	GT=1.0s, PT=0.8s, $\alpha=0.7$	GT=2.5s, PT=0.5s, $\alpha=1.0$
N-PEPS-PG	GT=1.0s, PT=0.8s, $\alpha=0.8$	GT=2.0s, PT=0.6s, $\alpha=0.9$
N-PEPS-PG+ \mathcal{U}	GT=1.0s, PT=0.8s, $\alpha=0.8$	GT=1.0s, PT=0.8s, $\alpha=1.0$
N-PEPS	GT=0.5s, PT=0.9s, $\alpha=0.8$	GT=1.0s, PT=0.8s, $\alpha=0.8$
N-PEPS+ \mathcal{U}	GT=1.0s, PT=0.8s, $\alpha=0.8$	GT=2.0s, PT=0.6s, $\alpha=0.9$

4.D.6. Variation across different runs and machines

To ensure robustness and reproducibility of our results, we performed experiments with variations along three dimensions: different runs on the same machine, different machines and different test splits. Table 4.7 presents the results of variation in success ratio for E1 when run across different test splits, machines and runs across a single machine. Each run consists of a single CPU thread and single core setting on a machine (Google Cloud instance with 120GB RAM). We can see that there is very little variation for runs across the same machine. Hence, for our main experiments we chose to report standard error across different test splits with single runs on machines that are chosen randomly from a pool of 7 Google Cloud instances with same configuration.

Table 4.7. Variation in success ratio for runs across the same machine (run1, run2, run3), different machines (M1, M2, M3) and different test splits (split-1, split-2) for E1

	split-1									split-2								
	M1			M2			M3			M1			M2			M3		
	run-1	run-2	run-3	run-1	run-2	run-3	run-1	run-2	run-3	run-1	run-2	run-3	run-1	run-2	run-3	run-1	run-2	run-3
GPS	77.4	77.4	78	77	77.8	77.2	77.2	77.2	77.6	78.4	78.2	78.6	78	78	78	78.2	78	78
Sum	82.8	83.2	82.8	82.6	82.8	82.8	83.2	82.8	82.8	84.6	84.6	84.8	84.4	84.4	84.4	84.6	84.6	84.6
Mean	82.8	82.6	82.6	82.4	82.6	82.4	82.6	82.6	82.6	85	85	85	84.8	84.8	85	85	85	84.8
Mean+ \mathcal{U}	82.8	82.8	82.8	82.8	82.6	82.4	82.6	82.8	82.8	85	85	85	84.8	85	84.8	85	85	85
N-PEPS-PP	86.8	86.8	86.6	86.6	86.6	86.6	86.6	86.6	86.8	89.4	89.4	89.4	89.2	89.2	89.2	89.4	89.2	89.4
N-PEPS-PP+ \mathcal{U}	86.4	86.6	86.6	86.4	86.4	86.4	86.6	86.6	86.6	88.6	88.6	88.6	88.4	88.4	88.4	88.6	88.6	88.6
N-PEPS-PG	86.4	86.4	86.4	86.4	86.4	86.4	86.4	86.4	86.4	87.6	87.6	87.6	87.4	87.4	87.4	87.6	87.6	87.6
N-PEPS-PG+ \mathcal{U}	87.8	88	88	87.8	87.8	87.8	88	88	87.8	89	89	89	88.8	89	88.8	89	89	88.8

4.E. Results for Variants of Key for E2

Table 4.8 presents the test results of ablation studies with different variants of keys for E2. Similar to E1, we see that all the variants perform better than the corresponding values for GPS and the three ablation baselines (see Figure 4 in Section 4.3). We also see that the variant mentioned in Section 3.2 (denoted by N-PEPS in the table) performs the best. Note that even though, these results are on test data, we had chosen the best variant based on the results on the validation data.

Table 4.8. Success Ratio with standard error for key variants for E2

Variant	Length = 5	Length = 8	Length = 10	Length = 12	Length=14
N-PEPS-PG	78.49 \pm 0.35	45.92 \pm 0.53	31.36 \pm 0.33	22.83 \pm 0.33	17.15 \pm 0.31
N-PEPS-PG+ \mathcal{U}	78.16 \pm 0.30	46.37 \pm 0.57	31.88 \pm 0.35	23.17 \pm 0.33	17.62 \pm 0.30
N-PEPS-PP	78.74 \pm 0.32	45.9 \pm 0.57	31.16 \pm 0.33	22.67 \pm 0.32	16.91 \pm 0.28
N-PEPS-PP+ \mathcal{U}	78.87 \pm 0.35	44.87 \pm 0.50	30.69 \pm 0.41	22.43 \pm 0.36	16.59 \pm 0.32
N-PEPS	79.18 \pm 0.31	47.23 \pm 0.49	32.3 \pm 0.34	23.34 \pm 0.28	17.35 \pm 0.31
N-PEPS+ \mathcal{U}	79.19 \pm 0.30	46.31 \pm 0.61	31.84 \pm 0.36	22.71 \pm 0.28	16.68 \pm 0.21

4.F. Additional Results

4.F.1. Longer Timeout Results

We wanted to know whether the performance gains of N-PEPS gets translated to scenarios with a higher computational budget (as opposed to a lower budget of 5s in our setting). We performed inference with a timeout of 1000s using our previously trained models for GPS and N-PEPS in the E2 setting. For one test split consisting of 500 examples of length=12, the success ratios for GPS and N-PEPS were 54.38% and 57.14%, respectively. As expected, when given a higher budget, the numbers for both methods increase. However, N-PEPS still outperforms GPS. Note that here we chose the inference hyperparameters based on an educated guess, i.e., $\alpha = 0.8$, PEPS timeout = 160s and the time given to the CA module = 200s. The test performance of N-PEPS is likely to increase further if the values of these hyperparameters are selected from the validation set. This result provides promising evidence towards the wide applicability of our framework for longer timeout settings.

4.F.2. Intent Generalization Results

There is an interesting scenario of *intent generalization* where generalization to examples outside of those given as specification is required, in assumption that the additional examples sufficiently define the intent. To see how N-PEPS fares in this setting, we performed experiments where we generated 5 additional IO examples apart from the 5 already present as part of our test data and then evaluated whether the discovered global solutions satisfy

Table 4.9. Success Ratio with standard error for intent generalization experiments

Method	Length = 4 (E1)	Length = 5 (E2)	Length = 8 (E2)	Length = 10 (E2)	Length = 12 (E2)	Length = 14 (E2)
GPS	75.80 \pm 0.38	68.31 \pm 0.38	33.87 \pm 0.35	18.19 \pm 0.30	10.99 \pm 0.26	7.48 \pm 0.17
N-PEPS	84.09 \pm 0.27	76.16 \pm 0.32	36.33 \pm 0.43	21.02 \pm 0.29	13.17 \pm 0.25	9.17 \pm 0.23

the newly generated examples. In Table 4.9 we provide the success ratio with standard error for GPS and N-PEPS across 30 test splits. As can be seen from the results that even though the numbers have reduced from those provided in the tables provided in Figures 3 and 4 of our paper (as expected because the examples are outside of the specification), N-PEPS still outperforms GPS in both E1 and E2 across all lengths.

4.F.3. Function wise performance

We wanted to see which instructions in the DSL are "difficult" and compare the difficulty across GPS and N-PEPS. To do this, we record the count of instructions in the cases where the model was not able to find any solution divided by the total count of the instructions. Note that we look only at the operator and not the full statement, i.e., we ignore the arguments. Figure 4.7 shows this plot for GPS and N-PEPS+ \mathcal{U} for E1 with numbers across all 30 test splits. We see that usually higher-order functions like COUNT, ZIPWITH are "difficult" and functions like MAXIMUM, MINIMUM are comparatively "easy". Also, when compared with GPS, PEPS improves the failure rate across all instructions with improvements ranging from 32.67% for SUM to 52.72% for FILTER. Other notable improvements being 49.10% for MAXIMUM, 44.69% for MAP, 45.67% for SCAN1L and 47.14% for TAIL.

4.F.4. Perfect PE solutions

One of the advantages of PEPS is that we may get a single PE solution which satisfies all IO examples (we call this perfect PE solution). In these cases, we do not even need to go to the CA and depending on when this perfect PE solution is discovered, it can lead to significant time savings (e.g., if the first PE solution discovered turns out to be a perfect solution, then the time taken to find the solution is equal to just the PEPS timeout which is upper bounded by 1/5th of the total timeout). Figure 4.8 shows the fraction of perfect PE solutions with the length of test programs for N-PEPS for E2. We see that as the program length increases, we have less chances to find a perfect PE solution. This is expected because it will be difficult for a single PE solution to satisfy all IO examples as the programs become lengthy (and hence complex). Note that even though we increase the depth of beam search based on the length of the test program, the overall budget (=5s) and the PEPS timeout (=0.8s in this case) remains the same across different lengths. This also means that for higher lengths, N-PEPS needs to rely more on CA to find a global solution.

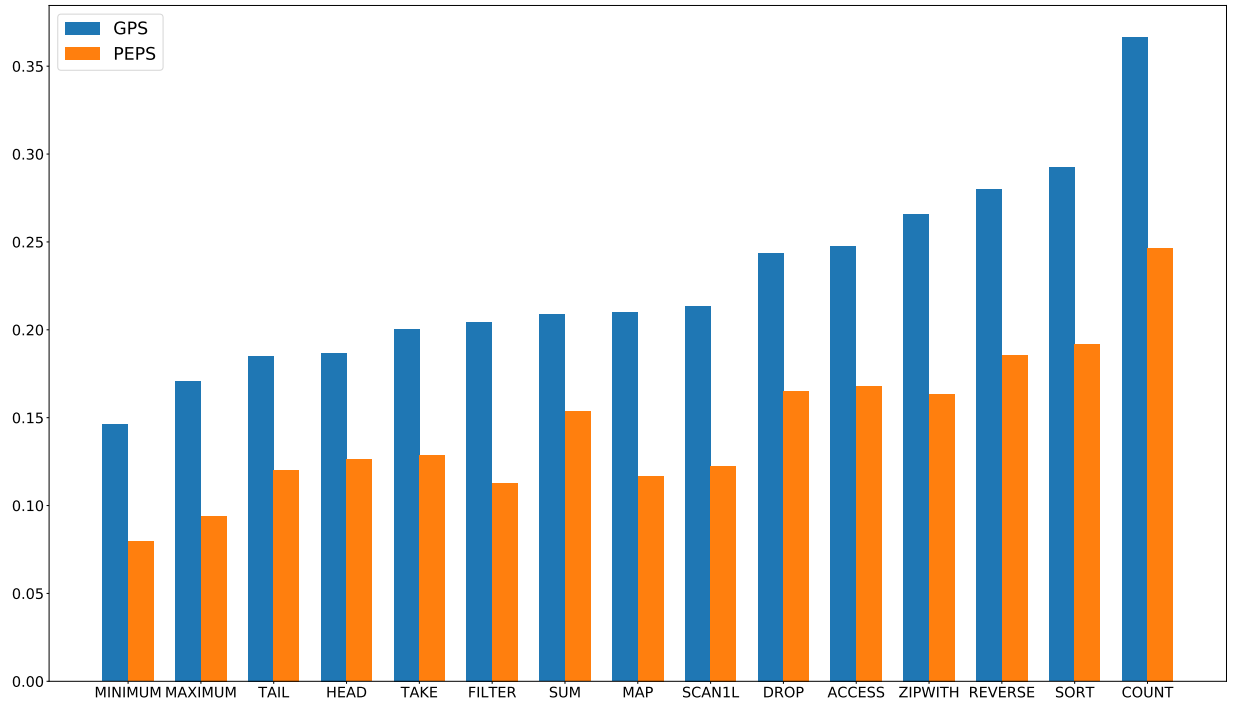


Fig. 4.7. Function-wise breakdown of failing cases for GPS and our N-PEPS model on E1

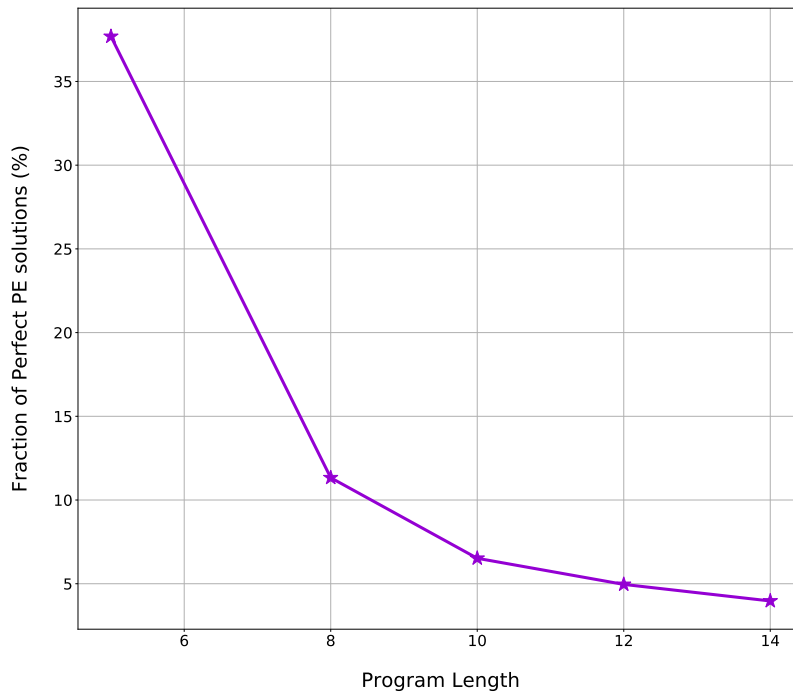


Fig. 4.8. Fraction of perfect PE solutions with length of test programs for our best model for E2

4.G. More insights into the workings of CA

We tried to gain more insights into how our Cross Aggregator mechanism works. First, we looked into some general patterns learnt by the CA module. Second, we were interested in finding out how often does CA rely on copying from the PE solutions, how often does it generate new operators (in isolation with the GPS module) and how does it generate new operators. The last question is important as it helps us understand the generalization capabilities of CA outside the statements in the PE solutions. Even though we found it difficult to figure out a fixed scheme that worked across all the settings and different examples, by doing nearest-neighbour analysis, we were able to find some useful patterns that might shed some light towards answering this question.

4.G.1. General Patterns Learnt by CA

To look for general patterns, we inspected the representations of the final linear layer of our trained CA model (that is responsible for providing the logits used in the global statement prediction). The size of this weight matrix is $n_s \times Z$, where the i -th row can be interpreted as a learned embedding corresponding to the statement index i . We ran t-SNE on these embeddings and looked for interesting clusters. We found many cases where functionally similar statements or statements with similar signatures were clustered together. We give few examples of these patterns below:

1. `REVERSE b` almost overlaps with `SORT b`. This is interesting because both take in the list `b` and return another list without performing transformations on the elements in `b`.
2. `MINIMUM b`, `MAXIMUM b`, `HEAD b`, and `TAIL b` are clustered together. This is interesting because all these operators select a single element from `b`.
3. `FILTER (ODD) a` is close to `FILTER (ODD) b`. In this case, there is a difference of only the argument. For cases, where the prior statements in the program lead to transformations such that the contents of lists `a` and `b` are the same, like `b = SORT a` or `b = REVERSE a`, swapping `FILTER (ODD) a` with `FILTER(ODD) b` and vice-versa will give the same result.

4.G.2. Overlap of PE Solutions with Global Solution

We wanted to see in how many cases do the operators present in the global solution also occur in one of the discovered PE solutions. This number gives us a rough estimate of how much can our attention mechanism do with just trying to copy these operators from one of the PE solutions when synthesizing the global solution. This is a rough estimate because we measure only the overlap of the operators and not statements, i.e., the arguments

to the operators in the PE solutions and the global solution can be completely different. Specifically, we record the number of operators that overlap between the global solution (taken as the ground-truth program) and one of the PE solutions, divided by the total number of statements in the ground truth programs across all cases.

The left part of Figure 4.9 shows the variation of this number with different lengths of test programs for N-PEPS for E2 when $\alpha = 0.8$ (which is the best-chosen hyperparameter value for this setting). We see that there is significant overlap between the operators indicating the quality of our PE solutions. However, the overlap decreases with length, which is also indicated by a decrease in overall success ratio with length (see left part of Figure 4). This is expected because we keep the same budget (PEPS timeout = 0.8s in this case) to discover PE solutions across all lengths. Improvements in the CA architecture focused to improve performance across longer length of programs in limited time budget, can be one of the potential directions to address this.

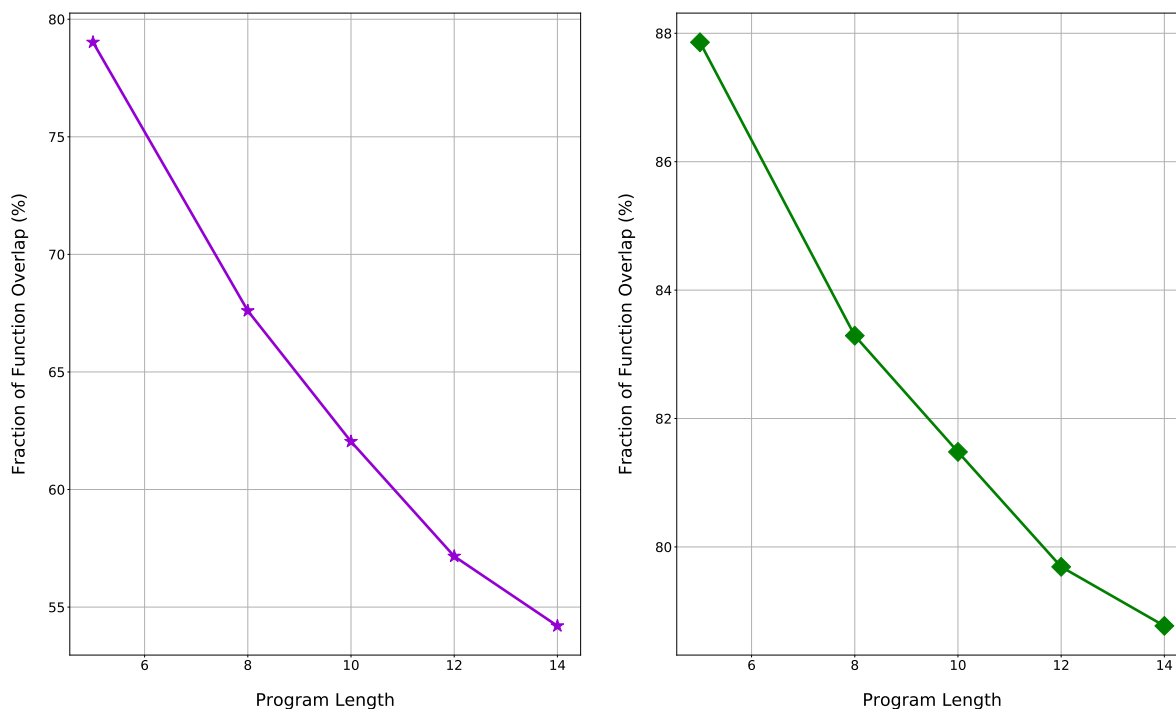


Fig. 4.9. Variation of fraction of operator overlap with length of test programs for our best model for E2: (Left) $\alpha < 1.0$ (CA + GPS); (Right) $\alpha = 1.0$ (CA alone)

There is significant overlap (about 79% for length 5), but not 100% between the operators, highlighting that in many cases (21% for length 5), N-PEPS performs discovery of new operators that are not present in the global solution. To further segregate the role of CA alone in the discovery of new operators as opposed to CA + GPS, we set $\alpha = 1.0$ and analyzed the operator overlap. The right part of Figure 4.9 shows this variation. As expected, the

overlap percentages increase as compared to the case when $\alpha < 1.0$. However, we can see that even when all the contribution to the global solution comes from the CA module alone, there is not a 100% overlap between the operators and therefore, there are non-zero chances of discovery of new operators. From these plots, we can conclude that the CA is not merely a copy mechanism and is useful in scenarios where the discovered PE solutions are not significantly overlapping with the global solution.

4.G.3. Sample cases where new operators are discovered

Apart from the analysis done above, we wanted to gain further intuition of how the new operators are being discovered by the CA module. To this effect, we looked at sample cases of the generation of new operators by just CA (i.e., $\alpha = 1.0$). In each box below (Figures 4.10-4.14), for test programs of length = 5, we report a case from the cases when the number of new operators discovered is 1, 2, 3, 4 and 5, respectively. The reported example shows the global solution discovered along with the corresponding PE solutions and is randomly chosen out of the total cases that fall within that category (i.e., not cherry-picked). For clarity, we bold the new operator in the global solution.

	PE Solutions:
	P₁ :
	a ← LIST
	b ← SCAN1L (+) a
	c ← MAP (+1) b
	d ← SORT c
	e ← SCAN1L (-) d
Global Solution:	P₂ :
a ← LIST	a ← LIST
b ← SORT a	b ← SCAN1L (+) a
c ← SCAN1L (+) b	c ← MAP (+1) b
d ← MAP (+1) c	d ← SCAN1L (-) c
e ← REVERSE d	P₄ :
f ← SCAN1L (-) e	a ← LIST
	b ← MAP (+1) a
	c ← SCAN1L (+) b
	d ← SORT c
	e ← MAP (-1) d
	P₃ = P₅ :
	FAILED

Fig. 4.10. New operators discovered = 1, Total cases = 2169

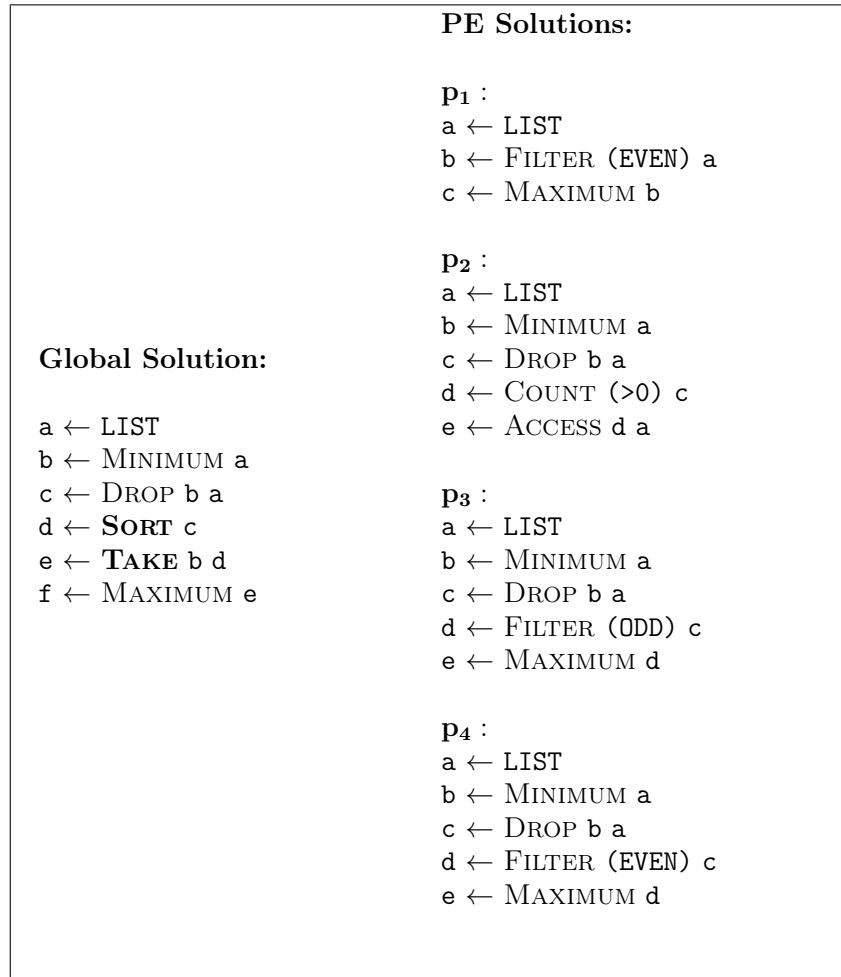


Fig. 4.11. New operators discovered = 2, Total cases = 1155

<p>Global Solution:</p> <p>a ← LIST b ← REVERSE a c ← COUNT (>0) b d ← ACCESS c b e ← TAKE d b f ← FILTER (ODD) e</p>	<p>PE Solutions:</p> <p>P₁ : a ← LIST b ← MINIMUM a c ← REVERSE a d ← FILTER (ODD) c</p> <p>P₂ = P₃ = P₄ : a ← LIST b ← FILTER (ODD) a c ← REVERSE b</p> <p>P₅ : a ← LIST b ← FILTER (<0) a</p>
--	--

Fig. 4.12. New operators discovered = 3, Total cases = 395

<p>Global Solution:</p> <p>a ← LIST b ← LIST c ← TAIL b d ← COUNT (>0) a e ← DROP d b f ← SORT e g ← ACCESS c f</p>	<p>PE Solutions:</p> <p>P₁ : a ← LIST b ← LIST c ← MAP (+1) b d ← TAIL c</p> <p>P₂ : a ← LIST b ← LIST c ← TAIL b</p>
---	---

Fig. 4.13. New operators discovered = 4, Total cases = 119

<p>Global Solution:</p> <p>a ← LIST b ← LIST c ← SORT b d ← MAP (+1) a e ← FILTER (>0) d f ← REVERSE c g ← ZIPWITH (+) f e</p>	<p>PE Solutions:</p> <p>P₁ = P₂ = P₃ = P₄ = P₅ : FAILED</p>
--	---

Fig. 4.14. New operators discovered = 5, Total cases = 15

Looking at the above samples, there appears to be a trend where discovering fewer and shorter PE solutions leads to more new operators discovered. This may be attributed to the fact that when there is less information in the PE solutions, there is usually more of a need to generate new operators. The example in Figure 4.14 is an extreme case of this, where no PE solutions were found, so all the operators need to be new.

4.G.4. Nearest-neighbour analysis for new operators

To gain intuition about how new operators are being generated by the CA module, we make two assumptions:

- If a statement occurs frequently among the PE solutions, there is a high likelihood that it will also be present in the global solution. We find some evidence of this from our experiments in the paper where we show that the Sum-PEPS baseline performs better than GPS.
- If two statements s_1 and s_2 are close to each other in the output embedding space (with embeddings e_1 and e_2), they will also be similar in their corresponding logits. Here, we are assuming that $e_1 \approx e_2 \rightarrow x * e_1 \approx x * e_2$, with x being the input activation.

With the above assumptions, for each of the examples provided in Appendix 4.G.3, we calculated the top-10 nearest neighbours of the PE statements (using the representations obtained in a way described Appendix 4.G.1). After this, we checked if the new operators in the global solution are present as part of the nearest neighbours of the PE statements. The presence of new operators points to a high likelihood of these being ranked higher in the beam search and hence being present in the global solution. In our analysis based on cases provided in Appendix 4.G.3, we did observe this trend. We provide some instances below:

- In Figure 4.10 above, the statements containing the new operator **REVERSE** occur as the topmost neighbour (based on distance) of **SORT c**, **MAP (+1) b**, as well as among top-3 neighbours of **SCAN1L (+) a**. Note that the variation in certain cases from the general pattern observed before might be attributed to the two assumptions mentioned above not completely holding true in all cases.
 - Top-3 neighbours of **SORT c** (occurs in p_1, p_4): [**REVERSE c**, **MAP (+1) c**, **COUNT (>0) c**]
 - Top-3 neighbours of **MAP (+1) b** (occurs in p_1, p_2): [**REVERSE b**, **SORT b**, **COUNT (>0) b**]
 - Top-3 neighbours of **SCAN1L (+1) a** (occurs in p_1, p_2): [**SCAN1L (-) a**, **SUM a**, **REVERSE a**]

- In Figure 4.11 above, new operator **Sort** is among the top-2 neighbours of **Count** (>0) **c**. Similarly, the new operator **Take** is among the top-2 neighbors of **Drop** **b a**.
 - Top-3 neighbours of **Count** (>0) **c** (occurs in p_2): [**Reverse** **c**, **Sort** **c**, **Maximum** **c**]
 - Top-5 neighbours of **Drop** **b a** (occurs in p_2, p_3, p_4): [**Access** **b a**, **Drop** **b c**, **Drop** **b d**, **Drop** **c a**, **Take** **b a**]

Chapter 5

Prologue to the second article

5.1. Article Details

On-the-Fly Adaptation of Source Code Models. Disha Shrivastava, Hugo Larochelle, Daniel Tarlow. This article ([Shrivastava et al., 2020](#)) was accepted for publication at the *Computer-Assisted Programming Workshop at Neural Information Processing Systems (NeurIPS) 2020*.

Personal Contribution The project began with discussions between Disha Shrivastava, Hugo Larochelle and Daniel Tarlow. Hugo Larochelle and Daniel Tarlow helped formulate the framework that is suited for adapting to local, unseen code context. Inspired by the concept of identifier locality within source code files, Daniel Tarlow proposed the notion of utilizing supporting information drawn from sections both preceding and following the target hole - location in the source code file where the prediction is intended. Our initial attempts were directed toward a meta-learning based source code model, but the empirical results were not that strong. Disha Shrivastava was involved in cleaning and preprocessing the data, writing all of the code, running the experiments, and writing significant parts of the paper. Hugo Larochelle and Daniel Tarlow advised on the project and were involved in the discussion of results, suggesting experiments to run, writing parts of the paper, and offering constructive critique during the drafting and revision phases of the paper.

5.2. Context

There are many factors that underscore the necessity for source code models to adapt to patterns during test time that were not encountered during training. These include new identifiers, coding styles, and naming conventions unique to specific organizations, projects, or developers. Therefore, the ability to adapt to unseen, local contexts is an important challenge that successful models of source code must overcome. One of the most popular

approaches for the adaptation of such models is dynamic evaluation. With dynamic evaluation, when running a model on an unseen file, the model is updated immediately after having observed each token in that file. In this work, we propose instead to approach this problem in two steps: (a) We select targeted information (*support tokens*) from the given context; (b) We use these support tokens to learn adapted parameters which are then used to predict the target hole. In terms of the broader theme of the thesis (see Section 1.1), the support tokens along with corresponding preceding tokens or support windows form the support context Z that are used to inform the prediction of the target hole Y . The adaptation framework that we propose for step (b) above constitutes the *Predict* module. The subsequent chapter (Chapter 6) discusses this work in detail.

5.3. Contributions

The paper proposed a novel framework called *Targeted Support Set Adaptation* (TSSA) that formulates the problem of adaptation to local, unseen context in source code by retrieving targeted information (support tokens) from both before and after the hole in a file. Our work demonstrated improved performance in experiments on a large scale Java GitHub corpus, compared to other adaptation baselines including dynamic evaluation, even with half the number of adaptation steps. Moreover, our analysis showed that, compared to a non-adaptive baseline, our approach improved performance on identifiers and literals by 44% and 19%, respectively.

5.4. Research Impact

This was one of the first works that considered the scenario of editing an existing file in an IDE where there can be code present both before and after the location where the edit is being made. Our evaluation setting that we called *line-level maintenance* imagines a cursor placed before a random token in a given file. We blank out the remainder of the line following the cursor to simulate a developer making an in-progress edit to the file and the task is to predict the blanked-out line. For our framework, we consider support tokens from code both before and after the line. This was different from the more popular language modelling setting at that time where code context present after the line was ignored. We utilized this finding in our later work (Chapter 8) where we use post-lines as an important context to include in the prompt. Recently, the importance of post context or suffix has become apparent with leading code models such as `code-davinci-002` from OpenAI, InCoder (Fried et al., 2022), StarCoder (Li et al., 2023) and CodeGen-2 (Nijkamp et al., 2023a) trained with fill-in-the-middle (Bavarian et al., 2022) objective that utilizes the context that comes after the target completion in the file.

Chapter 6

On-the-Fly Adaptation of Source Code Models

6.1. Introduction

The availability of large corpora of open-source software code like GitHub and the development of scalable machine learning techniques have created opportunities for the use of deep learning to develop models of source code (Allamanis et al., 2018a). Statistical language models for source code (Hindle et al., 2012), like natural language, are usually designed to take as input a window of tokens w and produce a predictive distribution for what the next token t might be. However, factors such as proliferation of vocabulary due to identifiers (such as names of classes, methods and variables) (Karampatsis & Sutton, 2019), the occurrence of repetitive patterns in local context (Tu et al., 2014) and faster rate of evolution of software corpora (Hellendoorn & Devanbu, 2017a), make modelling source code different from modelling natural language. According to Allamanis & Sutton (2013), in the Java GitHub corpus test set, for each project, on average 56.49 original identifiers (not seen in the training set) are introduced every thousand lines of code. There are also coding styles and conventions that are specific to each file and may not necessarily be seen in the training data. Each organization or project may impose its own unique conventions related to code ordering, library and data structure usage, and naming conventions. Additionally, developers can have personal preferences in coding style (e.g., preferring j as a loop variable to i). These motivate us to develop models that adapt their parameters to unseen contexts “on the fly”, i.e. they efficiently adapt to test files, even if the file contains identifiers and conventions that were unseen at training time.

A popular approach for model adaptation employed for natural language (Mikolov et al., 2010; Krause et al., 2018) and also advocated for source code (Karampatsis et al., 2020) is dynamic evaluation. With dynamic evaluation, we allow updating the parameters of a trained model on tokens in test files, from the first token to the last. To avoid bias and obtain an unrealistically optimistic measure of performance (i.e. cheating), the prediction of a token in a test file is made before updating the model’s parameters.

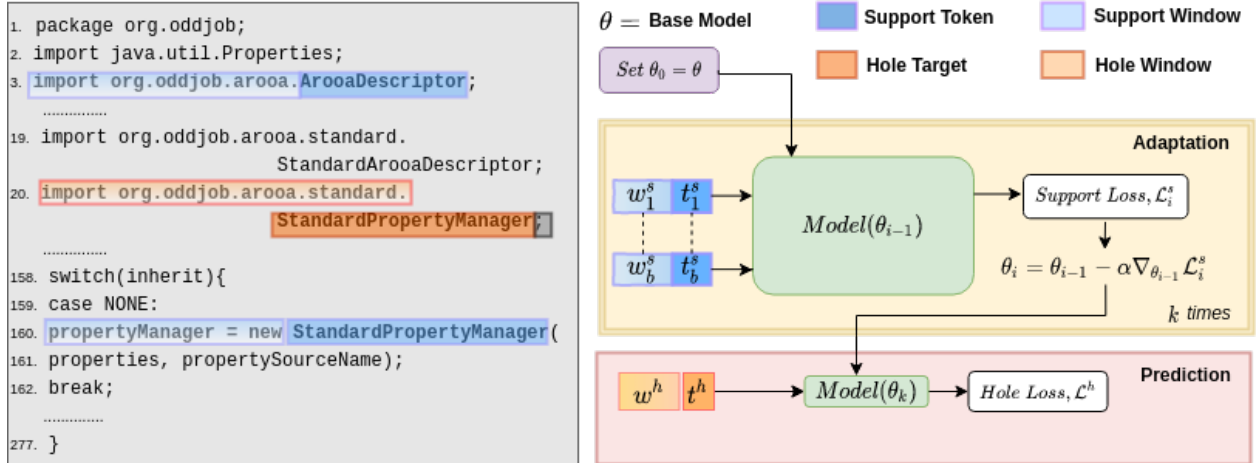


Fig. 6.1. Block diagram illustrating our approach for a sample file. To predict hole target `StandardPropertyManager` using *hole window* (w^h), our model learns parameter θ_k by performing k steps of gradient update using *support tokens* (t^s) and *support windows* (w^s) in its inner loop.

In this work, to reflect the way a software developer uses auto-completion in an IDE, we consider an evaluation setting that we call *line-level maintenance*. We imagine a cursor placed before a random token in a given file. We blank out the remainder of the line following the cursor to simulate a developer making an in-progress edit to the file. The task is then to predict the token (or *hole target*) that follows the cursor. This setting is different from the language modelling setting, where a test file is generated from scratch one token at a time, from top to bottom. Similarly, dynamic evaluation is ill-suited to this setting, as it processes tokens in that same order. Instead, we propose to select targeted information from both before and after the hole as a basis for adaptation.

In this work, we introduce Targeted Support Set Adaptation (TSSA), which leverages the notion of *support windows* and *support tokens* retrieved “on the fly” at test time. Figure 6.1 presents the specific task of predicting, on line 20, a hole target t^h from its hole window w^h or preceding tokens. To improve this prediction, in TSSA we leverage support tokens t^s (along with preceding tokens or support window w^s), which are tokens from around the file that we believe to be particularly influential in defining the nature of the local context. Intuitively, these could be tokens that are unique to the file and hence provide strong signal for adaptation. In Figure 6.1, lines 3 and 160 show the corresponding support windows (light blue shading) and support tokens (dark blue shading). The inner loop predicts support tokens t^s from support windows w^s and takes multiple gradient steps to update the parameters of the source code model and reduce the loss of its predictions. The updated parameters are then used to predict the hole target t^h from the hole window w^h . Our contributions can be listed as follows:

- We introduce TSSA, which formulates the problem of adaptation to the local, unseen context in source code by retrieving targeted information (support tokens) from both before and after the hole in a file. (Section 6.3.2).
- We consider a new setting that we call line-level maintenance for evaluating models for source code in a way that is directly inspired by the way developers operate in an IDE (Section 6.3.1).
- Via experiments on a large-scale Java GitHub corpus, we demonstrate that TSSA significantly outperforms baselines including dynamic evaluation, even with half the number of adaptation steps. Further, via ablations, we show that we improve performance on identifiers and literals by about 44% and 19% respectively (Section 8.3.3).

6.2. Related Work

There have been numerous efforts in developing models for source code, such as n -gram based (Hindle et al., 2012; Nguyen et al., 2013), CRF-based (Raychev et al., 2015; Bichsel et al., 2016), probabilistic graphical model based (Maddison & Tarlow, 2014; Raychev et al., 2016; Bielik et al., 2016); and Neural-networks based (White et al., 2015; Allamanis et al., 2018b; Dam et al., 2016). Some of these focus specifically on code-completion applications (Raychev et al., 2014; Alon et al., 2019; Svyatkovskoy et al., 2020; Li et al., 2018; Wang et al., 2020; Svyatkovskiy et al., 2020). To tackle the specific challenge of local context adaptation Tu et al. (2014) combined an n -gram with the concept of a cache. Later, Helleendoorn & Devanbu (2017a) extended this idea to develop nested n -gram models combined with a cache. The components in the cache could then come not only from the current file but also from other files in the directory or project, leading to significant improvements in performance. This idea could be adapted to our setting, by collecting support tokens beyond just the current file. Follow-up work from Karampatsis et al. (2020) has established the current state-of-the-art. They use deep recurrent models based on subword units. They apply dynamic evaluation by performing updates using information from all the files in a project and carrying over the updated value of parameters from one test file in the project to another during evaluation. However, on average this results in a long chain of adaptation steps before a prediction is made, which may present challenges when deploying in a real IDE (e.g., how to do quality control when the parameters used in the deployed system won't be known at release time?). In this work, we instead focus on and perform controlled experiments in a single file setting with a much smaller number of allowed update steps, which is more generally applicable.

6.3. Methodology

6.3.1. Line-level Maintenance

The line-level maintenance task is both more realistic (developers typically edit files rather than generating them from left-to-right) and creates the need for stronger forms of adaptation. More concretely, we refer to a file f as a sequence of tokens t_1, t_2, \dots, t_N . As per [Karampatsis & Sutton \(2019\)](#), we represent each token $t_n = (s_1, s_2, \dots, s_{l_n})$ as a list of l_n subtokens. Our task is to predict the first token (called *hole target*) in the blanked-out range, which occurs at a particular position in the file. For an example, refer to [Figure 6.1](#) where the hole target is highlighted in dark orange and the blanked out range is highlighted in black. Note that we are not allowed to use any token from the blanked-out range.

6.3.2. Adaptation

Base Model We begin by defining a *base model*, which is a Seq2Seq ([Sutskever et al., 2014](#)) model trained to predict the sequence of subtokens in the hole target t^h from the sequence of subtokens in the hole window w^h using parameters θ . The probability of hole target given its window can be written as

$$p(t^h|w^h; \theta) = \prod_{s_i \in t^h} p(s_i | s_{i-1}, \dots, s_1, w^h; \theta). \quad (6.1)$$

During the training of the base model, each token in the file is used as a hole target.

Targeted Support Set Adaptation (TSSA) To adapt the base model to the local file context, we consider regions from the file that potentially provide useful cues for predicting a given hole target. We call this set of tokens and preceding windows the *support set*, inspired by the usage of the term in few-shot learning ([Vinyals et al., 2016](#)). Each element of the support set, $S = \{(w^s, t^s)\}$ is a pair of support window w^s and support token t^s . The support windows and support tokens can come from anywhere in the file except for the blanked out remainder of the line following the hole target.

To adapt the model given a support set, we perform k steps of gradient descent over each of the k mini-batches of support windows and tokens. In each step, we predict the support token from the corresponding support window using the base model with parameters from the previous step. The *support loss* at step i and the updated parameters at step i can be written as

$$\mathcal{L}_i^s = \frac{1}{b} \sum_{j=1}^b \log p(t_{ij}^s | w_{ij}^s; \theta_{i-1}) \quad (6.2)$$

$$\theta_i = \theta_{i-1} - \alpha \nabla_{\theta_{i-1}} \mathcal{L}_i^s \quad [Inner \ Update], \quad (6.3)$$

where $i \in \{1, \dots, k\}$, $\theta_0 = \theta$, b = mini-batch size and α = hyperparameter corresponding to the inner adaptation learning rate. We then use the updated parameters θ_k to predict the hole target from its hole window, resulting in the *hole loss* \mathcal{L}^h

$$\mathcal{L}^h = \log p(t^h | w^h; \theta_k). \quad (6.4)$$

Support Set Selection Strategies A key novelty in this work is the idea of actively choosing a support set that leads to effective adaptation. This is in contrast to, e.g., few-shot learning, where the support set is defined by the task and cannot be changed. We can think of it being similar to self-supervised learning in the sense that the tasks are created from the given context.

In source code, identifiers are the most difficult to predict (Allamanis & Sutton, 2013) and also the most frequent of all token-types (Broy et al., 2005), making it the most common use-case for auto-complete systems. Thus, our definitions of support tokens are aimed at providing additional context that should help in predicting identifiers. We are motivated by the fact that identifiers are frequently re-used within a file even if they are uncommon across files (or even if they only appear in one file). Further, even when there is not an exact match, it is common for there to be repeated substructure in identifiers. Our work offers advantage compared to just using a powerful base model, like a transformer which has fixed context window size around the target hole and hence is ineffective to make use of these patterns which are far away from the cursor in the current file, especially if the file is long.

With this in mind, we explored four definitions of support tokens (which contribute towards determining the support sets): (a) **Vocab**: Tokens that are rare in the corpus; (b) **Proj**: Tokens that are relatively common in the current project but are rare in the rest of the corpus; (c) **Unique**: Single occurrence of a token in the support set; and (d) **Random**: Tokens are randomly selected. More details about each of these can be found in Appendix 6.A.

6.4. Experiments and Results

6.4.1. Experimental Details

For our experiments, we work with the Java GitHub Corpus provided by Allamanis & Sutton (2013). All our models are Seq2Seq networks where both encoder and decoder networks are recurrent networks with a single layer of 512 GRU (Cho et al., 2014) hidden units, preceded by a trainable embedding layer of equal size. To train the base model, we create minibatches of successive target holes as in standard training of language models, and we train to minimize average token loss. We use mini-batches of support tokens and the Adam (Kingma & Ba, 2015b) optimizer in the adaptation inner loop. An important note is that during evaluation, at the beginning of each inner loop execution, we not only set θ_0 to θ ,

but also set the state of the Adam optimizer to its value from the end of training. The latter step ensures that the statistics for Adam are not carried from one file to another. Details about the dataset and preprocessing; and best hyperparameter values for all settings can be obtained from Appendix 6.B and Appendix 6.C, respectively.

6.4.2. Evaluation Setup

There is a trade-off between accuracy and number of inner loop updates of adaptation. More inner loop updates generally improve cross-entropy but come at the cost of computation time and ultimately latency in a downstream auto-complete application. To control for this, we fix the size of batches and number of updates per hole target prediction across all adaptive methods. We measure the performance of our models in terms of token cross-entropy, MRR@10 and Recall@10 (see Appendix 6.E for details on these metrics). We experimented with the following methods:

- **Base model:** This is the pretrained base model used as is, without any contextual adaptation. This comparison allows us to confirm the benefit of adaptation in general.
- **TSSA- k :** This corresponds to doing k steps of inner loop adaptation using support tokens. We also report results for TSSA-1 (single inner-loop update), to highlight the value of multiple updates.
- **Dynamic Evaluation:** We also implement dynamic evaluation in our framework which is a bit different from in Karampatsis et al. (2020). Here, 1) the support sets are made of all window/tokens pairs (w^s, t^s) appearing *before* the hole target (and none after), and 2) we constrain the inner-loop optimization to order its updates by starting at the beginning of the file, until the token right before the hole target. Thus, the first inner-loop mini-batch of size b contains tokens at the beginning of the file, while the tokens immediately before the hole target only appear in the last mini-batch. Moreover, if the hole target is the m th token in the file, then there will be $\text{ceil}(m/b)$ updates in total. The variants of TSSA assume a fixed number k of inner-loop updates, unlike dynamic evaluation. To allow for an overall fair comparison, we set k to the average number of updates performed by dynamic evaluation, which was found to be approximately 16 for our test data.

6.4.3. Results

Performance on Hole Target Prediction: In Table 6.1, we report the average cross-entropy, MRR@10 and Recall@10 for test hole targets (all token types and identifiers). In these results, we sample five holes per file to measure test performance. For each method, we select the best values of hyperparameters using the performance on the validation data. As can be seen from the table, TSSA-16 gives the best performance in terms of cross-entropy,

Table 6.1. Performance on hole target prediction on test data in terms of token cross-entropy, MRR@10 and Recall@10. We also report 95% confidence intervals for each entry. We highlight the best performing models (in terms of mean) for each column.

Model	Cross Entropy	MRR@10 (All)(%)	MRR@10 (Identifiers)(%)	Recall@10 (All)(%)	Recall@10 (Identifiers) (%)
Base Model	5.222 ± 0.10	65.20 ± 0.42	24.90 ± 0.64	75.74 ± 0.42	36.20 ± 0.78
Dynamic Evaluation	3.540 ± 0.08	68.95 ± 0.41	34.44 ± 0.70	80.39 ± 0.39	48.86 ± 0.82
TSSA-1	3.461 ± 0.07	66.94 ± 0.40	35.76 ± 0.70	81.00 ± 0.38	52.04 ± 0.82
TSSA-8	3.383 ± 0.06	67.52 ± 0.40	35.14 ± 0.70	80.65 ± 0.38	50.27 ± 0.82
TSSA-16	3.240 ± 0.06	68.63 ± 0.40	36.74 ± 0.70	81.51 ± 0.38	52.34 ± 0.82

Table 6.2. Comparison of cross-entropy on prediction of identifiers and literals for TSSA-16 vs. a non-adaptation model.

Token Type	Base model	TSSA-16	% Improvement
Identifiers	13.16	7.35	44.15
Literals	7.18	5.82	18.94

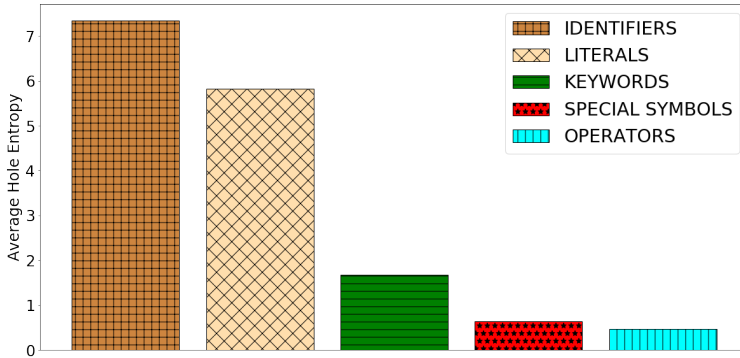


Fig. 6.2. Average hole target cross-entropy for each token-type for our TSSA-16 model.

MRR(Identifiers) and Recall; and is comparable to dynamic evaluation in terms of MRR (all). It is interesting to note that even TSSA-1 and TSSA-8 outperform dynamic evaluation in terms of cross-entropy, MRR(Identifiers) and Recall; even though they perform significantly less adaptation steps (single and half the number of adaptation steps, respectively as compared to dynamic evaluation (16)). This huge saving in terms of computational cost, is especially attractive while deploying models in an IDE where low latency is required.

Performance based on Token-Types: We analysed how our framework performs with hole targets of different token types (See Appendix 6.D for categorization of token-types). As can be seen from Figure 6.2, identifiers and literals (string literals, char literals, etc.) are the most difficult to predict amongst all token types. Table 6.2 shows the comparison of average test cross-entropy values for the non-adaptive base model as compared to our best model (TSSA-16). As can be seen from the table, we obtain significant reduction in cross-entropy

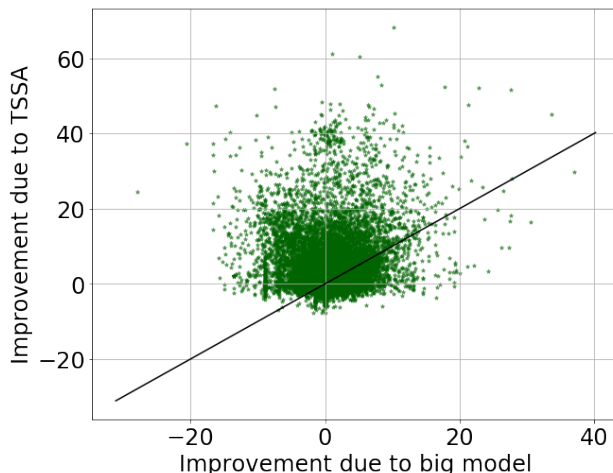


Fig. 6.3. Improvement due to TSSA on small capacity model ($b_{low} - m_{low}$) vs. Improvement due to big model ($b_{low} - b_{high}$)

values of about 44% and 19%, respectively in case of identifiers and literals. This in turn leads to better performance overall.

TSSA vs. Bigger Model One question is if benefits gained from TSSA are similar to or orthogonal to benefits that would arise from using larger and more sophisticated models. To study this question, we start from a “small base model” (256 hidden units) and build two models that improve, but in different directions. The first “big base model” increases the model size to 512 hidden units. The second “small TSSA” model leaves the hidden sized fixed but employs TSSA-16. We then compare how individual examples benefit from each kind of modelling improvement. Specifically, let the hole target cross-entropy for the small base model be b_{low} , for the big base model be b_{high} , and for the small TSSA model be m_{low} . In the right part of Figure 6.3 we plot the improvement obtained due to higher capacity model $b_{low} - b_{high}$ on the x-axis and improvement due to the low-capacity meta-learnt model $b_{low} - m_{low}$ on the y-axis. Each point represents a different test hole target. The line marks cases where improvement from both models is equal. First, we see that the majority (57.7%) of the points are above the line, indicating that applying TSSA improves on more cases than increasing the model size. Second, and perhaps more interestingly, there are many points where the improvement due to increasing model size is near zero, indicating that we have achieved saturation in benefit due to increasing model size in these cases. However, using TSSA here, even with the small model, often leads to a large improvement in performance. This shows that TSSA can help in adapting even when we reach saturation in terms of model capacity.

Performance based on Support Token Selection Strategies: We also experimented with the definition of support tokens where in one case we fixed the number of updates (16), while in the second we fixed the number of support tokens (256). Figure 6.4 displays the

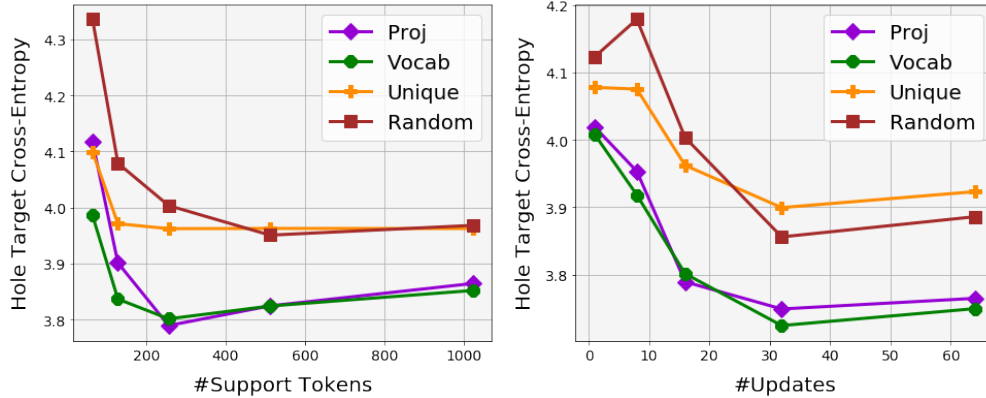


Fig. 6.4. Variation of token cross-entropy for val data with different definition of support tokens. (*Left*) With fixed number of updates; (*Right*) With fixed number of support tokens.

results for validation data. We see that the *Vocab* definition of support tokens performs best closely followed by *Proj*. On the other hand, *Unique* and *Random* perform worse in both cases. This highlights the fact that how we define support tokens indeed plays a role in performance improvement. We provide sample cases as well as some additional results including ablation studies with variation in the number of updates and number of support tokens in Appendix 6.G and 6.F, respectively.

6.5. Conclusions

In this work, we propose TSSA: an approach which selects targeted information from the local context and then uses this to learn adapted parameters, which can then be used for predicting a hole target in the current file. Our experiments on a large-scale Java GitHub corpus reveal the following: (a) Our formulation significantly outperforms all baselines including a comparable form of dynamic evaluation, even with significantly less adaptation steps in many cases; (b) Most of our performance benefits comes from reducing the cross-entropy on identifiers and literals. For future, we want to learn the criteria for building support sets.

Appendix for the second article

6.A. Support Set Definitions

In all cases, we ensure that the selection of support sets does not depend on the hole target or the blanked out region following the hole target.

- (1) **Vocab:** We try to capture tokens that are rare in the corpus as part of support tokens. We take all the tokens from the file and sort them based on their frequency in the vocabulary in reverse order and then take the top- N entries.
- (2) **Proj:** Here, as part of support tokens, our target is to capture tokens that are relatively common in the current project but are rare in the rest of the corpus. We divide each token’s frequency in the project with the frequency in the vocabulary, sort them and then take the top- N entries.
- (3) **Unique:** To study if multiple occurrences of the same token in the support set helps, we form a set of tokens in the file. We then take a subset of N tokens as part of our support set. Here, each support token in the support set is unique.
- (4) **Random:** We take N random tokens from the file as support tokens.

6.B. Dataset and Preprocessing

We work with the Java GitHub Corpus provided by [Allamanis & Sutton \(2013\)](#). It consists of open-source Java repositories for more than 14000 projects. Java is a convenient choice as it is one of the most popular languages for software development and has been widely used in previous works ([Karampatsis & Sutton, 2019](#); [Tu et al., 2014](#)). Following [Hellendoorn & Devanbu \(2017a\)](#), we focus on a 1% subset of the corpus. The name of the projects in training, validation and test splits of the dataset were taken from [Hellendoorn & Devanbu \(2017a\)](#)¹. Statistics of the data are provided in Table 6.3. Note that while we show results on Java, our method is otherwise applicable to corpora of any programming language.

We made use of the lexer provided by [Hellendoorn & Devanbu \(2017a\)](#)¹ to tokenize the files, preserving line-breaks. Note that the lexer also removes comments in the file. We need to use a Java-specific tokenizer because characters such as dot or semi-colon take a

¹<https://github.com/SLP-team/SLP-Core>

Table 6.3. Corpus Statistics for 1% split of the dataset. M indicates numbers in millions.

Feature	Train	Val	Test
# Projects	107	36	38
# Files	12934	7185	8268
# Lines	2.37M	0.50M	0.75M
# Tokens	15.66M	3.81M	5.31M
# Identifiers	4.68M	1.17M	1.79M

special meaning in Java and are not tokenized as individual tokens by NLP parsers. To get the Java token-types, we made use of Python’s Java-parser.² Subword tokenization was performed using the subword text encoder provided by Tensor2Tensor (Vaswani et al., 2018). As in Karampatsis & Sutton (2019), we use a separate vocabulary data split, consisting of a set of 1000 randomly drawn projects (apart from the projects in 1% split), to build the subword text encoder. In addition, we append an extra end-of-token symbol (EOT) at the end of each Java token. The final size of the subword vocabulary is 5710.

6.C. Details of Hyperparameter Values

In all settings of our Seq2Seq Models, the initial decoder state is set to be the last state of the encoder. The first input to the decoder is the last step output of the encoder. A dense layer with softmax output is used at the decoder. Also, note that both the parameters of the model and the state of Adam is reset after each hole target during evaluation. We use a dropout = 0.5 and gradient clipping = 0.25. We embedding layer dimension is equal to the hidden layer dimension = 512. We take both the support and hole window size to be 200. In Table 6.5 we define the best hyperparameter values for all our settings. Notation for reading Table 6.5 is provided in Table 6.4. For our experiments, we use NVIDIA P100 and K80 GPUs with 16GB memory each. To reduce model computation while decoding, we remove hole targets of length greater than or equal to 20 subwords. These constitute only 0.2% of the total number of tokens in training data and 0.1% in validation and test data, making it less significant.

²<https://pypi.org/project/javac-parser/>

Table 6.4. Notation for terms occurring in Table 6.5

Symbol	Meaning
lr	learning rate of Adam optimizer
hbs	hole batch-size
dbn	batch-size of tokens in dynamic evaluation
sbs	support tokens batch-size
#up	number of inner loop updates
snum	number of support tokens
sdef	definition of support tokens
ilr	learning rate of inner update Adam optimizer
T:	while training/ meta-training
E:	while evaluation

Table 6.5. Best hyperparameter values for all our settings.

Model	Hyperparameters
T: Base Model	lr = 1e-4, hbs = 512
E: Base Model	hbs = 1
E: Dynamic Evaluation	lr = 1e-3, , hbs = 1, dbn = 20
E: TSSA-1	lr = 5e-3, hbs = 1, sdef = proj, snum = 1024
E: TSSA-8	lr = 1e-3, hbs = 1, sbs = 20, sdef = vocab, #up = k = 8, snum = 256
E: TSSA-16	lr = 5e-4, hbs = 1, sbs = 20, sdef = vocab, #up = k = 16, snum = 256

6.D. Categorization of Token Types

Table 6.6. Description of Java token-types given by Python’s Java-parser into broad token categories for ease of visualization.

Token Category	Java Token-Type
Identifiers	identifier
Keywords	import, break, throws, extends, for, public, return, protected, boolean, package, new, class, void, static, int, this, volatile, synchronized, if, private, final, implements, super, catch, try, throw, else, instanceof, long, abstract, enum, case, byte, char, break, interface, finally
Operators	dot, gt, lt, eq, plus, eqeq, colon, bangeq, ques, ampamp, sub, bang, plusplus, barbar, star, amp, gteq, subsub, bar, ellipsis
Literals	stringliteral, intliteral, charliteral, longliteral, null, false, true
Special Symbols	semi, rparen, lparen, lbrace, rbrace, comma, monkeys_at, rbracket, lbracket

6.E. Evaluation Metrics

- **Cross-Entropy.** It is the average negative log probability of tokens, as assigned by the model. It rewards accurate predictions with high confidence and also corresponds to the average number of nats required in predicting a token. The cross-entropy of a sequence T with probability $p(T)$ under a model, is:

$$H_p(T) = -\frac{1}{m} \log p(T) \quad (6.5)$$

We evaluate the average under a distribution over hole target tokens where we first sample a file uniformly from the set of all files and then sample a hole target token uniformly from the set of all tokens in the file. This reflects the assumption that a developer opens a random file and then makes an edit at a random position in the file.

- **MRR/ Recall:** Since our approach can be used for code-completion (predicting the hole target), we need some metrics to measure the accuracy at this task. Mean Reciprocal Rank ($\text{MRR}@n$) is the average of the inverse of the position of the correct answer in a ranked list of size n . $\text{Recall}@n$ is 0 or 1 based on the absence or presence of the correct answer in the ranked list of size n .

6.F. Variation with #support tokens and #updates

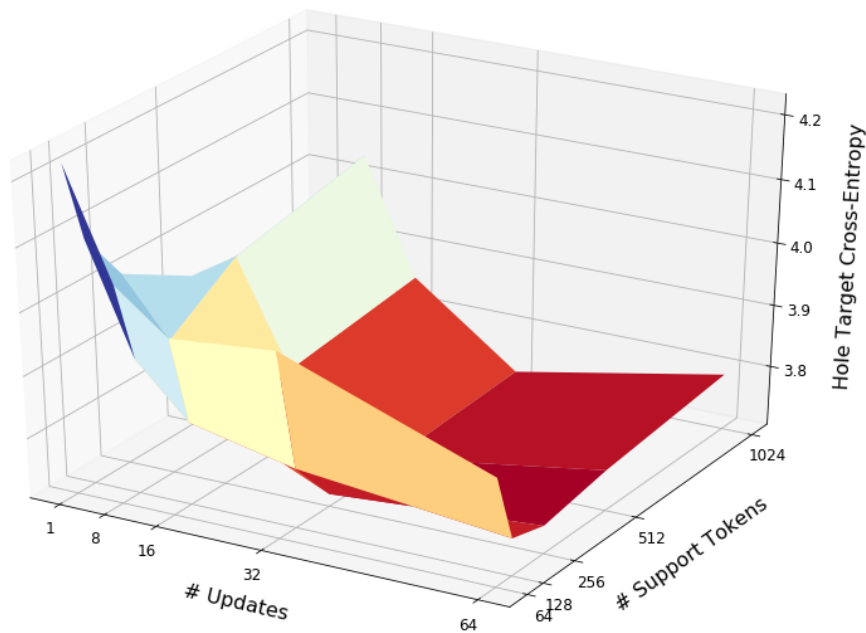


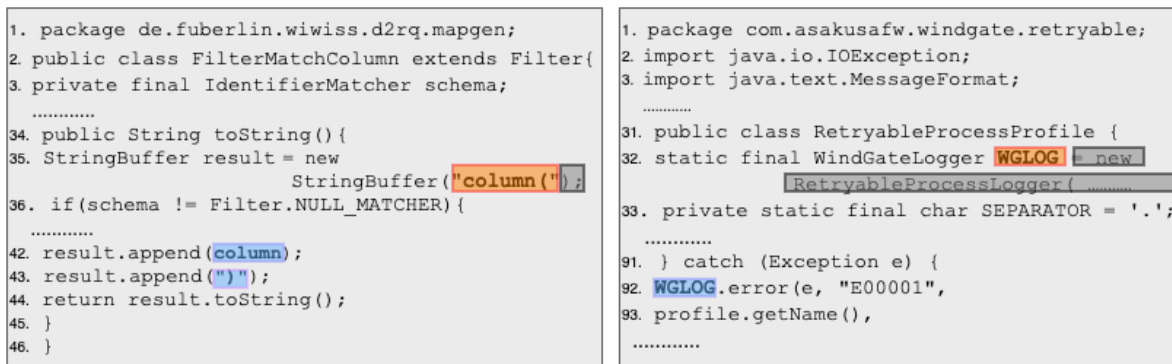
Fig. 6.5. Variation of hole target cross-entropy values with number of updates and number of support tokens for the validation data.

We took our best performing TSSA-16 for all the experiments that follow. In Figure 6.5, we plot the variation of hole target cross-entropy values with the number of updates and number of support tokens (N from Section ??), for validation data. As can be seen from the plot, the cross-entropy decreases with more updates. We also see that for a fixed number of updates, the cross-entropy decreases with the number of support tokens only until it reaches a certain point after which it increases. This likely arises from the way we form mini-batches

of support tokens where we first shuffle the support tokens and then cycle through them until exhausting the number of updates. This suggests that going past the point where each support token has been visited once creates redundancy that is detrimental.

6.G. Sample Cases

In Figure 6.6 we showcase two such sample cases. For the left one, we have a string literal as hole target (“column(”). We can see that fragments of it can be found in support tokens (highlighted in blue). The right one has an identifier (WGLOG) as hole target. Somewhere far later in the file, we find a support token that exactly matches the hole target, contributing to a large gain in performance of TSSA as compared to no adaptation. In neither of these cases does a larger or more sophisticated base model help in harnessing this extra information.



```
1. package de.fuberlin.wiwiss.d2rq.mapgen;
2. public class FilterMatchColumn extends Filter{
3. private final IdentifierMatcher schema;
   .....
34. public String toString(){
35. StringBuffer result = new
       StringBuffer("column(");
36. if(schema != Filter.NULL_MATCHER){
   .....
42. result.append(column);
43. result.append(")");
44. return result.toString();
45. }
46. }
```

```
1. package com.asakusafw.windgate.retryable;
2. import java.io.IOException;
3. import java.text.MessageFormat;
   .....
31. public class RetryableProcessProfile {
32. static final WindGateLogger WGLOG = new
       RetryableProcessLogger( .....);
33. private static final char SEPARATOR = '.';
   .....
91. } catch (Exception e) {
92. WGLOG.error(e, "E00001",
93. profile.getName(),
   .....

```

Fig. 6.6. Sample cases illustrating the benefits of TSSA on low capacity model: (*Left*) Hole target is string literal with partial match in support tokens; (*Right*) Hole target is identifier with exact match in support tokens.

Chapter 7

Prologue to the third article

7.1. Article Details

Repository-Level Prompt Generation for Large Language Models of Code. Disha Shrivastava, Hugo Larochelle, Daniel Tarlow. This article ([Shrivastava et al., 2022](#)) was accepted for publication at the *International Conference on Machine Learning (ICML) 2023*.

Personal Contribution The project began with discussions between Disha Shrivastava, Hugo Larochelle and Daniel Tarlow. Given the excitement around the recent launch of GitHub Copilot¹ and the growing recognition of prompting as a potential strategy for adapting LLMs to downstream tasks in NLP, Disha Shrivastava decided to systematically investigate the effect of placing varied code contexts in the prompts for Codex ([Chen et al., 2021](#)) - the model used in GitHub Copilot and the state-of-the-art model of source code at that time. We extended the concept of obtaining useful context from beyond the scope of the current file (Chapter 6), to also incorporate a more comprehensive level, encompassing the entirety of the repository. Disha Shrivastava was involved in coming up with the idea, formulating the framework for selecting relevant repository-level context, writing all of the code, running the experiments, and writing the paper. Hugo Larochelle and Daniel Tarlow advised on the project and were involved in the discussion of results, suggesting experiments to run and offering constructive critique during the drafting and revision phases of the paper.

7.2. Context

The goal of this work was to provide mechanisms to incorporate domain-specific knowledge in the prompt design process for an LLM of code. We proposed a framework called *Repo-Level Prompt Generator* (RLPG) that learns to condition on the context around the intended completion and generates a relevant prompt using prompt proposals. The prompt

¹<https://copilot.github.com/>

proposals take context from the entire repository, thereby incorporating both the structure of the repository and the context from other relevant files (e.g. imports, parent class files). In terms of the broader theme of the thesis (see Section 1.1), RLPG serves as the *Enhance* module that helps select the most relevant context given contextual cues in the form of different prompt proposals (support context Z). This context in turn when used to prompt the pretrained LLM (*Predict* module) helps in generating the desired code completion (target Y). The subsequent chapter (Chapter 8) discusses this work in detail.

7.3. Contributions

We proposed a framework that, without requiring access to the weights of the LLM, generates a relevant prompt that is conditioned on the example at hand as well as the overall structure and contents of the repository. On the task of single-line code-autocompletion, we show that an oracle constructed from our proposed prompt proposals gives up to 36% relative improvement over Codex. This improvement is pleasantly surprising as Codex has never seen prompts made from these prompt proposals during training. Further, we show that when we use our prompt proposal classifier to predict the best prompt proposal, we can achieve up to 17% relative improvement over Codex, as well as improve over other baselines. In addition, we **open-sourced** the code, data and trained models for the work to facilitate future research in this direction.

7.4. Research Impact

One of the major advantages of our work is that it doesn't require any access to the weights of the LLM making it applicable in cases where we only have black-box access to the LLM. This encompasses most of the currently high-performing LLMs, such as GPT-4 and Bard. As LLMs continue to be integrated into products, this trend is set to expand, thereby emphasizing the importance and potential impact of our work. Even though our work is relatively new, it has inspired other works (Zhang et al., 2023; Ding et al., 2022) that try to use context from the repository to prompt the LLM. Recently, there is also a work-in-progress project to incorporate repository-level context in GitHub-Copilot² in order to make more context-aware predictions.

²<https://githubnext.com/projects/copilot-view/>

Chapter 8

Repository-Level Prompt Generation for Large Language Models of Code

8.1. Introduction

Large Language Models (LLMs) have demonstrated remarkable performance in natural language processing tasks (Brown et al., 2020a; Chowdhery et al., 2022), text-to-image generation (Ramesh et al., 2022a; Rombach et al., 2022) and even as a generalized agent (Reed et al., 2022). As opposed to the *pretrain-finetune* paradigm, *prompting* these LLMs have been found to yield good performance even with few-examples (Liu et al., 2021a). Besides providing a mechanism to control and evaluate a LM, prompts have been shown to elicit emergent behaviour as well. Examples of this behavior include GPT-3 (Brown et al., 2020a) doing better in tasks it has never seen during training and improved reasoning capabilities with few-shot (Wei et al., 2022) and zero-shot (Kojima et al., 2022) prompts that encourage a chain of thoughts. These factors highlight the importance of designing an effective task-specific prompt. However, currently we have a limited understanding of how to do this (Reynolds & McDonell, 2021). LLMs have also been used for modeling source code with impressive results (Austin et al., 2021; Fried et al., 2022; Xu et al., 2022a). In particular, one of the best performing LLM, Codex (Chen et al., 2021), has been deployed as part of GitHub Copilot¹, a state-of-the-art in-IDE code assistant. Despite the growing popularity of LLMs of code, there is no work that systematically tackles different aspects of prompt generation in relation to source code. One such aspect is that when it comes to code, the relevant context to be put in the prompt can come from not just the current file, but also from outside, such as imports, parent classes, files within the same directory, and API documentation. Also, depending on the scenario, the relevant context can be scattered across multiple locations. Since the LLMs have a limited context length available for the prompt, it becomes increasingly crucial for our domain-specific understanding to guide the selection of

¹<https://copilot.github.com/>

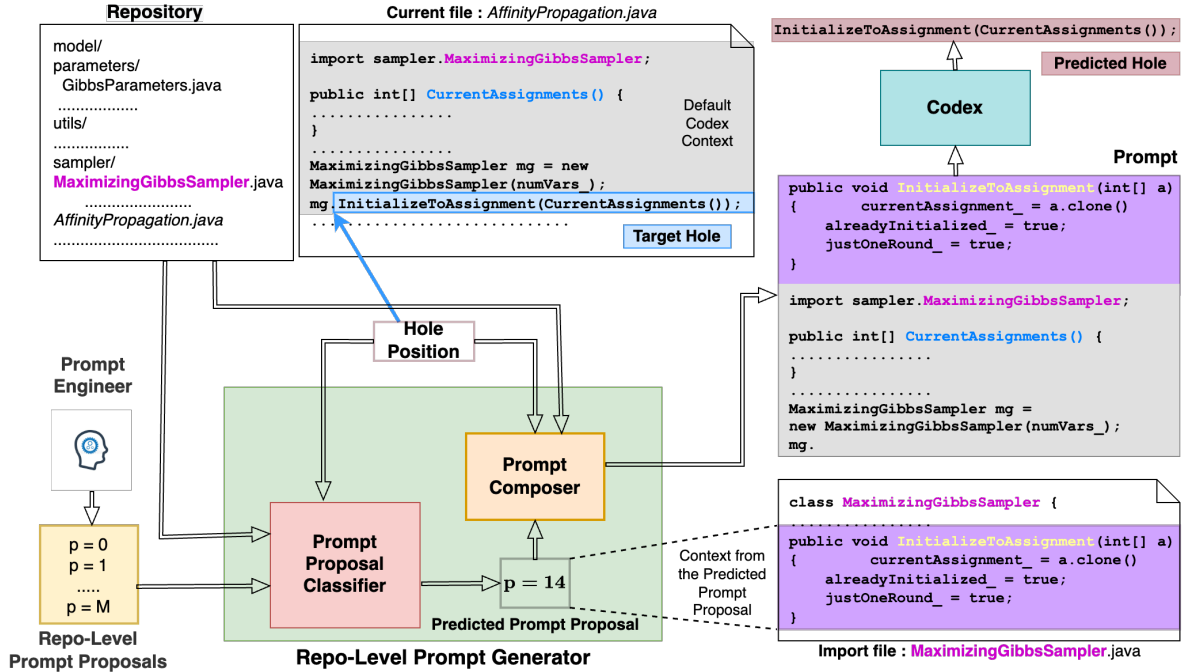


Fig. 8.1. Figure explaining the idea of **Repo-Level Prompt Generator**: Given a list of prompt proposals and the target hole position along with the associated repository as input, the prompt proposal classifier predicts a prompt proposal. The context from the predicted prompt proposal $p = 14$, i.e., method names and bodies from the imported file (highlighted in violet) is then combined with the default Codex context or context prior to the position of the hole in the current file (highlighted in gray) to compose a prompt. Prompting Codex with the generated prompt produces a prediction for the target hole (highlighted in dark red).

relevant context. Currently, it is not clear how to integrate this domain knowledge of what constitutes a relevant context, into the generation of prompts. Addressing this question has potential benefits in other domains such as question answering (Liu et al., 2022) and multi-document summarization (Xiao et al., 2022), where domain-specific structured retrieval of context can be useful.

In this work, we address this problem by proposing *Repo-Level Prompt Generator (RLPG)*, a framework that while generating the prompt, incorporates both the structure of the repository as well as the relevant context in the files in the repository. In RLPG, the choice of *where* from and *what to take* from the repository is specified by a set of *prompt proposals*. For example, one of the prompt proposals can be to take all the identifiers used in the first import file. These prompt proposals allow the prompt engineers to induce their domain expertise in the prompt-designing process. With the increasing use of LLMs as assistive agents to humans, the demand for transparency, and the desire of software engineers to tailor prompts to suit their requirements (Jiang et al., 2022; Sun et al., 2022), this capability becomes important. Similar to some previous works in NLP (Shin et al., 2020;

(Schick & Schütze, 2021), our prompt proposals are discrete. However, rather than fixing one particular prompt proposal for each example, we instead predict the best prompt proposal conditioned on the example. We do this by coming up with a neural network called *Prompt Proposal Classifier (PPC)* that *learns* to select a prompt proposal such that the resulting prompt is likely to produce the desired output. Therefore, RLPG allows the introduction of domain expertise, and at the same time facilitates automatic example-specific prompt generation via a learned neural network. Note that there are some techniques for automatic prompt generation in NLP (Li & Liang, 2021; Shin et al., 2020; Lester et al., 2021) that require updating some or all of the weights of the LLM. However, the strongest LLMs are not publicly available (e.g. OpenAI provides access *only* to the generated output from Codex via an API <https://openai.com/blog/openai-codex/> and no access to model weights and data is provided), making these techniques less useful under this scenario. RLPG addresses this limitation by generating prompts assuming only *black-box access* to the LLM. Even for cases where we have access to the model weights, RLPG provides a way to adapt to the repository-level context without having the need to finetune the model repeatedly. This can be particularly useful when adapting to a repository that contains proprietary or niche software, that the model has limited chances of seeing during training.

We focus on the task of single-line code autocompletion in an IDE, where the objective is to predict the blanked-out portion (or *target hole*) starting from the position of an imagined cursor to the end of the line (highlighted in blue in Figure 8.1). We operate under the line-level maintenance setting (Shrivastava et al., 2020; Hellendoorn & Devanbu, 2017b) that reflects the scenario where a user is editing an existing file. This means that there can be code following the line. Figure 8.1 provides an illustration of our approach. The prompt proposal classifier takes in the hole position (position of the cursor) in the current file, the repository to which the current file belongs, and a set of repo-level prompt proposals as input, and predicts a prompt proposal. In our illustrated example, the predicted prompt proposal corresponds to taking the method names and bodies from `MaximizingGibbsSampler.java` (`mg.before` the hole position indicates that a method from the imported file is likely to be invoked). The *Prompt Composer* uses the context from the predicted prompt proposal and combines it with the *default Codex context*, i.e., code prior to the position of the hole in the current file. The resulting prompt consists of the method name `InitializeToAssignment` (from the prompt proposal context) and the method `CurrentAssignments()` (from the default Codex context), resulting in a successful prediction (brown box on the top) of the target hole. Our key contributions are as follows:

- We propose a framework called the *Repo-Level Prompt Generator (RLPG)* that *learns* to generate prompts conditioned on the example, without requiring access to the weights of the LLM.

- RLPG allows us to use both the structure of the repository as well as the relevant context from all files in the repository, thereby providing a mechanism to incorporate domain knowledge in the prompt generation process.
- On the task of single-line code-autocompletion, we show that an oracle constructed from our proposed prompt proposals gives up to 36% relative improvement over Codex. This improvement is pleasantly surprising as Codex has never seen prompts made from these prompt proposals during training. Further, we show that when we use our prompt proposal classifier to predict the best prompt proposal, we can achieve up to 17% relative improvement over Codex, as well as improve over other baselines.

8.2. Repo-Level Prompt Generator (RLPG)

In this section, we provide details of our framework. We start by describing our prompt proposals and then discuss our prompt proposal classifier which is followed by a description of the prompt composer.

8.2.1. Repo-Level Prompt Proposals

The core idea of RLPG consists of substituting part of the default context used by Codex with context coming from somewhere else in the repository. The decision of what to take and from where in the repository to take from is governed by a set of prompt proposals. These prompt proposals were decided based on manual inspection of our training data and intend to capture common coding patterns (but more generally can also include project/organization-specific coding practices). A prompt proposal can be thought of as a function that takes as input a target hole’s position and the repository that the hole is a part of, and that returns the *prompt proposal context* (a string constituted by the context from the prompt proposal). A prompt proposal is specified by a prompt source and a prompt context type. We mention each of these along with their motivation below.

Prompt Source: For a target hole position, a prompt source determines from *where* should we take code that will be part of the prompt proposal context. We propose ten different prompt sources:

- (1) **Current:** take code from the current file excluding the contents of the target hole. The current file is the file that contains the target hole. The code in the current file (e.g. the lines after the hole position) can be very useful in predicting the target hole.
- (2) **Parent Class:** take code from the file that contains the parent of the class to which the target hole belongs. The intuition behind this is to account for cases where a method present in the parent class is invoked in the current file (i.e. the child class).

- (3) **Import:** take code from the import files used in the current file. The dependencies specified via imports can provide useful cues to predict the target hole.
- (4) **Sibling:** take code from the files that are in the same directory as the current file. Files in the same directory tend to share code variables (e.g. identifiers).
- (5) **Similar Name:** take code from files that have a similar name as the current file. Similar names are determined by splitting the file name based on underscore or camel-case formatting and then matching parts of the filename. If one or more parts matches, two files are considered to have similar names. The intuition behind this is that software developers tend to name files based on the functionality of the code written in that file. Therefore, a similar name file might contain some portion of the code that is common with the current file and hence might be useful for predicting the target hole.
- (6) **Child Class:** take code from files that have the current file as their parent class file.
- (7) **Import of Parent Class:** take code from the import files used in the parent class files.
- (8) **Import of Sibling:** take code from the import files used in the sibling files.
- (9) **Import of Similar Name:** take code from the import files used in the similar name files.
- (10) **Import of Child Class:** take code from the import files used in the child class files.

The last four prompt sources are useful when the target hole occurs at the very beginning of the current file. In these cases, there would be less context coming from other prompt sources. For each prompt source, we can get either a single file or a ranked list of files (see Appendix 8.B.1). In the latter case, we will take context from these files until we exhaust the maximum context length allocated to the prompt proposal.

Prompt Context Type: The prompt context type determines *what* code to take from the prompt source. We propose seven different prompt context types (Appendix 8.B.2 has examples of each type):

- (1) **Post Lines (PL):** Take all the lines after the target hole line till the end of the current file ².
- (2) **Identifiers (I):** Take all the identifiers used in the prompt source.
- (3) **Type Identifiers (TI):** Take all the type identifiers used in the prompt source.
- (4) **Field Declarations (FD):** Take all the field declarations used in the prompt source.
- (5) **String Literals (SL):** Take all the string literals used in the prompt source.
- (6) **Method Names (MN):** Take all the method names along with their signatures used in the prompt source.

²We also conducted experiments (Appendix 8.D.3) where we take lines starting from the 4th line after the hole.

- (7) **Method Names and Bodies (MNB)**: Take all the method names along with their signatures and corresponding bodies used in the prompt source.

By combining prompt sources with prompt context types, we get a total of 63 prompt proposals (see Appendix 8.B.4 for details). Note that depending on the target hole, not all prompt proposals would be applicable (e.g. if there are no parent classes in the current file, prompt proposals with prompt source as parent class file won't be applicable). In Figure 8.1, the predicted prompt proposal corresponds to taking prompt source **Import** and prompt context type **MNB**. We aimed for a set of prompt proposals that offer more diversity rather than a set of prompt proposals that are all good. This in turn ensures that for any hole position, a significant number of prompt proposals are applicable.

8.2.2. Prompt Proposal Classifier (PPC)

Given a hole position, the goal of the prompt proposal classifier is to predict the prompt proposal p that will lead to success, where success happens when the predicted hole \hat{h} exactly matches the target hole h . This task is formulated as a multi-label binary classification problem since for a given target hole, more than one prompt proposals can lead to success. In this formulation, we treat the default Codex context as one of the prompt proposals. Next, we describe the training procedure for PPC.

Training: For each target hole h , we generate a ground-truth vector $Y^h = [y_p^h]_{p=1}^M$ which is a multi-hot vector of size M , where M is the total number of prompt proposals. This vector is obtained by feeding the prompt generated from prompt proposal p into Codex and then seeing whether $\hat{h} = h$. If there is a match, we say that the prompt proposal p is successful. For hole h , if a prompt proposal p is applicable and leads to success, $y_p^h = 1$ and will be zero otherwise. For each hole h , we obtain a mask T^h where $T_p^h = 1$ when p is applicable or zero otherwise. The overall training loss \mathcal{L} can be expressed as the sum of individual hole losses \mathcal{L}^h :

$$\mathcal{L} = \frac{1}{N} \sum_{h=1}^N \mathcal{L}^h = \frac{1}{N} \sum_{h=1}^N \frac{1}{M^h} \sum_{p=1}^M BCE(\hat{y}_p^h, y_p^h) * T_p^h$$

In the above equation, $M^h = \sum_p T_p^h$ denotes the total number of applicable prompt proposals for h , N is the total number of holes encountered while training and BCE corresponds to the binary cross entropy loss. Masking ensures that we consider only the prompt proposals that are applicable. Next, we describe our two variants of PPC that can be used to obtain the prediction \hat{y}_p^h .

RLPG-H: Let H^h be the *hole window* that includes code present around the hole h excluding the hole itself. In our work, we take two lines before the hole position, the code up to the hole position and two lines after the hole position. We use a pretrained model F_ϕ to obtain a context representation vector of size Z , where Z is the dimension of the hidden state

of the model. Specifically, we take the hidden state at the first position, i.e. the representation of the [CLS] token. To make training of PPC computationally efficient, the parameters ϕ are frozen during training. The RLPG-H model takes the context representation of the hole window and projects it to the prompt proposal space of size M via two dense layers with a non-linearity in between (see Equation 8.1). Taking the sigmoid of this output gives the prediction of the prompt proposal.

$$\begin{aligned}\hat{y}_p^h &= P(y_p^h = 1 | H^h) \\ &= \text{sigmoid}(W^2(\text{relu}(W^1(F_\phi(H^h)) + b^1)) + b^2)\end{aligned}\tag{8.1}$$

RLPG-R: The motivation behind this variant is to use the similarity of the hole window and the prompt proposal context to determine which prompt proposal can be useful. Given a particular hole h , let C_p^h denote the prompt proposal context from prompt proposal p . Intuitively, if the hole window contains variables (e.g. identifiers) that are similar to the variables in the prompt proposal context, then there are chances that h might occur somewhere in C_p^h . The similarity is modeled using a multiheaded attention mechanism (Vaswani et al., 2017b), by treating the projected hole window representation as a query Q^h and the projected prompt proposal context representation K_p^h as a key. The value V_p^h is the same as the key.

$$Q^h = F_\phi(H^h), \quad K_p^h = F_\phi(C_p^h), \quad V_p^h = F_\phi(C_p^h)$$

The output from the multi-headed attention module, $MultiHead(Q^h, K_p^h, V_p^h)$ is fed to module G consisting of two layers of a feedforward network with relu activation in between (see Appendix 8.C for more details). The resulting output is then linearly projected and a sigmoid is applied to get the predicted prompt proposal.

$$\begin{aligned}\hat{y}_p^h &= P(y_p^h = 1 | H^h, C_p^h) \\ &= \text{sigmoid}\left(W_p G(MultiHead(Q^h, K_p^h, V_p^h)) + b_p\right)\end{aligned}$$

8.2.3. Prompt Composer

The prompt composer combines the context from the selected prompt proposal (given by PPC) with the context normally used by Codex (default Codex context) to generate the prompt. Since the total length that can be used for a prompt is fixed, we adopted a dynamic context allocation strategy where if the prompt proposal context is shorter than its allocated length, we assign the remaining portion from the prompt proposal context to the default Codex context. The prompt proposal context is always added before the default Codex context. For all prompt proposals, we assign half of the total context length to the prompt proposal context and the remaining to the default Codex context. For post lines, in addition, we also assign one-fourth and three-fourths of the total context length to the

Table 8.1. Statistics of our dataset.

Feature	Train	Val	Test	Total
# Repositories	19	14	14	47
# Files	2655	1060	1308	4757
# Holes	92721	48548	48288	189557

prompt proposal context. If the prompt proposal context or the default Codex context is greater than the context length allocated to it, we truncate it (see Appendix 8.B.3 for our truncation strategies).

8.3. Experiments and Results

In this section, we describe how we created the dataset, details of experiments along with different methods and their results, and interesting ablation studies.

8.3.1. Dataset Creation

To mitigate the effects caused by potential memorization of the code present in the dataset used for training Codex, we avoided code repositories from GitHub (Chen et al., 2021). Instead, we scraped Google Code <https://code.google.com/archive/> for repositories in Java (removing the ones that matched with a repository on GitHub with the same name). We selected the repositories that had a permissive license giving us a total of 47 repositories. We divided the repositories into train, validation, and test splits, where each repository in its entirety is part of a split. In each file within a repository, we remove lines that are either blank or part of comments and set the hole position to be the middle character in the line. All the characters from the middle position to the end of the line constitute the target hole.

Since code duplication has been shown to have adverse effects (Allamanis, 2018), within a repository, we look for files that are exact replicas of each other but placed in a different folder. We mark all such copies as duplicates and omit all of them when creating target holes for our dataset. Further, we found that the repositories were quite uneven in terms of their size. To avoid large repositories dominating the training of PPC, we capped the maximum contribution of holes from a repository to 10000, i.e. if the total number of holes in the repository exceeded 10000, we selected 10000 holes randomly from the total holes. Please see Table 8.1 for statistics of our dataset. The #Holes represent the holes after deduplication and capping. For some of our prompt proposals, we require semantic information that can be obtained with a parse tree. We used the tree-sitter API for Java³ that enables us to get the AST of a file and query it. Since our prompt proposals need information at a repository level, we stored some extra information that allowed us to collate the information from individual

³<https://github.com/tree-sitter/tree-sitter-java>

files according to the directory structure inside the repository (see Appendix 8.3.1 for more details).

8.3.2. Experimental Details

Prompt Generation: We used the OpenAI Codex Completions API for generating the predicted hole from the Codex model. In particular, we used the `code-davinci-001` engine with the temperature set to 0.0 and stop criteria as a newline. The completion length was 24 and the maximum prompt length was 4072. To allow for fast computation, we used simple models like CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2020) as our pretrained models. One of the limitations of these pretrained models is that the maximum context length that can be taken as input by these models is much smaller than the maximum context length allowed by Codex. Therefore, in PPC when we obtain the representation of the prompt proposal context, we need to truncate the context. This might lead to omitting important parts of the prompt proposal context in certain cases. Using pretrained models that allow larger context length or models that augment the context (Wu et al., 2022) offer avenues for future work. See Appendix 8.D.5 for results when we use a smaller context length with Codex.

Computational Complexity and Scalability of RLPG: To collect the ground-truth data for training our prompt proposal classifier, we queried the Codex API for each applicable prompt proposal per hole (with batching of 20 queries, we get a maximum rate limit of 400 holes per minute). This amounts to $\sim 150k$ queries to get the labels for the training data, $\sim 80k$ queries to get the labels for the validation data, making a total of $\sim 230k$ queries for training, i.e., 1.63 queries per target hole. The computational complexity of training our larger RLPG-R variant (3.6M parameters, 141269 holes, and 9.19 minutes per epoch on a single Tesla V100 GPU) is much smaller than finetuning all or some part of Codex (175B parameters). During inference, we need to calculate the repo-level statistics just once and all the subsequent hole completions in the repo can utilize this cached information, incurring no additional computational complexity. Besides training the PPC, all our experiments were performed on a CPU with 8GB RAM. Our prompt proposals are based on concepts such as post lines, imports, similar name files, method names, and identifiers that are quite general and applicable to other programming languages. In addition to the existing prompt proposals, our framework provides the flexibility to incorporate new prompt proposals. Since the cost of retraining RLPG with the extended prompt proposals is extremely low (much lower than finetuning Codex with the new prompt proposals), our framework can be used to make interventions on the LLM to address observed weaknesses as long as the intervention can be expressed as a prompt proposal that adds the missing context to the LLM. As opposed to techniques that perform prompt engineering in the latent space and require access to the

weights of the LLM such as Li & Liang (2021), RLPG facilitates expressing intent in the form of prompt proposals that are intuitive for humans, easy to understand, and do not require access to the weights of the LLM.

Methods: We experimented with the following methods for generating the prompt:

- (1) **Codex:** Using the default context from Codex as the entire prompt.
- (2) **Oracle:** Using the ground-truth vector Y^h (mentioned in Section 8.2.2). The prompt generated corresponds to using any of the successful prompt proposals (i.e., $y_p^h = 1$). Since this information is not available at inference, the oracle represents an upper bound.
- (3) **Fixed Prompt Proposal:** Using the most successful prompt proposal for all target holes. This was chosen based on the performance on the validation set and corresponded to taking 75% of the total context length from post lines in the current file.
- (4) **RLPG-H and RLPG-R:** Using the prompt proposal predicted by the RLPG-H and RLPG-R variants of PPC. The selected prompt proposal corresponds to taking the argmax of the predicted probabilities over different prompt proposals.
- (5) **RLPG-BM25:** Instead of using PPC to rank prompt proposals, use the scores obtained by BM25 (Jones et al., 2000) to select the best prompt proposal. The scores are calculated with the hole window being the query and prompt proposal contexts being the search documents. This serves as a non-learned retrieval method that makes use of our prompt proposals.
- (6) **File-level BM25:** Same as above, except that instead of using our prompt proposal contexts, search documents consist of full context from other files in the repository.
- (7) **Random:** For each target hole, select a context randomly from anywhere in the repository.
- (8) **Random NN:** Same as **Random**, except that amongst the randomly chosen contexts, we take the nearest neighbours of the hole window in the representation space of a pretrained model. This is analogous to the technique used in Liu et al. (2022).
- (9) **Identifier Usage:** For each target hole, we take the closest identifier and take usage windows of that identifier from everywhere in the repository. The usage window consists of two lines above and two lines below the usage line, including the usage line. We can rank the usage windows either randomly (**random**) or based on the nearest neighbour distance to the hole window in the representation space (**NN**).

The last four methods help us understand the performance when a context other than the prompt proposal context is used. To generate a prompt using these methods, we take 50% of the context from these followed by the default Codex context that takes up the remaining context length. For the NN baselines, we use CodeBERT (Feng et al., 2020) as the pretrained model. The contexts are taken in the increasing order of the nearest neighbour distances

Table 8.2. Performance of the oracle relative to Codex.

Data Split	Success Rate Codex(%)	Success Rate Oracle(%)	Rel. ↑ over Codex(%)
Train	59.78	80.29	34.31
Val	62.10	79.05	27.28
Test	58.73	79.63	35.58

until we exhaust the allocated context length. RLPG-BM25 helps us understand the role of PPC. See Appendix 8.C.3 for more details on the implementation of these methods.

Evaluation Metric: As mentioned in Section 8.2.2, to measure success, we use an exact match between the predicted hole string generated by Codex and the target hole string. In our experiments, we report the percentage of successful holes divided by the total number of holes for each split. We will call this *success rate* (SR) going forward.

8.3.3. Results

In this section, we present the results of the following two research questions explored in the paper:

- **[RQ1]-** Is it useful to generate a prompt that is composed of code context that is different from the default Codex context? If yes, what context can be useful?
- **[RQ2]-** For each target hole, is there a way of automatically selecting the prompt? If yes, how does this system perform relative to Codex?

RQ1 - Performance of Prompt Proposals: We found that combining the prompt proposal context (context from other files in the repository) with the default Codex context led to substantial improvement in performance. Table 8.2 shows the performance of an oracle constructed from our prompt proposals. We see that across all data splits, the prompt proposals contribute to significantly large improvements over Codex (upto 36% for test split). These results might seem surprising as Codex has not been trained on prompts that consist of context other than the default Codex context. What makes this result more surprising is that in most of the cases, the prompt consists of mashed-up context without logical ordering that may not even look like a semantically meaningful chunk of code (e.g. list of string literals from a sibling file followed by the default Codex context or post lines placed before the default Codex context as opposed to after). These results might suggest that as long as the relevant context (in our case repo-level knowledge in the form of prompt proposals) is present in any form in the prompt, it can be quite effective.

RQ2 - Performance of PPC: Having seen promise in our prompt proposals, next, we present the results of RLPG. Table 8.3 presents the success rates along with the percentage of relative improvements for the test data. The success rate is calculated by averaging across all holes in the test data (hole-wise). As can be seen from the table, all the RLPG

Table 8.3. Success Rate (SR) of different methods on the test data when averaged across all holes.

Method	Success Rate(%)	Rel. ↑(%)
Codex (Chen et al., 2021)	58.73	-
Oracle	79.63	35.58
Random	58.13	-1.02
Random NN	58.98	0.43
File-level BM25	63.14	7.51
Identifier Usage (Random)	64.93	10.55
Identifier Usage (NN)	64.91	10.52
Fixed Prompt Proposal	65.78	12.00
RLPG-BM25	66.41	13.07
RLPG-H	68.51	16.65
RLPG-R	67.80	15.44

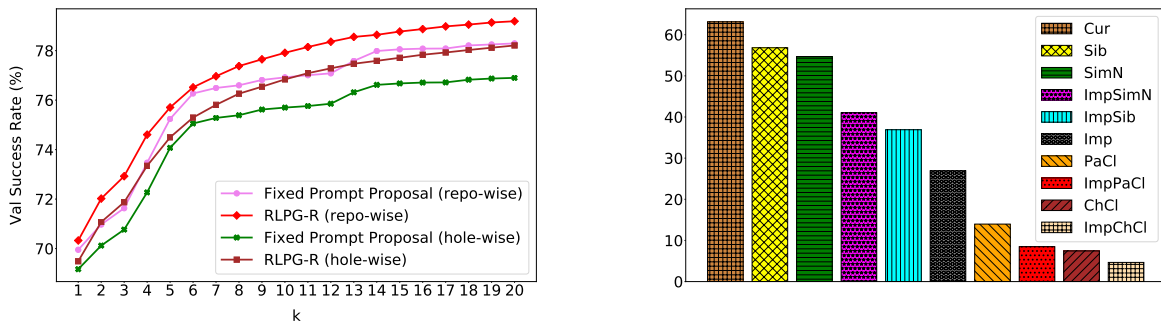


Fig. 8.2. (Left) Variation of RLPG and Fixed Prompt Proposal with #attempts (k); (Right) Mean success rates of different prompt sources when they are applicable.

variants as well as the fixed prompt proposal improve the performance significantly over Codex. The random baselines are either worse or on par with Codex. Identifier usage is a good baseline but still performs worse than either the fixed prompt proposal or RLPG. We see that File-level BM25 shows that even though better than Codex, it performs inferior to the methods that use some semantically meaningful notion of context (e.g. method bodies or field declarations). However, when we combine BM25 with prompt proposal contexts (RLPG-BM25), the performance improves a lot. All RLPG-based methods are better than fixed prompt proposal, showing the value of generating example-specific prompts using RLPG. However, both the learned variants of RLPG, i.e., RLPG-H and RLPG-R outperform the RLPG-BM25, highlighting the importance of learning PPC. See Appendix 8.D.1 and Appendix 8.D.7 for the performance of all methods across individual repositories. Note that even though we consider identifier usage as a separate baseline, one could consider it as one of the prompt proposals leading to further improved performance of RLPG.

Despite our efforts of avoiding overlap, since the training data for Codex is not exactly known, there might be a possibility that part of our Google Code data is part of the training data for Codex. Even if there were an overlap, we want to point out that since Codex has been trained with the default Codex context, during inference, it would be more beneficial for it to use the default Codex context in the prompt (rather than the context from the prompt proposals or any other context from other methods). This means that under this scenario, our evaluation would be more generous to the Codex baseline, leading to results more in favor of the Codex baseline than other methods we have used.

Variation with #attempts: Imagine a scenario where we have a human-in-the-loop who has been given k attempts to prompt the LLM and then choose one of the k predictions. We wanted to see how the performance of our framework varies with #attempts under this setting. This corresponds to using k prompts generated with top- k prompt proposals (one prompt per proposal) and marking success if any of the k prompts lead to success. The left side of Figure 8.2 shows the variation of SR over the validation data with the value of k . For RLPG, the top- k prompt proposals were chosen based on the decreasing order of probabilities given by PPC. For the fixed prompt proposal, the top- k prompt proposals were decided based on decreasing order of success rate of the individual prompt proposals on the validation dataset. From the figure, we notice that as we increase the value of k , the performance increases gradually at first and then saturates towards the oracle performance (79.05% for val data). This behaviour is observed for both fixed prompt proposal as well as RLPG. However, we see that for the same value of k , the success rate for RLPG is higher indicating that PPC learns a useful ranking of the prompt proposal contexts that can scale well with the #attempts.

Performance based on Prompt Proposals: The right side of Figure 8.2 shows the mean success rate of prompt sources, where success is counted only when the corresponding prompt source is applicable. From the figure, we see that the current file is the most important prompt source. Closely following are sibling files and similar name files. We see that all prompt sources have non-zero chances of success, highlighting the usefulness of each prompt source. See Appendix 8.D.2 for a similar breakdown based on prompt context type and Appendix 8.E for analysis of sample cases that lead to success and failure for RLPG.

Table 8.4. Edit distance based performance evaluation.

Method	Normalized Edit Distance(%)	Rel. ↑(%)
Codex	30.73	-
RLPG-H	22.55	26.62
RLPG-R	23.00	25.14

Edit Distance as a Metric: In addition to measuring string exact match, we also assess the performance of RLPG using the character-level edit distance ⁴ as a metric. Table 8.4 reports the average character-level edit distance normalized by the total number of characters in the target hole (lower is better). We see that both RLPG variants show significant relative improvements over Codex with RLPG-H getting as high as 26.62% relative improvement.

Experiments with code-cushman-001: To investigate whether the improvements achieved with RLPG are applicable to a different code model, we conducted experiments on the code-cushman-001 model from OpenAI ⁵. This model supports a context length of up to 2048 tokens, which is half the context length of code-davinci-001 and is expected to be relatively smaller (see Appendix A.2 of Rajkumar et al., 2022).

For evaluating RLPG, we choose the prompt proposal contexts based on the predictions from our trained RLPG models (trained on labels obtained from code-davinci-001). These contexts are then used as prompts for code-cushman-001 in order to get the completions. As shown in Table 8.5, on the test set, RLPG-H gets a relative improvement of 10.87% and RLPG-R gets a relative improvement of 10.95% over using the prior context in the file. These results suggest that RLPG has the potential to show improvements across different code completion models. We expect that having RLPG models trained on labels from code-cushman-001 would improve the results even further. However, in our opinion, the fact that we can use a single RLPG model (trained on code-davinci-001) to get improvements for two different code completion models (code-cushman-001 and code-davinci-001) is quite interesting.

Table 8.5. Success Rate (SR) with code-cushman-001.

Method	Success Rate(%)	Rel. ↑(%)
code-cushman-001	58.40	-
RLPG-H	64.74	10.87
RLPG-R	64.79	10.95

8.4. Related Work

LLMs for Code: Recently, there has been a lot of work around large language models of code. Decoder-only models correspond to generating code from left to right (Chen et al., 2021; Austin et al., 2021; Wang & Komatsuzaki, 2021; Black et al., 2021; Xu et al., 2022a; Fried et al., 2022). Encoder-only models use a masked language modeling objective (Feng

⁴<https://pypi.org/project/editdistance/>

⁵<https://platform.openai.com/docs/models/codex>

et al., 2020; Guo et al., 2020; Kanade et al., 2020). We also have encoder-decoder models that generally use a bidirectional encoding of a context to decode a series of masked tokens (Yue Wang, 2021; Li et al., 2022).

Repo-Level Info: Hellendoorn & Devanbu (2017b) propose a nested n-gram model that utilizes a locality-based cache where the locality consists of all directories from the root of the project (inclusive of the current file). Zhang et al. (2021) uses the parent class to generate the comments for the child class. Pashakhanloo et al. (2022b,a) convert the repository into a relational database and propose a graph-walk-based mechanism for pruning the unrelated context whereas Wang et al. (2021a) proposes a multi-relational graph neural network that uses inter-class and intra-class contexts to obtain code summaries. Lyu et al. (2021) incorporates the API-dependency graph in an LSTM-based Seq2Seq model to assist in code generation whereas, Zhou et al. (2023a) trains a model to augment code documentation to a natural language intent. Xu et al. (2022b) incorporates three types of structural locality features while training the kNN-LM (Khandelwal et al., 2020). These features are binary variables that correspond to the presence or absence of a similar hierarchy. The three levels of hierarchy are (a) sibling file, (b) file in the same repo (c) no hierarchy. In contrast, we have a much richer set of prompt proposals incorporating the semantics and structure of the repository. Also, we assume black-box access to the model and generate a prompt for the LLM without performing any finetuning of the LLM.

Prompt Generation: There have been promising works around prompt generation techniques in NLP. Broadly, there are two categories of automatic prompt-generation techniques. The first category corresponds to producing continuous/soft prompts where the prompt is described in the latent space of a language model (Li & Liang, 2021; Qin & Eisner, 2021; Bragg et al., 2021; Lester et al., 2021; Liu et al., 2021b). For example, Prefix-Tuning (Li & Liang, 2021) adds a prefix to the LM that can be learned by finetuning on examples from the downstream task. The second category produces discrete prompts where the prompt is a text string that can be interpreted by a human (Shin et al., 2020; Gao et al., 2021; Schick & Schütze, 2021). For example, Autoprompt (Shin et al., 2020) generates a prompt using a fixed template consisting of trigger tokens. The trigger tokens are shared across all inputs and determined by a gradient-guided search involving the LM. Our work falls in the category of discrete prompt generation techniques as we produce a prompt consisting of code tokens that can be easily interpreted by a human. However, in contrast to prior works that use a set of fixed templates for all examples, we learn to produce prompts conditioned on each example. Another important distinction is that we do not require access to the weights of the LM. A concurrent work as ours, (Wang et al., 2022) studies the role of prompt-tuning when compared to finetuning for code translation, defect localization, and code summarization. However, their technique requires access to the weights of the LLM and they perform experiments over models that are much smaller in scale than Codex. To

the best of our knowledge, our work is the first to explore automatic prompt generation in a black-box access setting in the domain of source code.

8.5. Discussion

We note that code-completion systems used in conjunction with LLM should be deployed with caution (Chen et al., 2021). Blind trust in these systems may lead to potential negative impact, as there might be cases where the generated code is insecure (Perry et al., 2022a) or contains sensitive information. After the submission of this paper, LLMs with larger input context lengths have been introduced, such as GPT-4⁶ which supports 32k tokens. With this expanded context length, one might consider including the entire content of the repository in the prompt. However, in practice, software repositories are often much longer. In our dataset (after deduplication), we observed that 70.22% of repositories contain more than 32k tokens. It is worth noting that apart from the current repository, there are other sources of relevant context, such as API documentation, tutorials, or related repositories, that can aid in code autocompletion. RLPG offers a mechanism to incorporate these additional sources of context through new prompt proposals. Therefore, regardless of the context length of the code-generating model, RLPG provides a valuable approach to determining which contexts are relevant to include in the prompt. With the increased context length in GPT-4, we anticipate less truncation of prompt proposal contexts, potentially leading to even greater improvements with RLPG.

In conclusion, we present RLPG, a framework that learns to automatically generate prompts conditioned on the example, without requiring access to the weights of the LLM. RLPG utilizes the structure of the repository as well as the context from other files in the repository using a set of easy-to-understand prompt proposals. In this work, we are taking context from only one prompt proposal. For future work, we want to learn a model that can automatically compose a prompt from multiple prompt proposals (see Appendix 8.D.4 for promising initial results). Other interesting directions include incorporating the user’s feedback in RLPG and extending RLPG to multi-line code auto-completion.

⁶<https://openai.com/product/gpt-4>

Appendix for the third article

8.A. Dataset Creation Details

8.A.1. Creation of Hole Completion Data

To collect the hole completion data, we scraped Google Code ⁷ for repositories tagged with the language “Java”. Then we deduplicated repositories by searching for a matching repository with the same name on GitHub. For those repositories with zero matching names on GitHub, we downloaded the archive and extracted the source code (preserving the directory structure). Next, we tried to determine the licenses of all repositories by either looking for a LICENSE file or matching with keywords "license", "copyright", "mit", etc. For repos for which our process was able to come up with a known license, we selected the ones having a permissive license, i.e., MIT, ApacheV2 and BSD. This was followed by removing files that are exact duplicates of each other within a repo. One of the reasons we found this inter-repository duplication may be because sometimes developers adopt lousy practices where instead of declaring a package and importing functions, they simply copy-paste the desired file into the current folder. The target holes coming from any of the duplicate files do not form part of the hole completion dataset. However, these files might be used to contribute to prompt proposal context for completing a target hole in a non-duplicate file. We felt comfortable with this choice since we wouldn’t want to predict a target hole in a duplicate file, but we can still use the context from the duplicate file to predict the hole in a file that is not its duplicate (e.g. in a sibling file). For the remaining files, we took each line that is not a blank line or a comment and chose the middle character as the hole position, i.e., all the characters from the middle of the line to the end of the line form the target hole. To avoid large repos having strong bias on our prompt proposal classifier, we capped the contribution from each repo to be a maximum of 10000 holes. If the number of holes in the repo exceeds 10000, we randomly select 10000 holes. Tokenization was done using the suggested tokenizer from OpenAI ⁸.

⁷<https://code.google.com/archive/>

⁸https://huggingface.co/docs/transformers/model_doc/gpt2#transformers.GPT2TokenizerFast

8.A.2. Creation of Data for Repo-Level Prompt Proposals

We used the tree-sitter API for Java ⁹ to get the parse-tree of an individual file in a repo. To get information at a repo level, for each file in the repo, we stored the following information:

- (1) list of all class names in the file. This helped us to get the parent or child class file corresponding to a given parent or child class.
- (2) the file corresponding to each import statement.
- (3) for each import statement in the file, the position in the file where the import is used. This is used for ranking the files based on the heuristics mentioned in Table 8.6.
- (4) list of sibling files
- (5) list of similar name files. This was done by splitting the filenames based on either camel-case or underscore. If the sub-parts of two files match, then they are said to have similar names.

The above meta-data was calculated only once for each repo. The subsequent hole completions can use the same cached information. In practice, we can use a hash to store and retrieve this info efficiently. For a prompt proposal, given the prompt source, we first obtain a single file or ranked list of files (see Table 8.6) using the info in the parse tree in conjunction with the above repo-level meta-data. All the prompt proposal context type information (MN, MNB, SL, I, TI, FD) can then be obtained by querying the parse tree of the selected file.

8.B. Prompt Proposal Details

8.B.1. Ranking of files based on prompt source

In Table 8.6, we provide details of how we select files for a given prompt source. Depending on the prompt proposal, we get either a single file or a list of files ranked based on some criteria. For example, if the prompt source is Import, we take all the import statements used in the current file and identify the location in the current file where the corresponding imports have been used. According to our heuristic, the closer the import usage to the hole position, the more likely it is for the prompt proposal context coming from the corresponding import file to be more relevant (to predict the target hole). We get a ranked list of import files sorted based on increasing order of distance (i.e., number of lines) between the import usage and the hole position. We start by taking all of the prompt proposal contexts from the first file in the ranked list and then keep iterating the ranked list until either the total context length allocated to the prompt proposal gets exhausted or we reach the end of the ranked list.

⁹<https://github.com/tree-sitter/tree-sitter-java>

Table 8.6. Selecting files for a prompt source

Prompt Source	File Ranking
Current	file with the target hole. Returns a single file.
Parent Class	file that contains the parent class that occurs closest to the target hole. Returns a single file.
Import	files with the corresponding import usage ranked based on the proximity to the hole. Returns a ranked list of files.
Sibling	files with import usage common to the current file and the sibling file ranked based on the proximity to the hole. The total number of common imports between the current and the sibling file is used as a tie-breaker. Returns a ranked list of files.
Similar Name	files with import usage common to the current file and the similar name file ranked based on the proximity to the hole. The total number of common imports between the current and the similar name file is used as a tie-breaker. Returns a ranked list of files.
Child Class	files with import usage common to the current file and the child file ranked based on the proximity to the hole. The total number of common imports between the current and the child class file is used as a tie-breaker. Returns a ranked list of files.
Import of Sibling	import files ranked based on the frequency of usage in all the sibling files. Returns a ranked list of files.
Import of Similar Name	import file ranked on the basis of frequency of usage in all the similar name files. Returns a ranked list of files.
Import of Parent Class	import file ranked on the basis of frequency of usage in all the parent class files. Returns a ranked list of files.
Import of Child Class	import file ranked on the basis of frequency of usage in all the child class files. Returns a ranked list of files.

8.B.2. Examples of Prompt Context Type

We provide examples of each of our prompt context types below:

- (1) Post Lines (PL): For the example shown in Figure 1 of the main paper, post lines will take all the lines after the line `mg.InitializeToAssignment(CurrentAssignments())` till we reach the end of the file (`AffinityPropagation.java`).
- (2) Identifiers (I): Identifiers are the names of variables used in the code. For example, for the prompt proposal context taken from the imported file shown in Figure 1 in the main paper (highlighted in violet), identifiers are `InitializeToAssignment`(line 1), `a` (line 1), `currentAssignment_` (line 2), `a`(line 2), `clone`(line 2), `alreadyInitialized_` (line 3), `justOneRound_`(line 4).
- (3) Type Identifiers (TI): Type Identifiers define the type of an identifier. For example, in the code snippet `class DPAffinityPropagation extends AffinityPropagation`

, `[AffinityPropagation` is labeled as a type identifier. Similarly in the snippet `DPAPPParameters parameters_;`, `DPAPPParameters` is a type identifier.

- (4) Field Declarations (FD): The variables of a class type are introduced by field declarations. For example, `double[] [] mHijMujT_;` and `MessageValuePair[] [] sortedMHijMujTs_;` are examples of field declarations.
- (5) String Literals (SL): A string literal is the sequence of characters enclosed in double-quotes. For example, in the code snippet, `System.err.println("DPAP load Warning: unknown parameter " + entries[0] + ", value = " + entries[1]);`, we have two string literals: (a) `"DPAP load Warning: unknown parameter "` ; (b) `", value = "` .
- (6) Method Names (MN): For the example shown in Figure 1 of the main paper, `public void InitializeToAssignment(int[] a)` is the method name prompt context type.
- (7) Method Names and Bodies (MNB): For the example shown in Figure 1 of the main paper, the part highlighted in violet represents the method names and bodies.

8.B.3. Truncation Strategies for Prompt Proposal Context

If the prompt proposal context is greater than the context length allocated to it, then we need to truncate the prompt proposal context. We followed the below two schemes for truncating context:

- **front:** We truncate the context from the front. This is used for all prompt sources except Parent Class and when we take PL from Current.
- **back:** We truncate the context from the back. This is used when the prompt source is Parent Class and when we take prompt context types other than PL from Current.

The truncation strategies for each case were selected based on the results of a small validation set. For the prompt source Current, except when the prompt context type is PL, we always start by taking the code of the prompt context type from after the hole position. This makes sense as the default Codex context will anyways contain code before the hole. Only if this turns out to be blank, we will use the code of context type from before the hole.

8.B.4. List of Prompt Proposals

Table 8.7. List of our proposed repo-level prompt proposals

Prompt Proposal ID	Prompt Source	Prompt Context Type
0, 1, 2, 3, 4	Current	MN, I, TI, SL, FD
5, 6, 7	Current	PL (taking 25%, 50% and 75% contribution to the total context length)
8, 9, 10, 11, 12, 13	Parent Class	MNB, MN, I, TI, SL, FD
14, 15, 16, 17, 18, 19	Import	MNB, MN, I, TI, SL, FD
20, 21, 22, 23, 24, 25	Sibling	MNB, MN, I, TI, SL, FD
26, 27, 28, 29, 30, 31	Similar Name	MNB, MN, I, TI, SL, FD
32, 33, 34, 35, 36, 37	Child Class	MNB, MN, I, TI, SL, FD
38, 39, 40, 41, 42, 43	Import of Sibling	MNB, MN, I, TI, SL, FD
44, 45, 46, 47, 48, 49	Import of Similar Name	MNB, MN, I, TI, SL, FD
50, 51, 52, 53, 54, 55	Import of Parent Class	MNB, MN, I, TI, SL, FD
56, 57, 58, 59, 60, 61	Import of Child Class	MNB, MN, I, TI, SL, FD
62	Codex	-

8.B.5. Other Prompt Proposal Variations

We experimented with other variations that include: (a) appending class names at the beginning of the prompt proposal context, (b) using newline or space to join the prompt proposal context and the default Codex context, (c) taking all or the top- k of the prompt context types, (d) ordering of top- k .

- **Context Separator:** This defines how we join the prompt proposal context string to the default Codex context string. We experimented with space and newline as context separators.
- **Prompt Proposal Context Formatting:** We can format the prompt proposal context before giving it to the Prompt Composer. We experimented with the following options:
 - (1) `class_name`: append [class name of the file] at the beginning of the prompt proposal context taken from each file that is part of the prompt source. For example, if we are taking prompt proposal context from two import files $f1$ and $f2$, the prompt proposal context will be formatted as [class name of $f1$] prompt proposal context from $f1$ + space + [class name of $f2$] prompt proposal context from $f2$. We use this when the prompt proposal context types are MN, I, TI, FD, and SL.
 - (2) `class_method_name`: we apply this only when the prompt proposal context type is MNB. We append method names at the beginning of each of the corresponding method bodies. We also append the prompt proposal context from a file with the name of the class as described in the previous item.

- (3) comment: Adding in the prompt proposal context as a comment, i.e., formatting it as/** prompt proposal context */. This wasn't found to be much useful.
- (4) none: passing the prompt proposal context as it is. We use this when the prompt proposal context type is PL.
- **Top-k Type:** For each of the prompt proposal context types, except PL, we experimented with taking the (a) first (b) last, and (c) all of the prompt proposal context types, i.e., we can take first-10 identifiers. We found 'all' to be the best among all.
- **Top-k:** We experiment with k values of (a) 10 (b) 20 and (c) all. We found 'all' to work best for all prompt context types.

8.C. Implementation Details

8.C.1. RLPG-H

We used Adam (Kingma & Ba, 2015a) optimizer with a learning rate of 3e-4 and batch size of 64. We used CodeBERT (Feng et al., 2020) as our pre-trained model F_ϕ to obtain the representation of the hole window. The size of the representation (corresponding to the hidden dimension of the [CLS] token) is 768. $W^1 \in \mathbb{R}^{512 \times 768}$, $b^1 = 512$, $W^2 \in \mathbb{R}^{63 \times 512}$, $b^2 = 63$.

8.C.2. RLPG-R

We used Adam (Kingma & Ba, 2015a) optimizer with a learning rate of 3e-4 and batch size of 64. We used CodeBERT (Feng et al., 2020) as our pre-trained model F_ϕ to obtain the representation of the hole window and prompt proposal context. The size of the representation (corresponding to the hidden dimension of the [CLS] token) is 768. The multiheaded attention (Vaswani et al., 2017b) is modeled as follows:

$$Q^h = F_\phi(H^h), \quad K_p^h = F_\phi(C_p^h), \quad V_p^h = F_\phi(C_p^h)$$

$$Att(Q^h, K_p^h, V_p^h) = V_p^h \text{softmax}\left(\frac{Q^h \top K_p^h}{\sqrt{d_k}}\right)$$

$$MultiHead(Q^h, K_p^h, V_p^h) = W^O \text{concat}(head_1, \dots, head_\tau)$$

where $head_i = Att(W_i^Q Q^h, W_i^K K_p^h, W_i^V V_p^h)$

In the above equations, d_k is the dimension of the key, W_i^Q, W_i^K, W_i^V are the query, key and value projection matrices, τ is the number of heads and W^O is the linear projection that combines the heads. The projection matrices $W_i^Q \in \mathbb{R}^{d_q \times d_{model}}$, $W_i^K \in \mathbb{R}^{d_k \times d_{model}}$, $W_i^V \in \mathbb{R}^{d_v \times d_{model}}$, $W^O \in \mathbb{R}^{d_{model} \times \tau d_v}$. For the multihead attention, we used $d_k = d_q = d_v = 32$, $\tau = 4$ and $d_{model} = 768$, $W_r \in \mathbb{R}^{63 \times 768}$ and $b_p = 63$. For each head, we perform scaled dot-product attention. G module consists of a dropout (Srivastava et al., 2014) layer, a

residual connection (He et al., 2016), a layernorm (Ba et al., 2016), followed by a sequence of (a) dense layer of weights= 2048×768 , bias= 768 , (b) relu activation, (c) dense layer of weights= 768×2048 , bias= 2048 , (d) dropout layer, (e) residual connection, (f) layer norm. A dropout value of 0.25 was used while training. Our model resembles one layer of the transformer encoder block (Vaswani et al., 2017b).

8.C.3. Baselines

Random baseline first selects a file randomly from the current repository followed by selecting a random line within that file. We choose all the lines starting from that line to the end line of the chosen file as context (excluding the hole window if the chosen file is the current file). The nearest neighbor similarity is based on the dot product between the representation of the hole window and the representation of the context, where we use a pre-trained CodeBERT (Feng et al., 2020) model to obtain the representations. For the Identifier Usage baseline, if the nearest identifier to the hole doesn't return any usage window, we proceed to the next nearest identifier. For faster computation and to avoid memory issues when running on our hardware, for NN baselines, we collect 64 random neighbors and then rank them based on the nearest neighbor distance. The BM25-based baselines use the Okapi BM25 implementation with default parameters given by the pip package `rank-bm25` 0.2.2¹⁰. For file-level BM25, if the file context exceeds the allocated context length, we truncate from the back.

8.D. Additional Results

8.D.1. Hole-wise and Repo-wise results

Table 8.8 shows the performance of all methods when averaged across all holes (hole-wise) and across individual repositories (repo-wise). Note that the latter metric is independent of the size of the repository.

¹⁰<https://pypi.org/project/rank-bm25/>

Table 8.8. Hole-wise and Repo-wise Success Rate (SR) of different methods on the test data.

Method	Success Rate(%) (hole-wise)	Rel. ↑(%) (hole-wise)	Success Rate(%) (repo-wise)	Rel. ↑(%) (repo-wise)
Codex (Chen et al., 2021)	58.73	-	60.64	-
Oracle	79.63	35.58	80.24	32.31
Random	58.13	-1.02	58.95	-2.79
Random NN	58.98	0.43	60.04	-0.99
File-level BM25	63.14	7.51	64.28	6.00
Identifier Usage (Random)	64.93	10.55	67.83	11.85
Identifier Usage (NN)	64.91	10.52	67.94	12.03
Fixed Prompt Proposal	65.78	12.00	68.01	12.15
RLPG-BM25	66.41	13.07	68.15	12.39
RLPG-H	68.51	16.65	69.26	14.21
RLPG-R	67.80	15.44	69.28	14.26

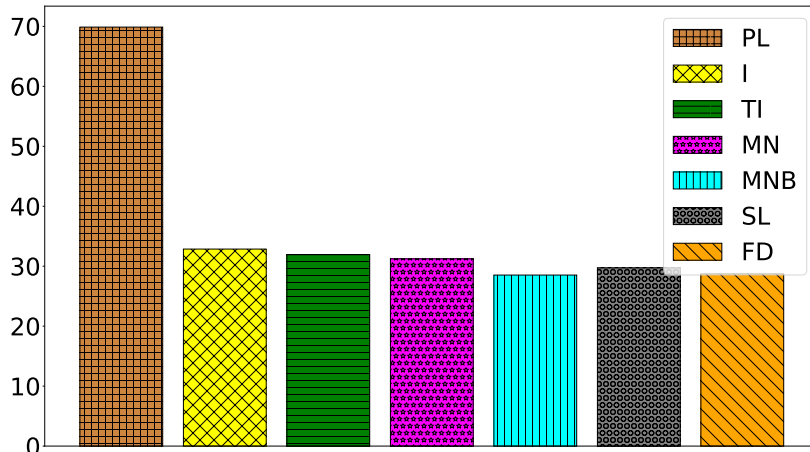


Fig. 8.3. Mean success rate on validation data based on prompt context type when they are applicable.

8.D.2. Ablation on Performance based on Prompt Proposal

Figure 8.3 shows the mean success rate of prompt context types when success is counted only for the cases when these prompt contexts are applicable. As can be seen from the figure, post lines is the most useful prompt context type on average. The contribution from other prompt context types though smaller than post lines is still significant highlighting the importance of each prompt context type.

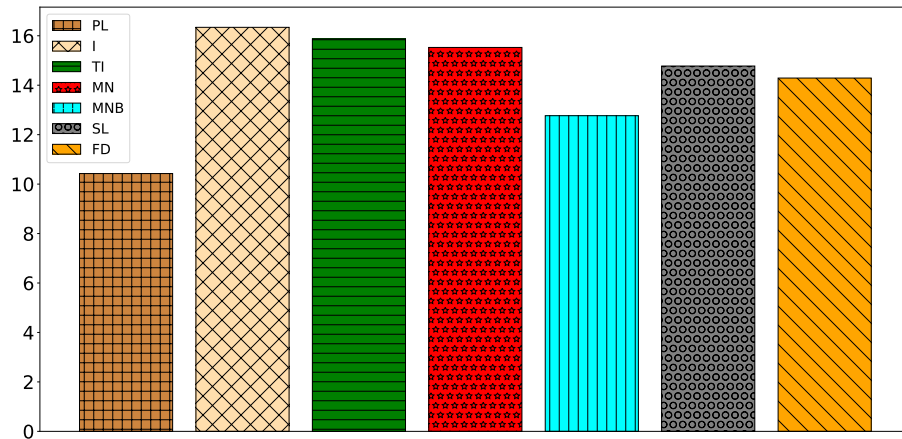
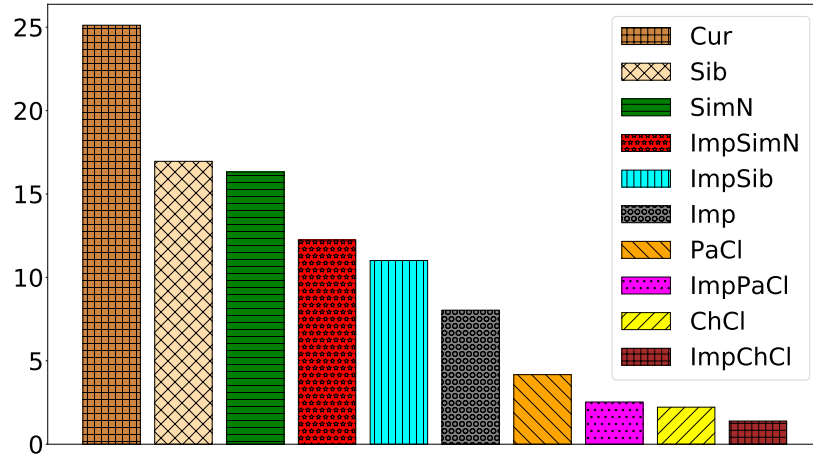


Fig. 8.4. (*Top*) Normalized success rate of prompt sources when applicable, (*Bottom*) Normalized success rate of prompt context types when applicable

Figure 8.4 shows the normalized success rates where the normalization is performed across the prompt proposals. This helps us understand the relative performance of prompt proposal sources and context types. The top part of the figure breaks down the performance based on prompt sources and the bottom part breaks down based on prompt context types. One thing to note from the plot of prompt context types is that when we consider relative performance, post lines is no longer the most dominant context type. This is because post lines is tied to only when the prompt source corresponds to the current file, thereby contributing to lower numbers when compared to most of the other context types that are tied to all prompt sources.

8.D.3. Performance on non-immediate Post Lines

Table 8.9 shows the performance of post lines when starting from the fourth line after the target hole line (i.e., skipping three lines after the target hole) as opposed to starting from the line that immediately follows the target hole. This experiment helps us understand the performance when we are interested in doing a much harder task of multi-line code autocompletion, wherein the objective is to predict not just the blanked out portion in the current line but also the next three lines which can correspond to completing a block of code like a function body. From the table, we see that when starting from the fourth line, a slight deterioration in performance occurs. This is expected because the farther away we move from the target hole, the less relevant the post lines context would be. However, the performance drop is not significant suggesting that post lines is still a very useful prompt context type that can be used under the setting of multi-line code-autocompletion. Equivalently, we can include this as one of the prompt proposals in our framework along with the current version of post lines.

Table 8.9. Success Rate (SR) when taking different versions of post lines.

Method	Success Rate(%) (hole-wise)	Rel. ↑(%) (hole-wise)	Success Rate(%) (repo-wise)	Rel. ↑(%) (repo-wise)
Codex (Chen et al., 2021)	58.73	-	60.64	-
Post Lines (immediate line after the hole)	65.78	12.00	68.01	12.15
Post Lines (skipping three lines after the hole)	65.11	10.86	66.42	9.53

8.D.4. Composition of prompt proposals

Table 8.10 shows the performance of the two versions of RLPG when we compose the prompt proposal context from l prompt proposals. We take the top- l prompt proposals given by RLPG based on decreasing order of probability. To decide how much context should be used for each prompt proposal, we divide the total context length in proportion to the normalized probabilities of the top- l prompt proposals. As can be seen from the table, even though PPC is not explicitly trained to perform composition (both the ground-truth vector and the representation of prompt proposal context involve a single prompt proposal), all the compositions lead to significant improvements over Codex. However, as expected the best results correspond to taking context from a single prompt proposal (i.e., the training setting). The drop in success rate with $l = 2$ and $l = 5$ is not that significant, which suggests that explicitly training RLPG to learn to compose contexts from different prompt proposals can lead to promising results and hence offers an interesting future direction.

Table 8.10. Success Rate (SR) of different compositions of the prompt proposals on the test set.

Method	Success Rate(%) (hole-wise)	Rel. ↑(%) (hole-wise)	Success Rate(%) (repo-wise)	Rel. ↑(%) (repo-wise)
Codex (Chen et al., 2021)	58.73	-	60.64	-
RLPG-H ($l = 1$)	68.51	16.65	69.26	14.21
RLPG-R ($l = 1$)	67.80	15.44	69.28	14.26
RLPG-H ($l = 2$)	67.07	14.20	67.87	11.91
RLPG-R ($l = 2$)	66.57	13.35	67.88	11.94
RLPG-H ($l = 5$)	66.60	13.40	67.91	11.98
RLPG-R ($l = 5$)	65.78	12.01	67.69	11.62
RLPG-H ($l = 10$)	65.53	11.58	67.24	10.88
RLPG-R ($l = 10$)	63.59	8.27	65.98	8.79

8.D.5. Effect of Context Length

To understand the effect of context length on the performance of our prompt proposals, we took half of the context length available for a prompt in Codex and observed the performance of the oracle and fixed prompt proposal. As before, we saw that an oracle constructed from our prompt proposals shows remarkable improvement over Codex highlighting the value of our prompt proposals (see Table 8.11). However, when compared to a larger context length, the relative gains are smaller. This is expected as a smaller context length means that the relevant context coming from a prompt proposal needs to be truncated to make it fit inside the prompt, thereby leading to loss of information.

Table 8.11. Success Rate (SR) of Codex and oracle over the test set when the total context length = 2048.

Method	Success Rate(%) (hole-wise)	Rel. ↑(%) (hole-wise)	Success Rate(%) (repo-wise)	Rel. ↑(%) (repo-wise)
Codex (Chen et al., 2021)	57.77	-	58.90	-
Oracle	61.90	7.15	67.18	14.07

8.D.6. Effect of Truncation

We calculated the percentage of times the context included in the prompt gets truncated. In Table 8.12, the second, third, fourth, and fifth columns represent the percentage truncation with the default codex context, the prompt proposal context, with either of them and with both of them, respectively. The truncation numbers suggest that a code completion model that allows a longer context length can be useful. We do not explicitly encourage the syntactic correctness of the included code snippet. Since the context length is limited, it is quite possible that the included context in the prompt is not syntactically correct as a

whole. However, on plotting the repository-wise numbers, we didn’t observe any particular correlation between the amount of truncation and performance. This makes us believe that the content of the included context (whether it comes from a selected prompt proposal) is what matters more rather than whether it is syntactically correct or not.

Table 8.12. Percentage truncation when using different contexts in the prompt.

Method	Default Codex Context	Prompt Proposal Context	Either	Both	Success Rate
RLPG-H	26.95	30.95	39.22	18.68	68.51
RLPG-R	26.82	31.31	38.88	19.24	67.80

8.D.7. Performance on individual repositories

Table 8.13. Success Rate of different methods on training data

Repo name	#Total Holes	Oracle	Codex	Fixed prompt proposal	RLPG-H	RLPG-R
largemail	1653	75.38	55.11	62.73	63.94	63.28
ftpserverremoteadmin	7323	86.44	66.11	76.09	76.21	76.76
myt5lib	838	91.65	53.58	61.34	73.51	74.46
seamlets	4890	92.74	62.25	62.72	71.55	74.27
gloodb	10000	91.07	57.50	57.50	70.32	72.31
jjskit	9043	80.36	65.61	72.18	72.00	72.44
mobileexpensetracker	2298	75.94	57.88	67.28	66.84	66.97
gfsfa	10000	80.55	57.33	57.33	59.28	65.24
swe574-group3	2029	76.79	54.46	66.19	65.16	64.91
strudem-sicsa	6131	77.83	64.96	72.55	73.25	73.32
soap-dtc	1370	81.24	64.82	70.73	71.61	72.70
openprocesslogger	7191	81.06	62.19	71.77	72.22	72.62
tapestry-sesame	397	72.54	45.84	61.21	60.71	63.98
exogdx	735	84.76	63.81	75.51	75.92	76.60
designpatternjavapedro	1069	78.30	54.82	64.36	63.99	68.57
quidsee	3020	81.66	60.79	69.50	70.36	70.26
healpix-rangeset	4734	63.54	48.71	54.67	54.94	55.07
sol-agent-platform	10000	73.76	58.22	65.72	65.65	65.94
rsbotownversion	10000	75.23	57.89	65.58	66.22	66.31

Table 8.14. Success Rate of different methods on validation data

Repo name	#Total Holes	Oracle	Codex	Fixed prompt proposal	RLPG-H	RLPG-R
tyrond	721	83.91	60.33	71.15	71.57	72.68
math-mech-eshop	2225	83.46	62.20	72.76	73.53	73.17
infinispan-storage-service	373	82.31	71.85	78.55	76.94	77.75
teammates-shakthi	7665	82.02	63.74	72.38	72.47	72.46
javasummerframework	10000	79.27	55.92	65.30	65.74	65.55
tinwiki	10000	73.67	69.27	69.27	69.12	69.58
jloogle	3145	84.55	73.16	77.87	77.17	77.36
jcontenedor	5464	81.26	58.99	67.77	67.95	68.32
sohocms	772	76.68	57.90	67.10	67.49	67.62
affinity_propagation_java	1466	79.54	59.14	70.33	70.26	70.26
jata4test	1921	71.06	44.09	54.92	55.91	57.47
swinagile	2595	79.69	63.01	72.29	72.49	72.68
navigablep2p	1322	75.72	59.76	65.43	65.13	65.28
springlime	879	83.50	62.34	74.18	74.86	74.40

Table 8.15. Success Rate of different methods on test data

Repo Name	#Total Holes	Oracle	Codex	Fixed PP	RLPG-H	RLPG-R	Random	Random NN	Iden Usage (Random)	Iden Usage (NN)	File-Level BM25	RLPG-BM25
dovetaildb	10000	76.89	57.12	66.45	66.06	66.25	57.45	57.58	61.39	60.77	59.39	66.09
project-pt-diaoc	10000	82.01	52.67	52.81	65.08	61.25	51.58	52.93	55.54	56.21	57.04	58.29
realtimegc	2513	77.64	57.58	67.01	67.85	68.48	57.78	58.89	63.51	63.99	61.84	66.69
fswuniceubtemplates	2070	77.44	55.7	58.89	66.81	65.8	55.22	55.89	65.7	66.43	59.28	66.71
qwikioffice-java	1138	76.45	70.21	70.21	69.86	70.56	46.13	48.15	60.37	62.92	64.41	58.17
glperaudsimon	1766	78.65	53.57	62.51	62.4	61.66	55.66	57.76	69.42	68.4	69.14	61.55
xiaonei-java-ap	839	73.42	57.57	62.1	62.69	63.29	57.09	57.21	71.28	72.35	63.77	63.29
ircrpgbot	6591	83.67	69.67	77.24	76.71	76.65	69.55	70.54	74.68	74.43	69.32	75.75
robotsimulator2009w	7514	75.63	56.28	67.55	67.53	67.55	56.4	56.18	64.61	64.71	62.96	66.12
gwt-plugindetect	73	84.93	60.27	68.49	65.75	68.49	58.9	57.53	63.01	63.01	50.68	75.34
apitfriends	1385	85.05	65.05	74.8	75.67	75.31	65.7	68.59	70.25	70.11	66.93	73.57
wicketbits	754	83.02	59.81	72.94	72.81	73.08	60.21	61.94	81.96	79.31	84.48	73.47
hucourses	590	84.41	70.68	77.46	77.63	77.97	70	72.2	70.68	72.54	53.39	75.08
xfuze	3055	84.09	62.82	73.62	72.73	73.62	63.67	65.17	77.25	75.97	77.32	74.01

Table 8.13, Table 8.14 and Table 8.15 present the success rates of different methods over individual repositories in the training, validation and test splits, respectively. The repo-wise averages in Table 2 in the main paper were calculated by taking the average of numbers corresponding to each column. The hole-wise averages correspond to multiplying the repo-wise numbers of each method by the total holes in the repo to get the total number of successful holes by that method for that repo. We then add the total number of successful holes across repos and divide it by the total number of holes in the entire data split to get the hole-wise averages.

8.E. Analysis of Sample Cases

In Figure 8.1, RLPG selects the prompt proposal that corresponds to taking method names and bodies from the imported file (i.e. `MaximizingGibbsSampler.java`). Note that `mg.` before the hole position indicates that a method used in the imported file is likely to be invoked. In this case, the prompt proposal context (highlighted in violet) contains the

method name `InitializeToAssignment`(part of target hole). This in conjunction with the default Codex context which contains the method `CurrentAssignments()`(part of target hole) leads to generation of a successful prompt. On the other hand, the prompt created from the default Codex context fails to predict the target hole in this case. In general, we observed that in the absence of a strong signal, Codex has a tendency to give preference to natural language comments occurring before the hole position, e.g. naming the method based on the comment. This in certain cases might hurt. We provide insatnces of positive and negative samples cases for RLPG below:

8.E.1. Positive Cases

We provide some examples of cases where RLPG led to the correct prediction and Codex failed.

- (1) Cases where part of the target hole is found exactly in the prompt proposal context.
 - RLPG = `Propagation(int numVars)` vs Codex = `Propagation()`
 - RLPG = `tersFromFile(String filename) {` vs Codex = `ters(String filename) {`
 - RLPG = `als("dampingFactor")) {` vs Codex = `als("numVars")) {`
 - RLPG = `] + ", value = " + entries[1]);` vs Codex = `]);`
 - RLPG = `stem.exit(1);` vs Codex = `stem.err.println("DPAP load error: " + ex.get`
- (2) Cases where Codex takes strong hint from the preceding natural language comment, thereby producing incorrect predictions.
 - RLPG = `d PassMessages()` vs Codex = `d DoOneRoundOfMessagePassing()`
 - RLPG = `teger> CurrentExemplars() {` vs Codex = `teger> ChooseExemplars() {`
 - RLPG = `ring FileName() {` vs Codex = `ring GetAlgorithmFilename() {`

8.E.2. Negative Cases

In certain cases, extra information from prompt proposal-context might lead to confusion and produce incorrect predictions.

- RLPG = `an hasConverged_;` vs Codex = `an converged_;`
- RLPG = `_[i][j] = -Double.MAX_VALUE;` vs Codex = `_[i][j] = 0;`

Chapter 9

Prologue to the fourth article

9.1. Article Details

RepoFusion: Training Code Models to Understand Your Repository. Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, Torsten Scholak. This article ([Shrivastava et al., 2023](#)) is under review at the *Neural Information Processing Systems (NeurIPS) 2023*.

Personal Contribution The project began with discussions between Disha Shrivastava, Dzmitry Bahdanau and Torsten Scholak preceding Disha Shrivastava’s part-time internship at ServiceNow Research. Having worked with and seen the promise of repository-level context ([Shrivastava et al., 2022](#)), Disha Shrivastava was looking into ways of training with repository contexts from multiple sources. Dzmitry Bahdanau suggested that Fusion-in-Decoder ([Izacard & Grave, 2021](#)) can be an effective strategy to do so. Further refinement of this idea through subsequent discussions, led to the development of RepoFusion. Denis Kocetkov contributed to the initial filtration process of The Stack ([Kocetkov et al., 2022](#)) dataset and assisted in scripting and conducting experiments for the finetuning of the CodeT5 model, which was used to initialize RepoFusion. Disha Shrivastava was responsible for formulating the RepoFusion framework, writing the code as well as running experiments for obtaining repository contexts from The Stack, data pipeline, training and evaluation of RepoFusion as well as writing the paper. Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak advised on the project and offered constructive critique during the drafting and revision phases of the paper.

9.2. Context

Despite the huge success of LLMs in coding assistants like GitHub Copilot ¹, these models struggle to understand the context present in the repository (e.g., imports, parent classes,

¹<https://github.com/features/copilot/>

files with similar names, etc.), thereby producing inaccurate code completions. This effect is more pronounced when using these assistants for repositories that the model has not seen during training, such as proprietary software or work-in-progress code projects. In this work, we build upon the concept of utilizing repository context during inference from our previous work [Shrivastava et al. \(2022\)](#) and propose *RepoFusion*, a framework to train models to incorporate relevant repository contexts from multiple sources. In terms of the broader theme of the thesis (see Section 1.1), RepoFusion serves as the *Predict* module that provides an effective way to combine relevant support context Z from diverse code snippets taken from the repository thereby helping the model to generate a more accurate and context-aware prediction for the current line (target Y). Our most successful model uses context coming from the prompt proposals used in our prior work (see Chapter 8). In contrast to our prior work, where the *Enhance* module - represented by RLPG - selects a single prompt proposal, RepoFusion learns to combine relevant context from multiple prompt proposals. Therefore, RepoFusion can also be viewed as carrying out the operations of both the *Enhance* and *Predict* modules. The subsequent chapter (Chapter 10) discusses this work in detail.

9.3. Contributions

In this work, we propose RepoFusion, a framework that helps code models to make better predictions by learning to combine relevant contextual cues from the repository. Experiments on single-line code completion show that our models trained with repository context significantly outperform much larger code models as CodeGen-16B-multi ($\sim 73\times$ larger) and closely match the performance of the $\sim 70\times$ larger StarCoderBase model that was trained with the Fill-in-the-Middle objective. We create and release *Stack-Repo*, a dataset of 200 Java repositories with permissive licenses and near-deduplicated files that are augmented with three types of repository contexts. In addition, we [open-sourced](#) the code and trained models for the work to facilitate future research in this direction.

9.4. Research Impact

This work is a novel and compelling demonstration of the gains that training with repository context can bring even with a significantly smaller model. One of the crucial findings of our work is that leveraging information from diverse sources within a repository is key for improved performance. This opens up avenues for extending our work to incorporate sources of relevant context other than the current repository such as tutorials, API documentation, and other related repositories within the organization.

Chapter 10

RepoFusion: Training Code Models to Understand Your Repository

10.1. Introduction

Large Language Models (LLMs) of code (Svyatkovskiy et al., 2020; Chen et al., 2021; Fried et al., 2022; Wang et al., 2021b; Nijkamp et al., 2023b; Li et al., 2022; Allal et al., 2023) have gained significant popularity. The demand for these models has further increased with their integration into code assistants like GitHub Copilot¹ and TabNine², and their popularity is anticipated to grow further as more developer-assistance products are developed around them.

Despite their remarkable capabilities, LLMs of code often struggle to generalize effectively in unforeseen or unpredictable situations, resulting in undesirable predictions. Instances of such scenarios include code that uses private APIs or proprietary software, work-in-progress code, and any other context that the model has not seen while training. To address these limitations, one possible approach is to enhance the predictions of these models by incorporating the wider context available in the repository. Leveraging the structure and context of the repository can take into consideration dependencies between files, such as imports and parent classes, and provide valuable insights into coding patterns that may be specific to the organization or user. Recent works (Shrivastava et al., 2022; Zhang et al., 2023; Ding et al., 2022) have shown promising results in utilizing repository-level context in conjunction with LLMs of code. It was also shown in Shrivastava et al. (2022) that without specialized training, it is challenging to integrate multiple relevant contexts from the repository. Building upon these findings we propose RepoFusion, a training framework for learning to combine multiple relevant contexts from the repository in order to generate more accurate and context-aware code completions. In this work, we focus on the task of single-line code

¹<https://github.com/features/copilot/>

²<https://www.tabnine.com/>

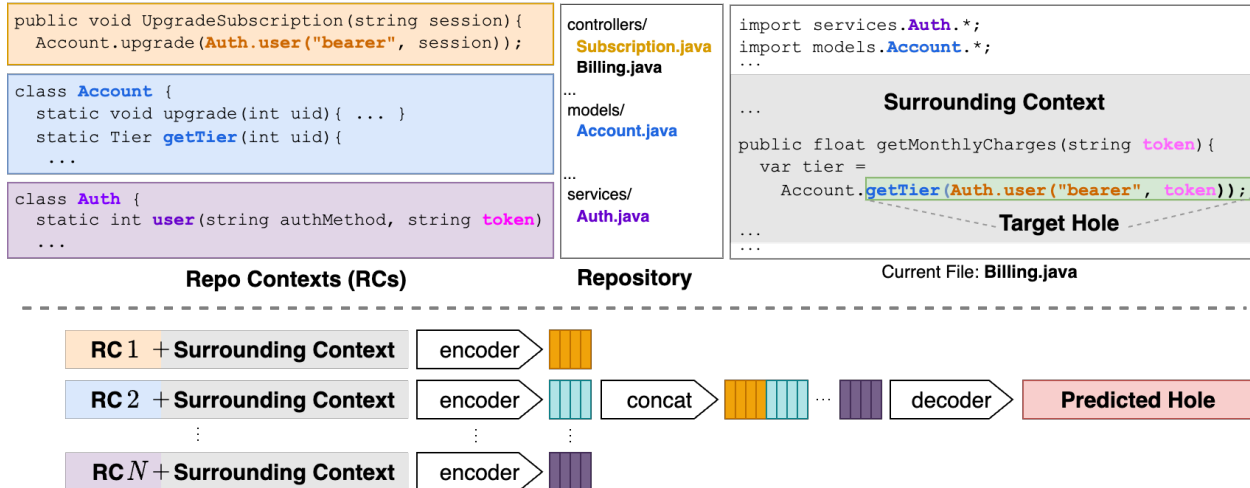


Fig. 10.1. Figure explaining the idea of RepoFusion. Given multiple relevant contexts from the repository (*Repo Contexts*), RepoFusion appends the *Surrounding Context* (highlighted in gray) to each repo context, encodes them separately, and combines them to produce a prediction of the target hole.

completion (Hellendoorn & Devanbu, 2017b; Shrivastava et al., 2020) which simulates real-world scenarios where users are editing existing files in an IDE. With reference to Figure 10.1, this means that we have to predict the missing section, referred to as the *target hole* (highlighted in green), starting from the cursor’s position until the end of the line. We see that the completion of this target hole will benefit from context not just in the current file (the variable name `token`), but also from other files in the repository. Specifically, the context from the imported file `Account.java` provides insight into the usage of the `getTier` method, while the sibling file `Subscription.java` offers guidance on the usage of `Auth.user("bearer",`, with the definition of `Auth` found in the imported file `Auth.java`. Given these relevant code snippets from across the repository which we call *Repo Contexts (RCs)*, RepoFusion uses the Fusion-in-Decoder (Izcard & Grave, 2021) architecture to combine these. Specifically, each repo context is appended with the *surrounding context* i.e., a window around the target hole excluding the target hole (highlighted in gray) and encoded separately. A decoder jointly attends to the concatenated encoded representations to produce a prediction for the target hole (highlighted in red). The key contributions of our paper can be listed as follows:

- We propose RepoFusion, a framework that helps code models to make better predictions by learning to combine relevant contextual cues from the repository.
- Through extensive experiments we establish that RepoFusion, a 220M parameter model, significantly outperforms several larger models trained on the next-token prediction objective such as CodeGen-16B (Nijkamp et al., 2023b). Furthermore, despite

being approximately 70 times smaller in size, our model closely matches the performance of StarCoderBase (Li et al., 2023), a 15.5B parameter LLM trained with the Fill-in-the-Middle (Bavarian et al., 2022) objective.

- We conduct thorough ablation studies to gain insights into the key factors influencing RepoFusion, such as the nature of repository contexts, their lengths, the number of repository contexts, and other training configurations. One of the crucial findings is that leveraging information from diverse sources within a repository is the key to RepoFusion’s effectiveness.
- We create and release *Stack-Repo*³, a dataset of 200 Java repositories with permissive licenses and near-deduplicated files that are augmented with three types of repository contexts. Our released resources can be found at: <https://huggingface.co/RepoFusion>.

10.2. Training with Repository Context

In this section, we briefly describe Fusion-in-Decoder (Izcard & Grave, 2021), the repository contexts we used, and the details of our RepoFusion framework.

10.2.1. Fusion-in-Decoder

Fusion-in-Decoder (Izcard & Grave, 2021) (FiD) is a method to train a language model to combine information coming from multiple sources. In the original work, FiD was used for open-domain question answering. In the FiD approach to question answering, a sequence-to-sequence model takes support passages concatenated with the question as inputs and produces an answer as the output. Each support passage is appended to the question and encoded independently by the encoder. The encoded representations are then concatenated and fed to the decoder which jointly attends to them to produce the answer. In this work, we adapt FiD for the setting of code completion.

10.2.2. Repository Contexts

In this work, we consider different ways of retrieving relevant code snippets from across the repository, i.e., repo contexts. Our most successful repo contexts are obtained by following the approach proposed by Shrivastava et al. (2022). Motivated by the syntax and semantics of programming languages as well as the common coding patterns, Shrivastava et al. (2022) proposed a set of repo-level prompt proposals that leverage the structure and the relevant context in files across the repository. A prompt proposal (PP) is a function that takes in the target hole’s location and the associated repository as input and returns a string called *Prompt Proposal Context (PPC)* as output. The prompt proposal context is created

³<https://huggingface.co/datasets/RepoFusion/Stack-Repo>

by extracting a particular type of context (prompt context type) from a particular category of related source files (prompt source). Examples of prompt sources are the current file, files that are imported into the current file, files that have a similar name as the current file, etc. Examples of prompt context types are lines following the target hole, method names and bodies, identifiers, string literals, etc. Combining these prompt sources and prompt context types gives us a total of 63 prompt proposals (see Appendix B.4 of [Shrivastava et al. \(2022\)](#) for details). It should be noted that the context from the beginning of the current file up to the position of the hole, as well as the context following the target hole within the current file are also types of prompt proposal contexts. We will refer to these as the prior PPC (or just *prior*) and the post PPC (or just *post*), respectively in the remainder of the paper. Note that depending on the target hole, some prompt proposal contexts might be empty (e.g. if the target hole is towards the very beginning of the file, there might not be any import statements from the current file to get context from).

Repo-level prompt proposals can be thought of as a deterministic retrieval mechanism that returns the relevant code snippets from the repository. We also consider two other mechanisms for retrieving repository-level context (see Appendix 10.B.3 for implementation details): (a) BM25: The context from each file in the repository is scored using BM25-based ([Jones et al., 2000](#)) similarity with the surrounding context, and (b) RandomNN (also used in [Shrivastava et al. \(2022\)](#)): From a list of randomly selected chunks from the repository, we select the top-k based on the similarity of the embedded chunks with the embedded surrounding context in the representation space. The PPC along with BM25 and RandomNN gives us three types of repo contexts.

10.2.3. RepoFusion

The core idea of RepoFusion is to train a code model to be aware of the context in the repository such that it helps in generating an accurate prediction of the target hole. Given a set of repo contexts, RepoFusion learns to combine the relevant parts of these contexts using the FiD approach as described in Section 10.2.1. The surrounding context is concatenated with each repo context and then encoded independently (see Figure 10.1, bottom). Note that for our purpose, since we want the code model to complete the target hole, we append the surrounding context toward the end of the repo context. This is different from the original work ([Izacard & Grave, 2021](#)), where the question (analogous to the surrounding context in our setting) is appended at the beginning of each passage (analogous to the repo context in our setting). RepoFusion uses N repo contexts of length l tokens each. Since the number and exact length of the prompt proposal contexts varies depending on the target hole, there can be different ways to map the PPCs to RCs. We experimented with four strategies for

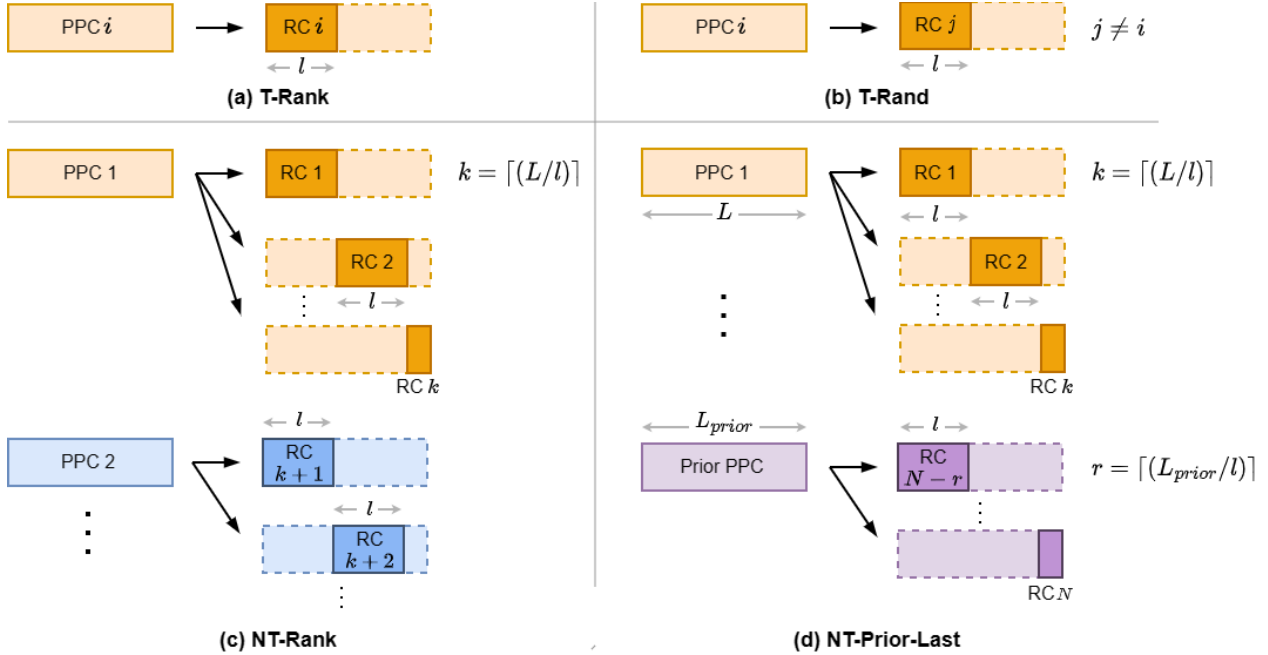


Fig. 10.2. Different strategies employed for producing repo contexts (RCs) from prompt proposal contexts (PPCs): **(a) T-rank:** we truncate the i -th ranked PPC to yield the i -th RC. **(b) T-rand:** we position the truncated i -th PPC at a random position j in RepoFusion’s sequences of RCs. **(c) NT-Rank:** each PPC yields as many RCs as necessary to exhaust all of its tokens without truncation. **(d) NT-Prior-Last:** we reserve the last r RCs for the Prior PPC and fill the rest RCs as in NT-Rank.

producing and ordering the RCs based on PPCs that express different aspects of the mapping (see Figure 10.2). We mention the strategies along with the motivation for each below.

- (1) **Truncated-Ranked (T-Rank):** In this setting, one prompt proposal context yields one repo context. We truncate each prompt proposal context (i.e., take only the first l tokens) to form the respective repo context and discard the rest. The repo contexts are ordered based on the ranking of the prompt proposals⁴ on the validation split of the Google Code archives dataset of Shrivastava et al. (2022). Given that our work and Shrivastava et al. (2022) both target Java, it seemed reasonable to us to directly use this ordering.
- (2) **Truncated-Random (T-Rand):** Same as T-rank except that the repo contexts are ordered randomly. This helps us understand the role of the specific ranking of PPs from Shrivastava et al. (2022).
- (3) **Not Truncated-Ranked (NT-Rank):** The prompt proposals are ranked based on the same order as in T-Rank. Unlike T-rank, here we avoid the truncation of prompt proposal contexts. Instead, we construct as many repo contexts from each prompt proposal context as necessary, namely a PPC of length L will contribute $k = \lceil (L/l) \rceil$

⁴https://github.com/shrivastavadisha/repo_level_prompt_generation/blob/main/get_info_from_hole_predictions.py

RCs. We then proceed to the next in order prompt proposal and continue so until we have selected N repo contexts. Unlike T-rank, this setting allows RepoFusion to see the entirety of top-ranked prompt proposals at the cost of potentially ignoring the lower-ranked ones.

- (4) **Not Truncated-Prior-Last (NT-Prior-Last)**: Same as NT-Rank except that the prior PPC is always ordered at the end. Since the decoder attends to the concatenated encoded representations of the repo contexts in the same order as it is presented as inputs, this strategy helps us understand the role of continuing code generation from the encoded representation of the prior PPC as the most recently attended representation in the decoder. Note that depending on the value of N , it may be necessary to remove certain chunks of top-ranked PPCs in order to accommodate the prior PPC at the end.

Similar to Izacard & Grave (2021), we format each repo context with special tokens to mark the beginning of the surrounding context and the repo context, as well as for the name of the repo context (which is the same as the name of the prompt proposal). Please see the Appendix 10.B for details on these tokens and other architectural details of RepoFusion.

10.3. Experiments and Results

In this section, we describe the process of creation of our dataset Stack-Repo and the details of experiments. We then present the results of evaluating RepoFusion and other models on the test set. This is followed by presenting the findings of extensive ablation studies carried out on the validation set to gain deeper insights into the individual contributions of each component in our framework.

10.3.1. Dataset Creation

In this work, we build upon a modified version of The Stack V1.1 (Kocetkov et al., 2022). The modified version ⁵ consists of near-deduplicated code repositories with permissive licenses from GitHub. For our experiments, we take only the Java subset (files with .java extension) of this dataset.

Creation of Target Holes: For creating target holes needed for training and evaluating RepoFusion, we choose a set of repositories randomly from the Java subset of the Stack and divide them into training, validation, and test splits in the ratios 2:1:1. We only consider repositories that contain at least 20 near-deduplicated files. For each repository, we choose target holes from every code line (excluding comments in natural language and blanks) in all the files. In order to tokenize a code line, we used common Java code delimiter

⁵<https://huggingface.co/datasets/bigcode/the-stack-dedup>

tokens ⁶. We chose a random token within the line and the rest of the line starting from that position till the end constitutes the target hole. By not selecting target holes based on the tokenizer of a specific code model, we can ensure that the tokenizer remains unbiased and does not implicitly favor any particular code model in our experiments. To avoid bias from large repositories while training, we cap the maximum contribution of target holes from a repository to 10000, i.e. if the total number of holes in the repository exceeds 10000, we select 10000 holes randomly from the total holes. Please see Table 10.1 for the statistics of Stack-Repo.

Table 10.1. Statistics of Stack-Repo

Feature	Train	Val	Test
# Repositories	100	50	50
# Files	20310	11172	13202
# Holes	435890	220615	159822

Creation of Repo Contexts: For each target hole, we use the implementation ⁷ from Shrivastava et al. (2022) to extract prompt proposal contexts. We take two lines above and two lines below the target hole excluding the target hole as the surrounding context. For obtaining the embeddings for RandomNN repo contexts, we use pre-trained CodeBERT (Feng et al., 2020). For constructing the BM25 repo contexts, we use the implementation from the Rank-BM25 package ⁸. To improve efficiency, we store the repo contexts for each target hole in advance. Note that even though our target hole and repo context creation strategies have been inspired from Shrivastava et al. (2022), our dataset, Stack-Repo is significantly bigger in size. Apart from code completion, Stack-Repo can serve as a benchmark for various other code-related tasks involving repository context, such as bug repair and pull request resolution. We plan to release it under the same license as The Stack (Kocetkov et al., 2022) to support future research in these areas.

10.3.2. Experimental Details

Training of RepoFusion: We use the 220M parameter CodeT5-base (Wang et al., 2021b) encoder-decoder model as our base code model for RepoFusion. We found that the pre-trained CodeT5 model was not good at completing Java code (see Appendix 10.C.3 for initial results). Therefore, to obtain a base model for RepoFusion training we finetuned CodeT5-base with an input context length of 512 using the next-token prediction objective on Java repositories from the dataset described in Section 10.3.1. Specifically, we used the

⁶[., (,), [,], , {, }, ,, :, ", ;]

⁷https://github.com/shrivastavadisha/repo_level_prompt_generation

⁸<https://pypi.org/project/rank-bm25/>

repositories that were not included in Stack-Repo. For each file, we randomly sample ten pivot points with the code context prior to the pivot location in the file serving as the input to the encoder of CodeT5. The finetuned CodeT5-base model was then used to initialize the training of RepoFusion. Based on the validation set performance, we found that for RepoFusion, NT-Prior-Last with $N = 32$ and $l = 768$ works the best. We provide complete details of training RepoFusion and finetuning CodeT5 in the Appendix 10.B.

Baselines: To benchmark the performance of RepoFusion, we conducted experiments with several methods, with each model utilizing the recommended tokenizers specific to the method and employing a maximum token generation limit of 128 per completion. To ensure a thorough analysis, we have incorporated encoder-decoder models as well as decoder-only models of different sizes, with varying context lengths and two different input context types. We present the details of the methods below:

- (1) **CodeT5 (FT):** In addition to the previously described finetuned (FT) version of **CodeT5-base**, we also finetuned **CodeT5-large** (770M) with a context length of 512. Next, we assessed the performance of these models using input context lengths of 2048 and 4096. The input context was constructed by either considering the prior PPC (*prior*) alone or by concatenating equal lengths of the post PPC (*post*) and prior.
- (2) **BigCode models:** We experimented with two models released by BigCode⁹. The first model is **SantaCoder** (Allal et al., 2023), which is a 1.1B parameter model which supports a maximum context length of 2048 tokens and the second is the recently released **StarCoderBase** (Li et al., 2023) model which is a 15.5B parameter model that can support up to 8192 tokens. Both of these models are trained with the Fill-in-the-Middle (Bavarian et al., 2022) (FiM) objective on versions 1.1 and 1.2 of The Stack (Kocetkov et al., 2022), respectively. These models were evaluated with both the prior and post+prior contexts as inputs. For experiments with post+prior, we used the FiM special tokens that were used while training these models. Since these models have been trained specifically to see the post PPC as suffix, they help us understand the role of training with multiple repo contexts in the way proposed by RepoFusion.
- (3) **CodeGen** (Nijkamp et al., 2023b): CodeGen is a decoder-only transformer-based autoregressive model trained with the next-token prediction objective. It supports a maximum context length of 2048 tokens. We experimented with three pre-trained variants of CodeGen, namely **CodeGen-2B-multi**, **CodeGen-6B-multi**, and **CodeGen-16B-multi**. As before, we tried the scenarios where the input context consists of the post + prior as well as when the input context consists of just the

⁹<https://www.bigcode-project.org/>

prior. These models help us understand the performance of large pre-trained models that are not trained with repo context.

It is important to note that when compared to our RepoFusion model, with the exception of CodeT5-base (FT), all other models are many times larger in size and have been trained on a significantly larger number of tokens. The rationale behind selecting these baselines is to compare the performance of training smaller models with additional repository context against training much larger models without incorporating repository context.

Evaluation Metric: We conduct an exact string match between the predicted hole and the target hole, where the predicted hole is the string up to the occurrence of the first newline in the completion. If an exact match is found, it is considered a success; otherwise, it is deemed a failure. We measure the fraction of exact matches over the dataset and call it *Success Rate*.

10.3.3. Results

Table 10.2 presents the hole completion success rate (along with standard error) in percentage for different methods on our test set, where the standard error is an estimate of the variability in the sample mean of the distribution of exact match. The top two sections of the table display the evaluation results of the finetuned encoder-decoder {CodeT5-base(FT), CodeT5-large(FT)} models and decoder-only {SantaCoder, CodeGen-2B, CodeGen-6B, CodeGen-16B} models, respectively when provided with prior context as input. The table’s next two sections present the results of evaluating these models when given with post+prior context as input. In the final section of the table, we showcase the evaluation results of RepoFusion using different effective input context lengths obtained by varying the values of N and l .

Baseline Performance Improves with Model Size and the Addition of Context: The performance of CodeT5 (FT) models improves as the model becomes bigger (CodeT5-large vs CodeT5-base) and as the input context length increases (2048 vs 4096). We observe a comparable pattern with decoder-only models, where there is a general enhancement in performance as the models grow larger (with a slight indication of saturation) while maintaining a fixed context length. Additionally, we note a substantial improvement in both categories of models when provided with post + prior context as input, compared to their respective performances with only the prior context. The SantaCoder model, specifically trained for the FiM task, exhibits the most significant improvement.

RepoFusion is Effective: RepoFusion not only exhibits a substantial improvement over its base model (CodeT5-base (FT)) but also surpasses other bigger models, even when utilizing the same effective context length. Furthermore, RepoFusion achieves superior performance compared to the significantly larger CodeGen-16B model, even when constrained to

Table 10.2. Completion success rate on the test set for different methods.

Model	Size (#params)	Effective context length	Context type	Success Rate (%)
CodeT5-base (FT)	0.22B	2048	prior	41.82 \pm 0.12
CodeT5-base (FT)	0.22B	4096	prior	46.45 \pm 0.12
CodeT5-large (FT)	0.77B	2048	prior	44.73 \pm 0.12
CodeT5-large (FT)	0.77B	4096	prior	48.92 \pm 0.12
SantaCoder	1.1B	2048	prior	39.51 \pm 0.12
CodeGen	2B	2048	prior	49.45 \pm 0.12
CodeGen	6B	2048	prior	49.19 \pm 0.12
CodeGen	16B	2048	prior	50.20 \pm 0.12
CodeT5-base (FT)	0.22B	2048	post+prior	48.89 \pm 0.12
CodeT5-base (FT)	0.22B	4096	post+prior	49.97 \pm 0.12
CodeT5-large (FT)	0.77B	2048	post+prior	51.72 \pm 0.12
CodeT5-large (FT)	0.77B	4096	post+prior	52.43 \pm 0.12
SantaCoder	1.1B	2048	post+prior	56.78 \pm 0.12
CodeGen	2B	2048	post+prior	53.18 \pm 0.12
CodeGen	6B	2048	post+prior	54.03 \pm 0.12
CodeGen	16B	2048	post+prior	54.09 \pm 0.12
RepoFusion ($N = 4, l = 512$)	0.22B	2048	NT-Prior-Last	65.96 \pm 0.12
RepoFusion ($N = 8, l = 512$)	0.22B	4096	NT-Prior-Last	70.38 \pm 0.11
RepoFusion ($N = 32, l = 768$)	0.22B	24576	NT-Prior-Last	77.32 \pm 0.10

Table 10.3. Comparison with StarCoderBase on a test set subset.

Model	Size (#params)	Effective context length	Context type	Success Rate (%)
StarCoderBase	15.5B	8192	prior	52.97 \pm 0.45
StarCoderBase	15.5B	8192	post+prior	79.79 \pm 0.36
RepoFusion ($N = 16, l = 512$)	0.22B	8192	NT-Prior-Last	73.67 \pm 0.43
RepoFusion ($N = 32, l = 2500$)	0.22B	80000	NT-Prior-Last	78.33 \pm 0.37

utilize fewer repository contexts to match the effective context length of CodeGen-16B. Furthermore, when provided with additional repo contexts, RepoFusion demonstrates further enhancements in performance.

We also compare RepoFusion with the recently released StarCoderBase (Li et al., 2023) model. StarCoderBase is a 15.5B parameter model which is trained with about one trillion tokens using a FiM objective employing a large input context length of 8192 tokens. The results of this comparison using a random subset of 12500 holes from our test set appear in Table 10.3. Learning to read additional repository context allows RepoFusion to

achieve success rate just 1.3% below the performance of the 70 times bigger state-of-the-art StarCoderBase model.

Prompt Proposals Matter: The right side of Figure 10.3 illustrates the success rate of RepoFusion using Random-NN, BM25, and PPC (refer to Section 10.2.2 for details) when employing T-Rank and NT-Rank. Note that when evaluating Random-NN and BM25, we employed corresponding RepoFusion models specifically trained to accept Random-NN and BM25 contexts as inputs. The results show that using the repo context from PP (Shrivastava et al., 2022) performs the best.

The NT-Prior-Last Strategy is Most Effective: Next, we compare performances of the four different repo context production and ordering strategies that we introduced in Section 10.2.3. The left side of Figure 10.3 illustrates the success rate for the four strategies in two distinct settings: $N = 32, l = 768$ and $N = 63, l = 512$. We see that the ordered repo contexts, specifically NT-Prior-Last, NT-Rank, and T-Rank perform better than random ordering of repo contexts (T-Rand). Also, the improved performance of NT-versions when compared to the T-versions, highlights the value of presenting complete context from top prompt proposals, as opposed to displaying truncated contexts from more prompt proposals.

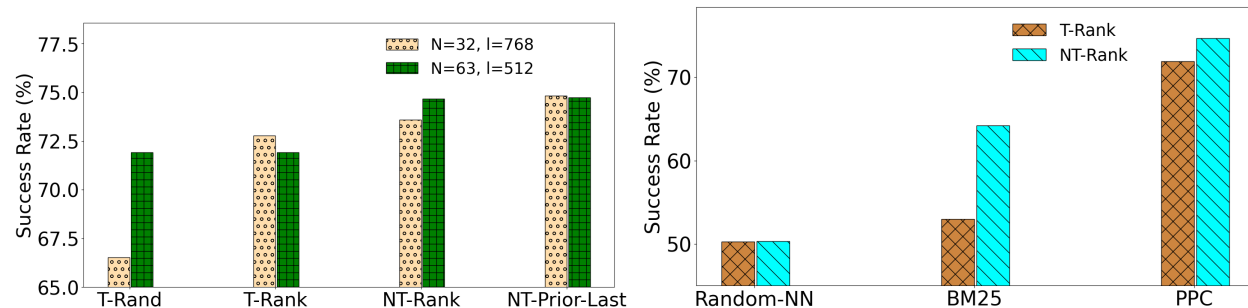


Fig. 10.3. Completion success rate with different approaches to producing repository contexts (RCs). *(Left)* Impact of RC production and ordering strategies; *(Right)* Impact of different RC retrieval methods.

Longer Repo Contexts are Better: In the top part of Figure 10.4, we plot the variation of success rate with different values of the repo context length l . For this experiment, we used our best-performing model that was trained using NT-Prior-Last. The results indicate an improvement in the performance with the size of each repo context. However, in both cases ($N = 32, N = 63$), the performance reaches a saturation point as the value of l increases.

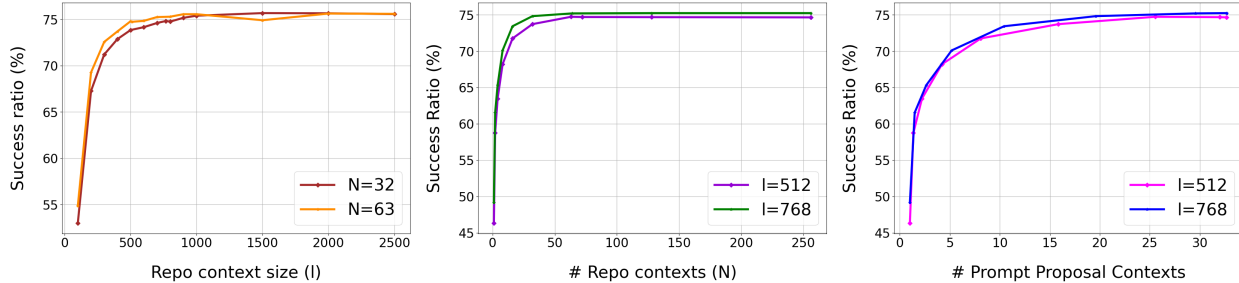


Fig. 10.4. Completion success rate as a function of (*Left*) the length of the individual repo context l ; (*Middle*) the number of repo contexts N ; (*Right*) the number of prompt proposal contexts that were used to produce the N repo contexts.

Using More Repo Contexts is Better: To understand the role of multiple repo contexts, we evaluated our best-performing model with different values of N . We see from the middle part of Figure 10.4 that the performance of RepoFusion increases up to $N = 63$ when $l = 512$ and up to $N = 32$ with a longer length of repo context $l = 768$. After this, increasing the value of N doesn't lead to further improvements.

RepoFusion Benefits from Diverse Prompt Proposals: We additionally look at the success rate as a function of the average number of different prompt proposal contexts that produced the considered repo contexts for each number of repo contexts N . Note that the number of PPCs is less than N because often one PPC yields multiple RCs. One can see from the right part of Figure 10.4 that using many diverse PPCs was essential for getting better performance with RepoFusion.

Finetuning the Base Model for Next Token Prediction is Important: Table 10.4 shows the results of evaluating RepoFusion when the corresponding model is trained by initializing with a pretrained CodeT5-base model versus initializing with a finetuned version. We observe that while training with repo contexts enhances the performance of the base pretrained CodeT5 model (see Appendix 10.C.3 for the performance with pretrained CodeT5), we see that in all cases, there are clear benefits of initializing the training with a model that is finetuned for code completion.

Table 10.4. Completion success rate when initialized from a pretrained vs finetuned model.

	Pretrained	Finetuned
T-Rand	54.67±0.50	66.53±0.47
T-Rank	59.57±0.49	72.78±0.45
NT-Rank	60.88±0.49	73.60±0.44
NT-Prior-Last	61.91±0.49	74.82±0.43

10.4. Related Work

Information from Outside the Current File: In the context of source code, harnessing information beyond the current file has been found to be useful. [Hellendoorn & Devanbu \(2017b\)](#) utilizes a nested n-gram model with a locality-based cache encompassing all directories in the repository. To capture the structure of the repository, [Pashakhanloo et al. \(2022b,a\)](#) convert it into a relational database and propose a graph-walk mechanism whereas, [Lyu et al. \(2021\)](#) incorporates the API-dependency graph in its LSTM-based code model. While training the kNN-LM ([Khandelwal et al., 2020](#)), [Xu et al. \(2022b\)](#) incorporates three types of binary variables corresponding to the presence or absence of similar local and global hierarchy. [Zhang et al. \(2021\)](#) leverages the parent class to generate comments for the child class.

Repository-level Context for Inference in LLMs: [Shrivastava et al. \(2022\)](#) proposes RLPG, a classifier that selects a prompt proposal based on the target hole and utilizes the context from the chosen prompt proposal and prior context to prompt Codex ([Chen et al., 2021](#)). Similarly, RepoCoder ([Zhang et al., 2023](#)) iteratively refines the prediction of the target hole by injecting the previous predictions from the LLM in addition to the retrieved context and prior context in the input prompt. In this work, we build upon the insights gained by [Shrivastava et al. \(2022\)](#) regarding the utilization of repository-level information during inference. We extend their findings to various configurations involving different code language models (LLMs), considering a range of context lengths and sizes. Additionally, our framework is trained with context from the repository and learns to effectively leverage multiple relevant contexts sourced from the repository.

Retrieval-augmented Code Models: In recent studies ([Zhou et al., 2023b](#); [Parvez et al., 2021](#); [Lu et al., 2022](#); [Zan et al., 2022](#); [Ding et al., 2022](#); [Borgeaud et al., 2022](#)), attempts have been made to enhance code LLMs by augmenting them with a sparse or dense retrieval mechanism that returns API documentation or relevant code snippets from the repository. The prompt proposals ([Shrivastava et al., 2022](#)) used in our work along with BM25 and Random-NN share similarities with these retrieval mechanisms. Note that RepoFusion is independent from the specific retrieval mechanisms employed and thus can seamlessly learn to integrate multiple retrieved contexts, even from different retrieval mechanisms.

10.5. Discussion

Limitations RepoFusion has a limitation in terms of computation scalability as it exhibits linear scaling with respect to the number of repo contexts N , leading to slower inference times for larger values of N . One possible solution to address this issue is to leverage FiDO ([de Jong et al., 2022](#)), an optimization technique for FiD that enables faster inference. Deploying RepoFusion, similar to any other code LLMs, requires careful consideration ([Chen](#)

et al., 2021). The generated code can often be challenging to understand or debug, resulting in developers spending significant time editing and revising the code (Vaithilingam et al., 2022; Mozannar et al., 2022; Barke et al., 2023; Bird et al., 2022). There can be instances where the generated code is less secure, posing potential risks (Pearce et al., 2021). Moreover, excessive dependence on these models can result in situations where users overlook errors in their code (Al Madi, 2022) or become overly self-assured, leading to the introduction of mistakes (Perry et al., 2022b).

Conclusions and Future Work We propose RepoFusion, a framework that allows training code models with multiple relevant contexts from the repository. By employing RepoFusion in experiments focused on single-line code autocompletion, we highlight the notable enhancements in performance attained through training smaller models with repository context, surpassing the results of training larger models without such context. RepoFusion, in combination with the Stack-Repo dataset, opens up exciting avenues for future research in the field of smaller retrieval-augmented LLMs for code. We believe our method can also extend to other code-related tasks such as bug repair, the merging of pull requests, and software documentation/tutorial writing.

Appendix for the fourth article

10.A. Details on Stack-Repo

We have made our dataset available at the link: <https://huggingface.co/datasets/RepoFusion/Stack-Repo>. The details of the license can be found in the Licensing Information section of the page. Stack-Repo consists of 200 near-deduplicated Java repositories (see Table 10.1 of the article for details). For each repository within a split (train, validation and test), we provide all files arranged in the directory structure within the repository along with three `.json` files that contain the PP, BM25 and RandomNN repo-contexts. One row of the `.json` file corresponds to a target hole consisting of the location of the target hole, the target hole as a string, the surrounding context as a string and a list of repo-contexts as strings.

10.B. Implementation Details

10.B.1. Finetuning CodeT5

As described in Section 10.3.2 of the article, to serve as a better initialization of RepoFusion (also served as a baseline) we finetuned a CodeT5-base model (220M parameters) with an input context length of 512 tokens using the CodeT5 tokenizer. We used an Adam optimizer with Decoupled Weight Decay Regularization (Loshchilov & Hutter, 2019) with weight decay of 0.05 and a learning rate of 4e-05. In addition, we used a linear scheduler with 100 warm-up steps, a dropout of 0.1, and gradient clipping with a max gradient norm of 1.0. To serve as a baseline, we also finetuned a CodeT5-large model (770 M parameters) with an input context length of 512. We used the same set of hyperparameters for this as mentioned before except that we used a learning rate of 1e-4. The training was carried out on 2 NVIDIA A100 GPUs with a memory of 80GB each and a batch size of 32 per GPU for the CodeT5-base model. For CodeT5-large we used 4 A100 GPUs with memory of 80GB each and a batch size of 12 per GPU. The evaluation run was carried out on a single 32GB V100 GPU with a batch size of 32 for CodeT5-base and 48 for CodeT5-large.

10.B.2. Training RepoFusion

We use the 220M parameter CodeT5-base (Wang et al., 2021b) encoder-decoder model as our base code model for RepoFusion. Our RepoFusion implementation was heavily built on top of the code released by Shrivastava et al. (2022)¹⁰, as well as the code released by Izacard & Grave (2021)¹¹. The former was used to obtain repo contexts and the latter was used for the FiD architecture.

Our best RepoFusion model was obtained by initializing the training from a finetuned CodeT5-base checkpoint (see Section 10.B.1 for details). The repo contexts used the NT-Prior-Last strategy (see Section 2.3 of the main paper for details) with 32 PP repo contexts each of size 768 tokens ($N = 32, l = 768$). Similar to Izacard & Grave (2021), we format each repo context with special tokens to mark the beginning of the surrounding context and the repo context, as well as for the name of the repo context (which is the same as the name of the PP taken from (Shrivastava et al., 2022)). We used `hole_context:` as prefix for the surrounding context, `rule_context:` as a prefix for PP repo context, and `rule_name:` as prefix for PP repo context name. We used Adam (Kingma & Ba, 2015c) optimizer with a learning rate 1e-5 and a warmup linear scheduler with 5000 warmup steps. We used gradient clipping with norm 1.0 and batch size of 1. Training was carried out on 2 NVIDIA A100 GPUs with a memory of 80GB each. Each evaluation run was carried out on a single 32GB V100 GPU.

The BM25 and Random NN versions of RepoFusion were obtained by using the same training hyperparameters as above and initialized from the same finetuned CodeT5 checkpoint except that we found that a learning rate of 2.5e-5 and the setting $N = 63, l = 512$ works the best. As before, we used NT-Prior-Last strategy and a prefix only for the surrounding context and no prefixes for repo contexts. The RepoFusion model that was initialized from a pretrained CodeT5-base version was obtained by using the same set of training hyperparameters as our best RepoFusion model but a learning rate of 1e-4 worked the best.

We release the RepoFusion models as well as the finetuned CodeT5 models at https://huggingface.co/RepoFusion/trained_checkpoints.

10.B.3. Retrieval Mechanisms

The BM25 repo contexts were obtained using the Okapi BM25 implementation with default parameters given by the pip package `rank-bm25 0.2.2`¹². The BM25 scores are calculated with the surrounding context being the query and full context from other files in

¹⁰https://github.com/shrivastavadisha/repo_level_prompt_generation (MIT License)

¹¹<https://github.com/facebookresearch/FiD> (Creative Commons Attribution-NonCommercial 4.0 International Public License)

¹²<https://pypi.org/project/rank-bm25/>

the repository being the search documents. Random NN repo contexts used the procedure followed by Shrivastava et al. (2022) using CodeBERT (Feng et al., 2020) to obtain the representations (See Appendix C.3 for details).

10.B.4. Other Baselines

We used the models available on Hugging Face hub, i.e. **Codegen-2B-multi**, **CodeGen-6B-multi**, **CodeGen-16B-multi**, **SantaCoder** and **StarCoder**. We used special FIM tokens, i.e., `<fim-prefix>` for pre context, `<fim-suffix>` for post context and `<fim-middle>` to prompt for completing the target hole. Each of these models used the recommended tokenizers and completion length of 128 tokens.

10.C. Additional Results

10.C.1. Effect of Repetition

In order to further assess the significance of diverse repo contexts, we conducted an analysis by repeating a PPC multiple times and using each repetition as a separate repo context. One can see from the right side of Table 10.5 that repeating the context from a single prompt proposal (prior, post, randomly chosen PP) has a negative impact on performance compared to using different repo contexts from multiple prompt proposals.

Table 10.5. Completion success rate with repetiting different types of PPCs multiple times.

	Success Rate(%)
Rand	37.18±0.48
Prior	50.69±0.50
Post	54.64±0.50
NT-Rank	71.92±0.45

10.C.2. Appending Surrounding Context

Table 10.6 shows the performance of RepoFusion when we do not append the surrounding context to each repo context. We see that the performance drops significantly for all strategies when compared to when the surrounding context is appended. It should be noted that for these experiments, we used our best RepoFusion model that is trained to take the concatenation of surrounding context and repo context as input. It is highly likely that a RepoFusion model trained to not append the surrounding context would suffer from much less performance drop.

Table 10.6. Completion success rate with and without appending surrounding context.

	without Surrounding Context	with Surrounding Context
T-Rand	13.89±0.35	66.53±0.47
T-Rank	25.06±0.43	72.78±0.45
NT-Rank	15.57±0.36	73.60±0.44
NT-Prior-Last	17.18±0.38	74.82±0.43

10.C.3. Performance of Pretrained CodeT5

Table 10.7 shows the performance on the test set when we directly use the pretrained CodeT5-base and CodeT5-large models. For these experiments, we use the special token `<extra_id_0>` to prompt the completion of the target hole. We see that the performance of these pretrained models is quite low, thereby creating the need to finetune these models on Java repositories on the next-token prediction objective. We see from the top section of Table 2 in the main paper that the finetuning helps a lot.

Table 10.7. Completion success rate on the test set for pretrained CodeT5.

Model	Size (#params)	Effective context length	Context type	Success Rate (%)
CodeT5-base	0.22B	512	prior	2.42 (0.04)
CodeT5-base	0.22B	2048	prior	3.94 (0.05)
CodeT5-large	0.77B	512	prior	4.56 (0.05)
CodeT5-large	0.77B	2048	prior	9.51 (0.07)

Chapter 11

Conclusion

The motivating theme of this thesis was to improve the generalization performance of deep learning models of code, especially when encountered with unseen context. To this end, we proposed different techniques for identifying and retrieving relevant contextual cues from the downstream task as well as different methods to effectively incorporate these contextual cues into the models. We evaluated our approaches in two domains, single-line code completion in an IDE and programming by examples, showing significant improvements in both these domains.

11.1. Summary of Contributions

We summarize the contributions of each of the articles in this thesis along with their key results and limitations.

- **Learning to Combine Per-Example Solutions for Neural Program Synthesis** (Chapter 4): In [Shrivastava et al. \(2021\)](#), we propose Neural Per-Example Program Synthesis (N-PEPS), a novel approach breaks the problem of finding a program that solves all examples into two stages: (a) finding programs that solve a single example (PE solutions) (b) making use of the program execution states to aggregate the PE solutions such that it leads to a program that solves all examples. For different evaluation settings, we show that when given the same time budget, N-PEPS significantly improves the success rate over PCCoder ([Zohar & Wolf, 2018](#)) and other ablation baselines. One of the limitations of this work is that our findings are based on the straight-line DSL from [Balog et al. \(2016\)](#). Though this DSL draws inspiration from real programming competition websites, it is relatively simple. A promising future direction would be to extend the general idea of N-PEPS, which involves breaking down a problem into simpler subproblems and combining their solutions, to more complex DSLs that include loops and conditionals. It would also be interesting to try out N-PEPS in connection with LLMs of code where we try to find

solutions to examples iteratively and then ask the LLM to combine the generated solutions by inserting appropriate instructions in the prompt.

- **On-the-Fly Adaptation of Source Code Models** (Chapter 6): In [Shrivastava et al. \(2020\)](#) we propose Targeted Support Set Adaptation (TSSA), an approach which selects targeted information from the local context and then uses this to learn adapted parameters, which can then be used for predicting a hole target in the current file. Our experiments on a large-scale Java GitHub corpus reveal the following: (a) Our formulation significantly outperforms all baselines including a comparable form of dynamic evaluation, even with significantly fewer adaptation steps in many cases; (b) Most of our performance benefits come from reducing the cross-entropy on identifiers and literals. One of the limitations of this work is that updating model parameters might become impractical in the context of large models as it might be computationally expensive or not even an option because of black-box access to the model parameters. Another limitation of this work is that it uses context only from the current file. We address both these limitations in our next article ([Shrivastava et al., 2022](#)) by incorporating relevant context from the entire repository directly into the input prompt of the LLM, without the need to adapt the parameters of the LLM.
- **Repository-Level Prompt Generation for Large Language Models of Code** (Chapter 8): In [Shrivastava et al. \(2022\)](#) we present Repository-Level Prompt Generator (RLPG), a framework that learns to automatically generate prompts conditioned on the example, without requiring access to the weights of the LLM. RLPG utilizes the structure of the repository as well as the context from other files in the repository using a set of easy-to-understand prompt proposals. On the task of single-line code-autocompletion, we show that an oracle constructed from our proposed prompt proposals gives up to 36% relative improvement over Codex. Further, we show that when we use our prompt proposal classifier to predict the best prompt proposal, we can achieve up to 17% relative improvement over Codex, as well as improve over other baselines. One of the limitations of this work is that it uses context from only one prompt proposal. We address this limitation in our next article ([Shrivastava et al., 2023](#)) where we combine multiple prompt proposals.
- **RepoFusion: Training Code Models to Understand Your Repository** (Chapter 10): In [Shrivastava et al. \(2023\)](#) we propose RepoFusion, a framework that allows training code models with multiple relevant contexts from the repository. We highlight the notable enhancements in performance attained through training smaller models with diverse repository contexts, surpassing the results of training larger models without such context. In addition, we create and release [Stack-Repo](#), a dataset of 200 Java repositories with permissive licenses and near-deduplicated files

that are augmented with three types of repository contexts. RepoFusion has a limitation in terms of computation scalability as it exhibits linear scaling with respect to the number of input repo contexts N , leading to slower inference times for larger values of N . One possible solution to address this issue is to leverage FiDO (de Jong et al., 2022), an optimization technique for Fusion-in-Decoder (Izacard & Grave, 2021) that enables faster inference.

11.2. Broad Applicability of Our Framework

While this thesis focuses on specific realizations of our problem formulation outlined in Section 1.1, it is important to note that the formulation itself is quite general and can be readily extended to other settings. In the following discussion, we explore key design considerations that play a vital role in ensuring the broad applicability of our problem formulation.

- **Size of Support Context:** Since limited context can be given as input to the prediction module *Predict*, determining the optimal number and length of each relevant context becomes important. Potential ways of addressing these questions can be to think of techniques to combine multiple relevant contexts (e.g. RepoFusion proposed in our fourth article), retrieval-augmented methods that work with an external memory (Wu et al., 2022; Borgeaud et al., 2021) and using an LLM with a large context length (e.g. Anthropic’s Claude model¹ offers 100k tokens context window). However, using more relevant contexts with larger sizes may come with increased inference costs. Striking a balance between context size and computational efficiency remains an ongoing challenge in this domain.
- **Capturing the Dependence between *Enhance* and *Predict*:** The two stages of context enhancement (*Enhance*) and prediction using the enhanced context (*Predict*) in our general framework closely influence one another. *Predict* should learn to effectively leverage the support context Z provided by *Enhance*. Similarly, *Enhance* should make use of the feedback signal coming from *Predict* to guide the selection of Z that is relevant to the task. The first article in our thesis incorporates both these components by iteratively generating each line of the global program, considering not only the per-example solutions and their execution states but also the execution state of the previously generated line of the global program. In the subsequent three articles, we primarily focus on the first component, but we can refine the selection of Z by conditioning the respective *Enhance* modules on the code predictions from the code completion module (*Predict*). This iterative retrieval of support context from the repository, based on the code predictions, has been explored

¹<https://www.anthropic.com/index/100k-context-windows>

in recent work such as [Zhang et al. \(2023\)](#). This work can be viewed as an extension of our third article to incorporate the second component.

Ideally, a joint learning approach for the *Enhance* and *Predict* modules would be desirable. However, in practice, this can be challenging due to various factors such as limited training data for end-to-end learning, difficulties in backpropagating gradients with discrete variables (programs), and the computational complexity of training large-scale code generation models (e.g. when *Predict* is an LLMs). In such scenarios, training the modules for *Enhance* and *Predict* independently offers more flexibility. *Predict* can leverage its pretraining on large amounts of data to possess general knowledge, while *Enhance* can incorporate task-specific nuances using methods that focus on incorporating domain-specific contextual cues. Additionally, *Enhance* can be implemented with smaller-sized models compared to *Predict*.

- **Performance-Latency Tradeoff:** As large-scale deep learning models of code are deployed in real-time applications, striking a balance between model performance and low latency is a challenging task, as it requires optimizing both the quality of predictions and the response time to ensure optimal user satisfaction. With regards to our framework, this means optimizing for resource allocation within *Enhance* and *Predict* during inference such that specific time and computational budgets are met.
- **Generality of the Support Context:** While there are benefits to approaches that model *Enhance* based on the specificity of the task at hand, exploring methods to automatically condition *Enhance* on a given task and generate relevant contextual cues offers an exciting direction. A method like this would eliminate the need for task-specific training of *Enhance* and provide a more flexible solution. One possible way of thinking about this idea is to consider an instruction-tuned LLM that serves as both *Enhance* and *Predict*. When prompted with instructions capturing the details of the task, the LLM acts as *Enhance* and generates relevant contextual cues, which are then fed back as input to the same LLM acting as *Predict* with instructions to generate predictions for the target. However, designing an approach like this that performs well across diverse tasks and is computationally efficient to train poses significant challenges.
- **Human-in-the-loop:** Recent research has explored various aspects of the interactions between code assistants and humans. These studies have highlighted certain challenges, such as the generated code being difficult to understand or debug, leading developers to spend considerable time editing and revising it ([Vaithilingam et al., 2022](#); [Mozannar et al., 2022](#); [Barke et al., 2023](#); [Bird et al., 2022](#)). Additionally, concerns have been raised about the security of the generated code, as it may pose potential risks ([Pearce et al., 2021](#)). Furthermore, excessive reliance on code assistants can lead to users overlooking errors in their code ([Al Madi, 2022](#)) or introducing

mistakes due to excessive confidence (Perry et al., 2022b). Our general framework, which includes separate stages for context enhancement and prediction, has the potential to address some of these challenges. By incorporating more human-interpretable contextual cues (e.g. prompt proposals in Chapter 8), users can have greater control over which cues are utilized during the prediction stage. One of the ways the feedback from the user can be used is to select relevant contextual cues and iteratively refine predictions that can help mitigate issues like API hallucination, leading to the generation of more secure and accurate code. We discuss this in more detail in Section 11.3.2.

11.3. Going Forward

Below we list a couple of potential future research directions that serve as interesting extensions to the ideas presented in the thesis.

11.3.1. Modeling the Code Ecosystem

Programming seldom happens in isolation. While writing code, software developers may refer to external sources such as other projects within the organization, API documentation, tutorials, or Q&A sites like StackOverflow. Software developers also rely on tools such as an IDE where they write the code, a compiler for running unit test cases, a static analyzer to fix errors, a version management software like GitHub and a linter for code formatting to name a few. In addition, during the software development process, the developer might interact with other people like peer software developers and architects, code reviewers, and people who submit pull requests to incorporate new features or to report a bug.

To develop an effective code assistant, it is important for the model to understand the complex programming workflow, which involves iterative changes to the program state and collaboration among multiple individuals and tools. One way of doing this is to derive relevant contextual cues from this entire code ecosystem. Examples of such contextual cues include the commit history of the developer, compiler execution traces corresponding to different stages of the program, details of errors faced by the developer before making a final commit, bug reports, code reviews, and their resolutions. Recently, Google DeepMind proposed a system called DIDACT² that trains code models to incorporate some of the aspects mentioned above.

11.3.2. Modeling User Interactions

The current evaluation of deep learning models for code often relies on metrics that may not directly reflect user preferences. However, the increasing adoption of code models in

²<https://ai.googleblog.com/2023/05/large-sequence-models-for-software.html>

consumer-facing products like GitHub Copilot and Bard, as well as the success of methods like RLHF in ChatGPT, highlight the need to incorporate user interactions into the design process of these models. For example, instead of using accuracy as a metric for code completion, the models can use metrics that are based on the actions of the users in response to the generated completion, such as the acceptance rate of generated completions, the extent of user edits to predicted completions, and the time taken for users to resume coding after viewing a completion. Directly asking users to rate the quality of predictions can be a valuable approach, particularly in cases where it is challenging to devise objective metrics to measure aspects like the generation of copyrighted or insecure code, or when the model hallucinates an API usage. User feedback can be integrated into both the *Enhance* stage for selecting relevant contextual cues and the *Predict* stage for refining prediction attributes such as type, length, and display rate.

According to the findings of [Barke et al. \(2023\)](#), users interacting with GitHub Copilot can be categorized into two modes: accelerated mode, where users already have a clear direction and use Copilot to speed up their progress, and exploration mode, where users are uncertain and rely on Copilot to explore different options. In accelerated mode, users tend to accept suggestions quickly, while in exploration mode, users take more time to consider suggestions. When the user is in the exploration phase, it would be beneficial to leverage diverse contextual cues and display multiple predictions, allowing users to make informed choices. On the other hand, in the accelerated phase it would make sense to continue with the current strategy of selecting contextual cues and displaying concise suggestions that do not break the user’s programming flow.

In [Bhat et al. \(2023\)](#), based on the insights of a user study on software tutorial authoring, we propose a framework to model the different stages of the interaction of users with an LLM. Through our analysis, we identify certain patterns in user behavior. One notable finding is that as users interact with the LLM, they update their beliefs or mental models about the capabilities of the model, which influences their future interactions. Leveraging the history of user interactions with the model, we can assign weights to accepted responses and iteratively adapt the *Enhance* and *Predict* stages based on this dynamic user mental model. We also found that users often struggle with trusting the correctness and validity of the generated content. One possible way to address this could be to rank contextual cues based on their correctness and relevance, while also providing the source of the cues to enhance user trust and allow for content verification. Furthermore, we observe that users have specific preferences for the writing style of tutorials. To accommodate this, *Enhance* can derive contextual cues from tutorials previously authored by the user, tutorials of a similar nature (e.g., related to a specific technology), or tutorials targeted at the same level of audience (e.g., basic vs. advanced). The ranked cues can then be combined using approaches like RepoFusion (Chapter 10) to generate predictions that align with the user’s requirements.

References

- Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K Lahiri, and Sriram K Rajamani. Guiding language models of code with global context using monitors. *arXiv preprint arXiv:2306.10763*, 2023.
- Naser Al Madi. How readable is model-generated code? examining readability and visual inspection of github copilot. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–5, 2022.
- Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. Santacoder: don’t reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023.
- Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code, 2018. URL <https://arxiv.org/abs/1812.06469>.
- Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 207–216. IEEE Press, 2013.
- Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 38–49, 2015.
- Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4): 81, 2018a.
- Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *6th International Conference on Learning Representations, ICLR, Conference Track Proceedings*, 2018b.
- Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models for any-code generation. *arXiv preprint arXiv:1910.00577*, 2019.
- Rajeev Alur, Pavol Černý, and Arjun Radhakrishna. Synthesis through unification. In *International Conference on Computer Aided Verification*, pp. 163–179. Springer, 2015.
- Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *TACAS*, 2017.

- Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.
- Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
- Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*, 2016.
- Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA): 1–29, 2020.
- Shraddha Barke, Michael B James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):85–111, 2023.
- Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022.
- Avinash Bhat, Disha Shrivastava, and Jin LC Guo. Approach intelligent writing assistants usability with seven stages of action. *arXiv preprint arXiv:2304.02822*, 2023.
- Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 343–355, 2016.
- Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: probabilistic model for code. In *International Conference on Machine Learning*, pp. 2933–2942, 2016.
- Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools. *Queue*, 20(6):35–57, 2022.
- Sid Black, Stella Biderman, Alex Andonian, Quentin Anthony, Preetham Gali, Leo Gao, Eric Hallahan, Josh Levy-Kramer, Connor Leahy, Lucas Nestler, Kip Parker, Jason Phang, Michael Pieler, Shivanshu Purohit, Tri Songz, Phil Wang, and Samuel Weinbach. GPT-NeoX: Large scale autoregressive language modeling in pytorch, 2021. URL <http://github.com/eleutherai/gpt-neox>.
- Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark,

- Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens. *CoRR*, abs/2112.04426, 2021.
- Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pp. 2206–2240. PMLR, 2022.
- Jonathan Bragg, Arman Cohan, Kyle Lo, and Iz Beltagy. FLEX: Unifying evaluation for few-shot NLP. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=_WnGcwXLYOE.
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020a. URL <https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf>.
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020b.
- Manfred Broy, Florian Deißeböck, and Markus Pizka. A holistic approach to software quality at work. In *Proc. 3rd world congress for software quality (3WCSQ)*, 2005.
- Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, 2018.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

- Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, 2014.
- Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.
- Hoa Khanh Dam, Truyen Tran, and Trang Pham. A deep language model for software code. *arXiv preprint arXiv:1608.02715*, 2016.
- Michiel de Jong, Yury Zemlyanskiy, Joshua Ainslie, Nicholas FitzGerald, Sumit K. Sanghai, Fei Sha, and William Cohen. Fido: Fusion-in-decoder optimized for stronger performance and faster inference. *ArXiv*, abs/2212.08153, 2022.
- Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural Program Meta-Induction. In I Guyon, U V Luxburg, S Bengio, H Wallach, R Fergus, S Vishwanathan, and R Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017a.
- Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML’17, pp. 990–998. JMLR.org, 2017b.
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Cocomic: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007*, 2022.
- Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019a.
- Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In *Advances in Neural Information Processing Systems*, pp. 9169–9178, 2019b.
- Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser,

- Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.
- Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.
- Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining across github. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pp. 254–265, 2016.
- Jaroslav Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. Autofolding for source code summarization. *IEEE Transactions on Software Engineering*, 43(12):1095–1109, 2017.
- Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A generative model for code infilling and synthesis, 2022.
- Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.
- Tianyu Gao, Adam Fisch, and Danqi Chen. Making pre-trained language models better few-shot learners. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 3816–3830, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.295. URL <https://aclanthology.org/2021.acl-long.295>.
- Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’11, pp. 317–330, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304900. doi: 10.1145/1926385.1926423.
- Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.
- Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. Synthesize, execute and debug: Learning to repair for neural program synthesis. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 17685–17695. Curran Associates, Inc., 2020.
- Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

- Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 763–773. ACM, 2017a.
- Vincent J. Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pp. 763–773, New York, NY, USA, 2017b. Association for Computing Machinery. ISBN 9781450351058. doi: 10.1145/3106237.3106290.
- Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International Conference on Learning Representations*, 2020.
- Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pp. 837–847. IEEE, 2012.
- Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- Gautier Izacard and Edouard Grave. Leveraging passage retrieval with generative models for open domain question answering. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pp. 874–880, Online, April 2021. Association for Computational Linguistics.
- Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. Discovering the syntax and strategies of natural language programming with generative language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, 2022.
- Karen Sparck Jones, Steve Walker, and Stephen E. Robertson. A probabilistic model of information retrieval: development and comparative experiments - part 1. *Inf. Process. Manag.*, 36(6):779–808, 2000. doi: 10.1016/S0306-4573(00)00015-7.
- John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.
- Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *International Conference on Learning Representations*, 2018.
- Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 12-18 July 2020*, Proceedings of Machine Learning Research. PMLR, 2020.

- Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.
- Rafael-Michael Karampatsis and Charles Sutton. Maybe deep neural networks are the best choice for modeling source code. *arXiv preprint arXiv:1903.05734*, 2019.
- Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code!= big vocabulary: Open-vocabulary models for source code. *arXiv preprint arXiv:2003.07914*, 2020.
- Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. In *International Conference on Learning Representations*, 2020.
- Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015a.
- Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun (eds.), *3rd International Conference on Learning Representations, ICLR Conference Track Proceedings*, 2015b.
- Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015c.
- Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*, 2022.
- Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*, 2022.
- Ben Krause, Emmanuel Kahembwe, Iain Murray, and Steve Renals. Dynamic evaluation of neural sequence models. In *Proceedings of the 35th International Conference on Machine Learning*, Proceedings of Machine Learning Research, pp. 2766–2775, 2018.
- Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.
- Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 2873–2882. PMLR, 10–15 Jul 2018.
- Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018,

- pp. 436–449, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356985. doi: 10.1145/3192366.3192410.
- Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 3045–3059, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. URL <https://aclanthology.org/2021.emnlp-main.243>.
- Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.
- Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention and pointer networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI*, pp. 4159–4165, 2018.
- Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.
- Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 4582–4597, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.353. URL <https://aclanthology.org/2021.acl-long.353>.
- Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode, 2022.
- Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for GPT-3? In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*. Association for Computational Linguistics, 2022.
- Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing, 2021a. URL <https://arxiv.org/abs/2107.13586>.

- Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. Gpt understands, too. *arXiv:2103.10385*, 2021b.
- Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL <https://openreview.net/forum?id=Bkg6RiCqY7>.
- Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722*, 2022.
- Chen Lyu, Ruyun Wang, Hongyu Zhang, Hanwen Zhang, and Songlin Hu. Embedding api dependency graph for neural code generation. *Empirical Softw. Engg.*, 26(4), 2021. ISSN 1382-3256. doi: 10.1007/s10664-021-09968-2.
- Chris Maddison and Daniel Tarlow. Structured generative models of natural source code. In *International Conference on Machine Learning*, pp. 649–657, 2014.
- Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.
- Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, pp. 1045–1048, 2010.
- Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. Reading between the lines: Modeling user behavior and costs in ai-assisted programming. *arXiv preprint arXiv:2210.14306*, 2022.
- Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 532–542. ACM, 2013.
- Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint*, 2023a.
- Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2023b.
- Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 4861–4870. PMLR, 09–15 Jun 2019.

- Augustus Odena and Charles Sutton. Learning to represent programs with property signatures. In *International Conference on Learning Representations*, 2020.
- Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. {BUSTLE}: Bottom-up program synthesis through learning-guided exploration. In *International Conference on Learning Representations*, 2021.
- Emilio Parisotto, Abdel rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis, 2016.
- Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pp. 2719–2734, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.232.
- Pardis Pashakhanloo, Aaditya Naik, Hanjun Dai, Petros Maniatis, and Mayur Naik. Learning to walk over relational graphs of source code. In *Deep Learning for Code Workshop*, 2022a.
- Pardis Pashakhanloo, Aaditya Naik, Yuepeng Wang, Hanjun Dai, Petros Maniatis, and Mayur Naik. Codetrek: Flexible modeling of code using an extensible relational representation. In *International Conference on Learning Representations*, 2022b.
- Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.
- Richard E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., USA, 2nd edition, 1994. ISBN 0471107026.
- Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. An empirical cybersecurity evaluation of github copilot’s code contributions. *ArXiv abs/2108.09293*, 2021.
- Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 609–620. IEEE, 2017.
- Hila Peleg and Nadia Polikarpova. Perfect is the enemy of good: Best-effort program synthesis. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. *ACM Sigplan Notices*, 49(6):408–418, 2014.

- Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with ai assistants? *arXiv preprint arXiv:2211.03622*, 2022a.
- Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with ai assistants? *arXiv preprint arXiv:2211.03622*, 2022b.
- Edoardo Maria Ponti, Rahul Aralikkatte, Disha Shrivastava, Siva Reddy, and Anders Søgaard. Minimax and neyman–Pearson meta-learning for outlier languages. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pp. 1245–1260, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-acl.106. URL <https://aclanthology.org/2021.findings-acl.106>.
- Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent code completion with bayesian networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):1–31, 2015.
- Guanghui Qin and Jason Eisner. Learning how to ask: Querying LMs with mixtures of soft prompts. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 5203–5212, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.410. URL <https://aclanthology.org/2021.naacl-main.410>.
- Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. *arXiv preprint arXiv:2212.04356*, 2022.
- Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 2022a.
- Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 2022b.
- Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 419–428, 2014.
- Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". *ACM SIGPLAN Notices*, 50(1):111–124, 2015.
- Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747, 2016.
- Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, et al. A generalist agent. *arXiv preprint arXiv:2205.06175*, 2022.
- Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm, 2021. URL <https://arxiv.org/abs/2102.07350>.

- Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022.
- Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily L Denton, Kamyar Ghasemipour, Raphael Gontijo Lopes, Burcu Karagol Ayan, Tim Salimans, et al. Photorealistic text-to-image diffusion models with deep language understanding. *Advances in Neural Information Processing Systems*, 35:36479–36494, 2022.
- Timo Schick and Hinrich Schütze. Exploiting cloze-questions for few-shot text classification and natural language inference. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pp. 255–269, Online, April 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.eacl-main.20. URL <https://aclanthology.org/2021.eacl-main.20>.
- Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pp. 464–468, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-2074.
- Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 803–817. IEEE, 2019.
- Kensen Shi, Jacob Steinhardt, and Percy Liang. Frangel: component-based synthesis with control structures. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.
- Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. Auto-Prompt: Eliciting knowledge from language models with automatically generated prompts. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2020.
- Disha Shrivastava, Eeshan Gunesh Dhekane, and Riashat Islam. Transfer learning by modeling a distribution over policies. *arXiv preprint arXiv:1906.03574*, 2019.
- Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. On-the-fly adaptation of source code models. In *NeurIPS 2020 Workshop on Computer-Assisted Programming*, 2020.
- Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Learning to combine per-example solutions for neural program synthesis. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, 2021. URL <https://openreview.net/forum?id=4PK-St2iVZn>.
- Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-level prompt generation for large language models of code. *arXiv preprint arXiv:2206.12839*, 2022.
- Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. Repofusion: Training code models to understand your repository. *arXiv preprint*

arXiv:2306.10998, 2023.

- David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.
- Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In John Paul Shen and Margaret Martonosi (eds.), *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pp. 404–415. ACM, 2006. doi: 10.1145/1168857.1168907.
- Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 2014.
- Jiao Sun, Q. Vera Liao, Michael Muller, Mayank Agarwal, Stephanie Houde, Kartik Talamadupula, and Justin D. Weisz. Investigating explainability of generative ai for code through scenario-based design. In *27th International Conference on Intelligent User Interfaces*, 2022.
- Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27*, pp. 3104–3112. 2014.
- Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. *arXiv preprint arXiv:2005.08025*, 2020.
- Alexey Svyatkovskoy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Franco, and Miltiadis Allamanis. Fast and memory-efficient neural code completion. *arXiv preprint arXiv:2004.13651*, 2020.
- Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 269–280. ACM, 2014.
- Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, pp. 1–7, 2022.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30*, pp. 5998–6008, 2017a.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS’17*, pp. 6000–6010, Red Hook, NY, USA, 2017b. Curran Associates Inc. ISBN 9781510860964.

- Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Lukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416, 2018.
- Oriol Vinyals, Charles Blundell, Timothy Lillicrap, koray kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 29*, pp. 3630–3638. 2016.
- Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. <https://github.com/kingoflolz/mesh-transformer-jax>, May 2021.
- Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, nov 2022. doi: 10.1145/3540250.3549113. URL <https://doi.org/10.1145%2F3540250.3549113>.
- Wenhan Wang, Sijie Shen, Ge Li, and Zhi Jin. Towards full-line code completion with neural language models, 2020.
- Yanlin Wang, Ensheng Shi, Lun Du, Xiaodi Yang, Yuxuan Hu, Shi Han, Hongyu Zhang, and Dongmei Zhang. Cocosum: Contextual code summarization with multi-relational graph neural network. *arXiv preprint arXiv:2107.01933*, 2021a.
- Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, Online and Punta Cana, Dominican Republic, November 2021b. Association for Computational Linguistics.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.
- Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pp. 334–345. IEEE Press, 2015.
- Yuhuai Wu, Markus Norman Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. In *International Conference on Learning Representations*, 2022. URL <https://openreview.net/forum?id=TrjbxzRcnf->.
- Wen Xiao, Iz Beltagy, Giuseppe Carenini, and Arman Cohan. PRIMERA: Pyramid-based masked sentence pre-training for multi-document summarization. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022.

- Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. A systematic evaluation of large language models of code. *arXiv preprint arXiv:2202.13169*, 2022a.
- Frank F. Xu, Junxian He, Graham Neubig, and Vincent Josua Hellendoorn. Capturing structural locality in non-parametric language models. In *International Conference on Learning Representations*, 2022b.
- Shafiq Joty Steven C.H. Hoi Yue Wang, Weishi Wang. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*, 2021.
- Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Wang Yongji, and Jian-Guang Lou. When language model meets private library. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pp. 277–288, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics.
- Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*, 2023.
- Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Raymond J. Mooney, Junyi Jessy Li, and Milos Gligoric. Learning to generate code comments from class hierarchies, 2021.
- Weixiong Zhang. Complete anytime beam search. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, AAAI '98/IAAI '98*, pp. 425–430, USA, 1998. American Association for Artificial Intelligence. ISBN 0262510987.
- Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. Docprompting: Generating code by retrieving the docs. In *International Conference on Learning Representations*, 2023a.
- Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*, 2023b.
- Amit Zohar and Lior Wolf. Automatic program synthesis of long programs with a learned garbage collector. In *Advances in Neural Information Processing Systems*, pp. 2094–2103, 2018.