Exploring Continual Learning for Code Generation Models

Prateek Yadav[1]*, **Qing Sun**[2] †, **Hantian Ding**[2], **Xiaopeng Li**[2], **Dejiao Zhang**[2], **Ming Tan**[2], **Xiaofei Ma**[2], **Parminder Bhatia**[2], **Ramesh Nallapati**[2], **Murali Krishna Ramanathan**[2], **Mohit Bansal**[1,3], **Bing Xiang**[2]
University of North Carolina, Chapel Hill[1], AWS AI Labs[2], Amazon Alexa AI[3]
{praty, mbansal}@cs.unc.edu
{qinsun, dhantian, xiaopel, dejiaoz, mingtan, xiaofeim, parmib, rnallapa, mkraman, mobansal, bxiang}@amazon.com

Abstract

Large-scale code generation models such as Codex and CodeT5 have achieved impressive performance. However, libraries are upgraded or deprecated very frequently and re-training large-scale language models is computationally expensive. Therefore, Continual Learning (CL) is an important aspect that remains underexplored in the code domain. In this paper, we introduce a benchmark called CodeTask-CL that covers a wide range of tasks, including code generation, translation, summarization, and refinement, with different input and output programming languages. Next, on our CodeTask-CL benchmark, we compare popular CL techniques from NLP and Vision domains. We find that effective methods like Prompt Pooling (PP) suffer from *catastrophic forgetting* due to the unstable training of the prompt selection mechanism caused by stark distribution shifts in coding tasks. We address this issue with our proposed method, *Prompt Pooling with Teacher Forcing* (PP-TF), that stabilizes training by enforcing constraints on the prompt selection mechanism and leads to a 21.54% improvement over Prompt Pooling. Along with the benchmark, we establish a training pipeline that can be used for CL on code models, which we believe can motivate further development of CL methods for code models. Our code is available at https://github.com/amazon-science/codetask-cl-pptf.

1 Introduction

Code generation models (Nijkamp et al., 2022b; Wang et al., 2021b; Le et al., 2022; Fried et al., 2022) can increase the productivity of programmers by reducing their cognitive load. These models require significant computation to train as they have billions of parameters trained on terabytes of data. Hence, they are trained once and are then used repeatedly for several downstream applications. However, as software development constantly evolves with new packages, languages, and techniques (Ivers and Ozkaya, 2020), it is expensive to retrain these models. Therefore, it is essential to continually improve these models to avoid errors, generate optimized code, and adapt to new domains and applications.

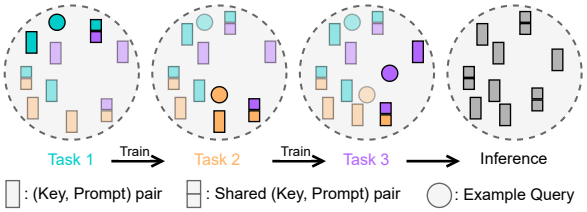


Figure 1: We show the process of prompt selection for Prompt Pooling with Teacher Forcing when learning multiple tasks sequentially. First, we initialize the prompt pool with (key, prompt) pairs (denoted by rectangles). Next, each (key, prompt) pair is assigned to either a single task or is shared by two tasks (denoted by colors). When learning Task 1 (green color), we obtain the query (green circle) for a given example and select the top-k (k=2 here) pairs from the assigned (key, prompt) pairs, highlighted in the figure. These selected pairs are then trained for the example. A similar process is followed for subsequent tasks. During inference, we remove task assignments and select the top-k pairs across all the pairs.

We explore continual learning (CL) (Ring, 1998; Thrun, 1998) abilities of code-generation models and aim to improve them. Specifically, we present a CodeTask-CL benchmark for code-based CL and aim to train a model on sequentially presented tasks with different data distributions without suffering from catastrophic forgetting (CF) (McCloskey and Cohen, 1989). This occurs when the model overfits the current task, resulting in a decline in performance on previously learned tasks.

Given the lack of CL benchmarks for the code domain, we create a benchmark called CodeTask-

*Work conducted during an internship at Amazon
†Corresponding author qinsun@amazon.com

CL using existing datasets. It consists of tasks like code completion (Iyer et al., 2018, 2019; Clement et al., 2020), code translation (Chen et al., 2018; Lachaux et al., 2020), code summarization (Wang et al., 2020a,b), and code refinement (Tufano et al., 2019). This benchmark presents a new and challenging scenario as it necessitates the adaptation of the model to varying input and output programming languages. Along with this benchmark, we also present a training framework to easily apply CL methods to code generation models.

Next, we evaluate the effectiveness of popular CL methods from NLP and Vision domains in the context of code generation models. We consider prompting methods (Wang et al., 2022b; Li and Liang, 2021a) and experience-replay (De Lange et al., 2019) due to their good performance for pre-trained models (Wu et al., 2022a). We also experiment with Prompt Pooling (PP) (Wang et al., 2022c), an effective prompting-based method for CL in the vision domain. Our results show that Prompt Pooling suffers from catastrophic forgetting on our proposed CODETASK-CL benchmark because of the complex distribution shift from varying input and output programming languages across tasks. With further investigation, we find that the unconstrained prompt selection mechanism leads to an unstable training problem. To address this, we propose our method *Prompt Pooling with Teacher Forcing* (PP-TF), which imposes constraints on prompt selection during training by assigning certain prompts to fixed tasks during training (see Figure 1). This results in stable training and better performance. Interestingly, we find when a replay buffer is available, the simple experience-replay (De Lange et al., 2019) method outperforms other CL methods and achieves performance similar to a multitask baseline (Crawshaw, 2020) where all tasks are provided at once.

In summary, our contributions include: (1) being the first study on CL for code generation tasks, (2) establishing a benchmark and a novel pipeline that supports CL for code generation to motivate future work, (3) identifying and addressing the unstable training issue of Prompt Pooling through our proposed method PP-TF, and (4) discussion on the best CL methods to use in different use cases.

2 Related Work

Code Generation Models. Code generation and language modeling for source code is an emerging research field experiencing active growth. Several model architectures have been examined recently, including encoder-only models (Feng et al., 2020; Guo et al., 2020), encoder-decoder models (Ahmad et al., 2021; Wang et al., 2021b), and decoder-only models (Nijkamp et al., 2022b; Chen et al., 2021; Nijkamp et al., 2022a). However, none of these models have been studied in the context of continual learning.

Continual Learning. There are various methods for Continual Learning (CL) and they fall into three categories: *Regularization*, *Replay*, and *parameter isolation* methods. **Regularization methods** (Kirkpatrick et al., 2017; Zenke et al., 2017; Schwarz et al., 2018) assign importance to model components and add regularization terms to the loss function. **Replay methods** (De Lange et al., 2019; Rebuffi et al., 2017; Lopez-Paz and Ranzato, 2017; Chaudhry et al., 2018) retain a small memory buffer of data samples and retrain them later to avoid catastrophic forgetting (CF). **Parameter isolation methods**, such as prompting-based methods (Wang et al., 2022b,a; Li and Liang, 2021a; Liu et al., 2021; Qin and Eisner, 2021), introduce or isolate network parameters for different tasks. For a more comprehensive overview of all CL methods, we refer the reader to Delange et al. (2021); Biesialska et al. (2020).

To the best of our knowledge, there are currently no studies or benchmarks for CL on code generation models. Therefore, we evaluate the effectiveness of prompting (Wang et al., 2022b; Li and Liang, 2021a) and experience replay (Chaudhry et al., 2018; Buzzega et al., 2020) based methods, which have demonstrated strong performance in CL on large pretrained models (Raffel et al., 2020). We do not consider regularization methods as they are not effective in continually learning large-scale pretrained models (Wu et al., 2022b). Next, we discuss our proposed benchmark and methods.

3 CODETASK-CL Benchmark

We present the CODETASK-CL benchmark to assess the CL abilities of code generation models. We also provide a novel training pipeline that can be used to continually train and evaluate code generation models. All of the datasets used to create the CODETASK-CL benchmark are available under the MIT license and more details on the dataset splits and input-output domains are in Table 2.

3.1 Coding Tasks

Code Generation aims to generate a code snippet from a natural language description. We use the CONCODE dataset (Iyer et al., 2018) which is a collection of tuples that consist of natural language descriptions, code environments, and code snippets, obtained from approximately 33,000 Java projects on GitHub. The objective of the study is to generate class member functions utilizing the natural language descriptions and class environment.

Code Summarization aims to generate a summary for a piece of code. We use the CodeSearchNet dataset (Husain et al., 2019), which consists of six programming languages (Python, Java, JavaScript, PHP, Ruby, and Go). The data for this task consists of the first paragraph of each documentation.

Code translation refers to the transformation of a program written in a particular programming language into another language while maintaining its functionality. We use the Java $\rightarrow$ C# dataset compiled by Lu et al. (2021) that provides pairs of code that perform the same tasks.

Code Refinement aims to improve the code by fixing bugs within the code automatically. We use the dataset provided by Tufano et al. (2019) consisting of pairs of faulty and corrected Java functions.

3.2 Evaluation

Next, we define the metrics used to evaluate a model continually on these datasets. We follow Lu et al. (2021) and evaluate each task using BLEU (Papineni et al., 2002). We follow (Chaudhry et al., 2018) to continually evaluate model's performance. We measure the *average BLEU* after learning all the tasks as, <BLEU> $= \frac{1}{N}\sum_{k=1}^{N} b_{N,k}$, where N is the total number of tasks and $b_{i,j}$ represents the BLEU score on task j after learning task i. Additionally, we report the average forgetting metric, denoted by <Forget>, to assess the model's ability to retain performance on previously learned tasks. This metric is calculated as the average difference between the maximum accuracy obtained for each task t and its final accuracy, given by <Forget> $= \frac{1}{N-1}\sum_{t=1}^{N-1}(\max_{k\in 1,\ldots,N-1} b_{k,t} - b_{N,t})$.

4 Prompt Pooling With Teacher Forcing

Prompt Pooling (Wang et al., 2022c) is a highly effective technique that possesses two key benefits. Firstly, the number of prompts required does not increase linearly with the number of tasks. Secondly, the prompts within the pool can be utilized across multiple tasks, thereby enabling the reuse of previously acquired knowledge. These abilities are advantageous in real-world scenarios, particularly when a model needs to be continually adjusted to accommodate a large number of users/tasks.

In Prompt Pooling (PP), a set of learnable prompts $P = \{P_i\}_{i=1}^{M}$ are defined and shared by multiple tasks. We follow Wang et al. (2022c) and utilize a query and key-matching process to select the prompts for each task. This process has four steps: (1) a learnable key, represented as $k_i \in \mathbb{R}^d$, is defined for each prompt, resulting in a prompt pool of the form $\{(k_i, P_i)\}_{i=1}^{M}$; (2) a query function $q(\boldsymbol{x})$ is defined, which takes an input $\boldsymbol{x}$ from a given task and produces a query vector $q_{\boldsymbol{x}} \in \mathbb{R}^d$; (3) the top-$k$ keys are selected based on the cosine similarity between the query $q_{\boldsymbol{x}}$ and all the key vectors $\{k_i\}_{i=1}^{M}$; (4) we obtain the final input vector $\boldsymbol{x}_p$ by pre-pending the example $\boldsymbol{x}$ with the prompts corresponding to the selected keys. Then $\boldsymbol{x}_p$ is fed into the pre-trained model f and we minimize the following loss function to *only* optimize the selected prompts and the corresponding keys while keeping the pre-trained model fixed.

$$\mathcal{L} = \mathcal{L}_{LM}(x_p, y) + \lambda \sum_{k_{s_i} \in K_s} sim(q(x), k_{s_i}) \quad (1)$$

where $\mathcal{L}_{LM}$ is the language modeling loss, $\boldsymbol{y}$ is the target sequence given the input $\boldsymbol{x}$, K_s is the set of selected keys from Step (3) above.

The query-key mechanism described above is an Expectation-Maximization (EM) (Moon, 1996) procedure. Given an example, we first select the top-k keys based on the cosine similarity (E-Step) and then train these selected keys to pull them closer to the query (M-Step). The training is stable when all tasks are jointly learned. However, in the CL context, tasks are sequentially trained which makes training unstable. Hence, we propose *Prompt Pooling with Teacher Forcing* (PP-TF) that removes the E-Step by assigning each $\{(k_i, P_i)\}$ pair to fixed tasks and only performs the M-Step of optimizing the keys. To encourage knowledge sharing, we allow a few $\{(k_i, P_i)\}$ pairs to be shared across tasks (see Figure 1). With these assignments/constraints in place, when training on task t, we use teacher forcing to select top-k prompts that are assigned to the task. Thus, for learning task t, our loss function becomes,

$$\mathcal{L} = \mathcal{L}_{LM}(x_p, y) + \lambda \sum_{k_{s_i} \in K_s \cap K_t} sim(q(x), k_{s_i}) \quad (2)$$

Method (↓)	Replay [5k]	Code Gen.	Code Trans.	Code Summ.	Code Ref.	<BLEU$_{Test}$>	<BLEU$_{Val}$>	<Forget$_{Val}$>
Sequential FT	✗	6.42	2.76	3.13	77.75	22.52	22.44	39.64
MTL	✗	32.24	74.87	14.69	79.23	50.26	49.25	-
Individual FT	✗	38.61	83.34	14.32	77.73	53.50	52.68	-
Shared Prompts	✗	0.63	6.75	0.37	78.5	21.56	21.71	30.33
Shared Prompts + ER	✓	13.82	45.87	14.36	78.64	38.17	36.93	8.46
Task Specific Prompts	✗	22.93	65.37	14.57	78.81	**45.42**	**44.56**	0.00
Prompt Pooling (PP)	✗	2.41	7.47	2.62	78.67	22.79	23.10	27.43
Prompt Pooling (PP) + ER	✓	16.33	50.96	13.13	78.71	39.78	38.47	6.41
PP + Teacher Forcing	✗	24.28	59.37	14.15	79.50	**44.33**	**43.10**	1.68
CodeT5 + ER	✓	32.92	77.94	11.74	78.43	**50.26**	**49.03**	2.22

Table 1: BLEU scores on the test set for the individual tasks and average BLEU (↑) and Forgetting (↓) metrics after sequentially learning Code Generation → Code Translation → Code summarization → Code Refinement Tasks.

where, K_t denotes the prompts assigned to task t for teacher forcing. As training progresses, the queries and keys learn to align in a stable manner, while also allowing for information sharing among tasks through the shared prompts. During inference, we discard the assignment for (key, prompt) pair and use cosine similarity to select the top-k pairs across the whole pool.

5 Experiments

We focus on the scenario of known task identities for continual learning. This is commonly the case in code-related domains and task identities can also be determined through input and output analysis in certain situations. In the field of NLP and Vision, methods utilizing experience replay and prompting have been highly effective for CL on large pre-trained models (Wang et al., 2022c, 2021a; Wu et al., 2022a). Moreover, regularization methods are shown to not work well in conjunction with pre-trained models (Wu et al., 2022a), and hence, we skip them from our study. Next, we present these methods along with some baseline methods.

5.1 Baselines

Sequential Finetuning (Yogatama et al., 2019) updates all model parameters for every incoming task in a sequential manner. This approach has been shown to suffer from catastrophic forgetting and serves as a lower bound for CL methods.

Individual Models (Howard and Ruder, 2018) finetune a separate models for each new task. This is considered an upper bound for CL methods.

Multitask Learning (Crawshaw, 2020) simultaneously learns multiple tasks at once, without experiencing distribution shift, resulting in a strong performance. For multitask learning, we prepend the task descriptors to the input and follow Wang et al. (2021b) to ensure balanced sampling across tasks with varying dataset sizes.

Shared Prompt Tuning (SP) defines M soft continuous prompts (Li and Liang, 2021b) which are added and fine-tuned for each example from all tasks. They are trained via gradient descent while keeping the pretrained model's parameters fixed.

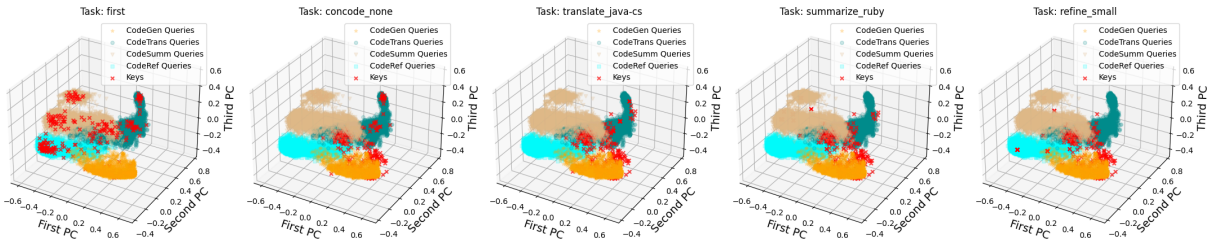
Task Specific Prompt Tuning (TSPT) defines a total of M soft continuous prompts (Li and Liang, 2021b) that are divided across N tasks, resulting in $\lfloor \frac{M}{N} \rfloor$ task-specific prompts.

Experience Replay (ER) (Riemer et al., 2019) involves maintaining a memory buffer B of examples from the previous task. The buffer randomly stores an equal number of samples from each past task and is used to retrain the model at later stages. Moreover, as several of the other methods outlined in this study can benefit from ER, we also include results with and without the utilization of ER.

5.2 Main Results

5.2.1 Task-CL Experiments

We use CodeT5 model (Wang et al., 2021b) as our pre-trained model when learning the CODETASK-CL benchmark. In Table 1, we report results for a single run on the methods described above and their ER variants. For more implementation details and hyperparameters used please refer to Appendix A.1. First, we find that the popular prompt pooling demonstrates catastrophic forgetting with a test BLEU score of 22.79%. Even when using ER with PP the performance is 39.78% which is still much worse than other methods. In contrast, *PP + TF* even without ER outperforms *PP* and *PP + ER* by 21.54% and 4.55% respectively. Moreover, our results show that the *CodeT5 + ER* method which finetunes the full CodeT5 model with ER performs the best with an average test BLEU score of 49.21%. Please refer to Appendix A.3 for experiments on the effect of buffer size on



(a) At Initialization (b) After CodeGen (c) After CodeTrans (d) After CodeSumm (e) After CodeRef

Figure 2: We plot the evolution of keys during the training process along with the fixed queries when sequentially learning, Code Generation → Code Translation → Code summarization → Code Refinement Tasks.

performance.

Discussion: We find that task-specific prompts are more effective than other prompting-based CL methods. However, due to their high storage requirements that scales linearly with the number of tasks, this approach is not feasible for large-scale applications where the model needs to be adapted for a large number of users or tasks. In contrast, a memory buffer might be available due to privacy concerns (Yoon et al., 2021) in many situations. In such cases, the *PP-TF* is the recommended method. Given these findings, we believe that the current Prompt Pooling based methods can be further improved in order to reuse knowledge across tasks.

5.2.2 Training Instability of Prompt Pooling

To show the root of catastrophic forgetting in prompt pooling, we evaluate how queries and keys align in the representation space after learning each task. To do so, we first select a subset of 5k training samples from four tasks resulting in 20k examples. We utilize a fixed codeT5 encoder as our query function that encodes provided examples to obtain queries. These queries remain unchanged during training and the keys are initialized using the data. We then use principal component analysis (PCA) (Pearson, 1901) on the queries and keys to obtain the first three principal components and plot them. After learning each task, we repeat the PCA step on the fixed queries and the updated prompt keys.

From Figure 2, we observe before the training starts, the keys (represented by red crosses) are evenly distributed among the queries of different tasks. However, after completing the training on the first task (CodeGen), most of the keys move toward the queries associated with that CodeGen (denoted by orange stars). This indicates that the prompts corresponding to these keys were primarily used for the CodeGen task and were trained by it. As a large portion of the prompts from the pool are utilized during the training of the CodeGen task, there are no key vectors available for allocation to the second task (CodeTrans). As a result, when learning the CodeTrans, some keys used for the previous task are pulled toward CodeTrans's queries and the corresponding prompts are updated. As each subsequent task is introduced, the key vectors are dynamically adjusted to align with the current task's queries, leading to a unstable process of matching in which updates to the key-prompt pairs are frequently in conflict with the previous tasks. Hence leading to catastrophic forgetting on the previous tasks.

6 Conclusion

In conclusion, we have introduced a novel benchmark, CODETASK-CL, tailored to cover a broad spectrum of tasks in the code domain, aiming to fuel advancements in Continual Learning (CL) for large-scale code generation models. Our study underscores the shortfalls of popular CL methods like Prompt Pooling when applied to coding tasks, predominantly due to catastrophic forgetting. However, we demonstrate that our proposed method, Prompt Pooling with Teacher Forcing (PP-TF), can effectively mitigate this issue, leading to a significant improvement of 21.54% over the baseline. Furthermore, we establish a comprehensive training pipeline catering to CL on code models. We believe that our contributions, both in the form of the CODETASK-CL benchmark and the PP-TF method, will ignite further exploration and innovation in CL techniques specifically designed for the dynamic and evolving realm of code generation.

Limitations

This work primarily focuses on evaluating the efficacy of existing continual learning (CL) meth-

ods for code generation models. It is important to note that many of these methods were specifically designed for natural language processing or computer vision domains and may not directly transfer to the code generation domain. Nevertheless, we have made efforts to identify and address any issues encountered during our analysis. It should be acknowledged, however, that the scope of our work is limited by the selection of methods and the benchmark used. While we have utilized the most popular CL methods from various categories, there may be methods that have not been included in this study due to their inefficacy in natural language processing or computer vision tasks but may be effective in code generation. As such, we encourage further research within the community to explore the potential of CL methods for code-generation models.

Acknowledgment

We thank Amazon for the *Amazon Post-Internship Fellowship* award that supported Prateek during this work. We also thank all the reviewers for their feedback on the paper.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Magdalena Biesialska, Katarzyna Biesialska, and Marta R. Costa-jussà. 2020. Continual lifelong learning in natural language processing: A survey. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 6523–6541, Barcelona, Spain (Online). International Committee on Computational Linguistics.

Pietro Buzzega, Matteo Boschini, Angelo Porrello, Davide Abati, and Simone Calderara. 2020. Dark experience for general continual learning: a strong, simple baseline. In *Advances in Neural Information Processing Systems*, volume 33, pages 15920–15930. Curran Associates, Inc.

Arslan Chaudhry, Marc'Aurelio Ranzato, Marcus Rohrbach, and Mohamed Elhoseiny. 2018. Efficient lifelong learning with a-gem. *arXiv preprint arXiv:1812.00420*.

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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Advances in neural information processing systems*, pages 2547–2557.

Colin B Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. Pymt5: multi-mode translation of natural language and python code with transformers. *arXiv preprint arXiv:2010.03150*.

Michael Crawshaw. 2020. Multi-task learning with deep neural networks: A survey. *ArXiv*, abs/2009.09796.

Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Ales Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. 2019. Continual learning: A comparative study on how to defy forgetting in classification tasks. *arXiv preprint arXiv:1909.08383*, 2(6).

M. Delange, R. Aljundi, M. Masana, S. Parisot, X. Jia, A. Leonardis, G. Slabaugh, and T. Tuytelaars. 2021. A continual learning survey: Defying forgetting in classification tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 328–339.

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

James Ivers and Ipek Ozkaya. 2020. Untangling the knot: Enabling rapid software evolution. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA.

Srinivasan Iyer, Alvin Cheung, and Luke Zettlemoyer. 2019. Learning programmatic idioms for scalable semantic parsing. *arXiv preprint arXiv:1904.09086*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven CH Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *arXiv preprint arXiv:2207.01780*.

Xiang Lisa Li and Percy Liang. 2021a. Prefix-tuning: Optimizing continuous prompts for generation.

Xiang Lisa Li and Percy Liang. 2021b. 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)*, pages 4582–4597, Online. Association for Computational Linguistics.

Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2021. Gpt understands, too. *arXiv:2103.10385*.

David Lopez-Paz and Marc'Aurelio Ranzato. 2017. Gradient episodic memory for continual learning. In *Advances in Neural Information Processing Systems*, pages 6467–6476.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Michael McCloskey and Neal J Cohen. 1989. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier.

T.K. Moon. 1996. The expectation-maximization algorithm. *IEEE Signal Processing Magazine*, 13(6):47–60.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Haiquan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022a. A conversational paradigm for program synthesis. *ArXiv*, abs/2203.13474.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022b. Codegen: An open large language model for code with multi-turn program synthesis. *ArXiv preprint, abs/2203.13474*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Karl Pearson. 1901. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572.

Guanghui Qin and Jason Eisner. 2021. 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*, pages 5203–5212, Online. Association for Computational Linguistics.

Colin Raffel, Noam M. Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *ArXiv*, abs/1910.10683.

Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. 2017. icarl: Incremental classifier and representation learning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 2001–2010.

Matthew Riemer, Ignacio Cases, Robert Ajemian, Miao Liu, Irina Rish, Yuhai Tu, , and Gerald Tesauro. 2019. Learning to learn without forgetting by maximizing transfer and minimizing interference. In *International Conference on Learning Representations*.

Mark B Ring. 1998. Child: A first step towards continual learning. In *Learning to learn*, pages 261–292. Springer.

Jonathan Schwarz, Jelena Luketina, Wojciech M Czarnecki, Agnieszka Grabska-Barwinska, Yee Whye Teh, Razvan Pascanu, and Raia Hadsell. 2018. Progress & compress: A scalable framework for continual learning. *arXiv preprint arXiv:1805.06370*.

Thomas Scialom, Tuhin Chakrabarty, and Smaranda Muresan. 2022. Continual-t0: Progressively instructing 50+ tasks to language models without forgetting. *arXiv preprint arXiv:2205.12393*.

Sebastian Thrun. 1998. Lifelong learning algorithms. In *Learning to learn*, pages 181–209. Springer.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29.

Chengyu Wang, Jianing Wang, Minghui Qiu, Jun Huang, and Ming Gao. 2021a. Transprompt: Towards an automatic transferable prompting framework for few-shot text classification. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 2792–2802.

Wenhua Wang, Yuqun Zhang, Zhengran Zeng, and Guandong Xu. 2020a. Transˆ 3: A transformer-based framework for unifying code summarization and code search. *arXiv preprint arXiv:2003.03238*.

Yanlin Wang, Ensheng Shi, Lun Du, Xiaodi Yang, Yuxuan Hu, Shi Han, Hongyu Zhang, and Dongmei Zhang. 2020b. Cocosum: Contextual code summarization with multi-relational graph neural network. *arXiv preprint arXiv:2107.01933*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021b. 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*, pages 8696–8708.

Zifeng Wang, Zizhao Zhang, Sayna Ebrahimi, Ruoxi Sun, Han Zhang, Chen-Yu Lee, Xiaoqi Ren, Guolong Su, Vincent Perot, Jennifer Dy, et al. 2022a. Dualprompt: Complementary prompting for rehearsal-free continual learning. *European Conference on Computer Vision*.

Zifeng Wang, Zizhao Zhang, Chen-Yu Lee, Han Zhang, Ruoxi Sun, Xiaoqi Ren, Guolong Su, Vincent Perot, Jennifer Dy, and Tomas Pfister. 2022b. Learning to prompt for continual learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 139–149.

Zifeng Wang, Zizhao Zhang, Chen-Yu Lee, Han Zhang, Ruoxi Sun, Xiaoqi Ren, Guolong Su, Vincent Perot, Jennifer Dy, and Tomas Pfister. 2022c. Learning to prompt for continual learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 139–149.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Tongtong Wu, Massimo Caccia, Zhuang Li, Yuan-Fang Li, Guilin Qi, and Gholamreza Haffari. 2022a. Pre-trained language model in continual learning: A comparative study. In *International Conference on Learning Representations*.

Tongtong Wu, Massimo Caccia, Zhuang Li, Yuan-Fang Li, Guilin Qi, and Gholamreza Haffari. 2022b. Pre-trained language model in continual learning: A comparative study. In *International Conference on Learning Representations*.

Dani Yogatama, Cyprien de Masson d'Autume, Jerome Connor, Tomas Kocisky, Mike Chrzanowski, Lingpeng Kong, Angeliki Lazaridou, Wang Ling, Lei Yu, Chris Dyer, et al. 2019. Learning and evaluating general linguistic intelligence. *arXiv preprint arXiv:1901.11373*.

Jaehong Yoon, Wonyong Jeong, Giwoong Lee, Eunho Yang, and Sung Ju Hwang. 2021. Federated continual learning with weighted inter-client transfer. In *International Conference on Machine Learning*, pages 12073–12086. PMLR.

Friedemann Zenke, Ben Poole, and Surya Ganguli. 2017. Continual learning through synaptic intelligence. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3987–3995. JMLR. org.

A Appendix

A.1 Implementation Details

In our experiments, we report the results of a single run. We used the *CodeT5-small* model (Wang et al., 2021b) with 60M parameters from Huggingface (Wolf et al., 2019), which is an encoder-decoder model pre-trained on CodeSearchNet (Husain et al., 2019). We use a separate and fixed codeT5 encoder model as the query function to encode the input examples for prompt pooling. For all prompting-related experiments, the CodeT5 model remains frozen and only the prompts are finetuned. In cases where we have ER with prompting methods, the ER is also applied while finetuning the prompts. Our prompt pool consisted of 500 prompts, with 100 prompts being selected to prepend to examples for each task. For the Shared Prompts method, we utilized 100 prompts that are used for all the tasks. For the Task-Specific Prompt method, we utilized different 100 prompts for each task. Unless otherwise specified, we used a buffer size of 5000 examples for all methods employing ER. The Adam (Kingma and Ba, 2014) optimizer was utilized, along with early stopping. The hyperparameters for our experiments were taken from Wang et al. (2021b), and the tasks from CODETASK-CL

Scenario	Task	Dataset Name	Input	Output	Train	Validation	Test
Task-CL	Generation	CONCODE	English	Java	100k	2k	2k
	Translation	CodeTrans	Java	C#	10k	0.5k	1k
	Sumarization	CodeSearchNet	Ruby	English	25k	1.4k	1.2k
	Refinement	BFP	Java	Java	46k	5.8k	5.8k

Table 2: Table providing the Dataset Statistics for the task used in CODETASK-CL benchmark. We specify the input and output domains along with the split sizes for train, validation, and test sets.

Method (↓)	Buffer Size	Code Gen.	Code Trans.	Code Summ.	Code Ref.	<$BLEU_{Test}$>	<$BLEU_{Val}$>	<$Forget_{Val}$>
CodeT5 + ER	100	24.11	61.87	10.72	77.82	43.63	41.25	14.18
	500	29.39	57.56	11.33	78.70	44.25	40.1	11.42
	1000	28.23	73.33	12.06	78.03	47.91	46.74	6.98
	2000	31.10	75.52	11.85	77.58	49.01	47.59	5.99
	5000	32.92	77.94	11.74	78.43	**50.26**	**49.03**	**2.22**
MTL	-	32.24	74.87	14.69	79.23	50.26	49.25	-
Individual FT	-	38.61	83.34	14.32	77.73	53.50	52.68	-

Table 3: Table showing performance on each task as we vary the Buffer Size when sequentially learning Code Generation → Code Translation → Code summarization → code Refinement Tasks.

benchmark were learned in random order specified in Table 1. The results of our experiments included the Average validation and test BLEU scores, as well as the forgetting metric on the validation set. The implemntation of BLEU was taken from the CodeT5 paper (Wang et al., 2021b). We ran experiments on a single A6000 GPU with 48 GB of memory with total computation of 14 GPU days.

A.2 Data Statistics for CODETASK-CL Benchmark

Table 2 shows the train, validation, and test data sizes for all the tasks used in the CODETASK-CL benchmark. We also present the input and output domains for each of the individual tasks. Given the input and output domains for these tasks are starkly different this makes this benchmark challenging as the distribution shift is large. Please refer to Section 3 in the main paper for more details about the benchmark. All of the datasets used to create the CODETASK-CL benchmark are available under the MIT license.

A.3 Impact of Buffer Size on ER Performance.

If ER replay is possible, we find that *CodeT5 + ER* is the most performant method. We go on to further assess the impact of buffer size on the performance. In Table 3, we present the aggregated results for a total buffer size of 100, 500, 1000, 2000, and 5000. Our findings suggest that the is an increase in performance as the buffer size increases. We observe that CodeT5 + ER with a small buffer size of 100 examples outperforms PP + ER (5k examples) by 3.85% respectively. Moreover, CodeT5 + ER with a buffer size of 1000 outperforms the best method without ER. Our findings are in line with that of Scialom et al. (2022) and demonstrate that whenever possible, we should use ER with pretrained models. Although in cases with no buffer with a large number of tasks, *PP + TF* is the best method to use.