CoTran: An LLM-based Code Translator using Reinforcement Learning with Feedback from Compiler and Symbolic Execution

Paper #2573 (Supplementary material)

A Appendix

A.1 Reproducibility

Our code and the AVATAR-TC dataset are publicly available at https://anonymous.4open.science/r/CoTran and included in the supplementary .zip file. The repository also includes the generated P2J and J2P translations obtained through the SoTA methods and CoTran variants (tabulated in Table A1). Additionally, we have made a significant effort to make it easy for the user to run the code. The root folder in the repository contains a README file that details the folder structure, library dependencies, and steps for running the code.

A.2 CoTran Implementation and Design Choices

Here, we detail some of the additional design choices we adopted during implementing CoTran. For the symbolic execution feedback (SF) implementation, Symflower generates JUnit tests for each method (named p) of the input Java code $\mathbf{s}$. These are tested on that method named p^* of $\widehat{\mathbf{s}}$, for which $p^* = \text{argmax}_{p' \in \widehat{\mathbf{s}}}\left(\text{JaccardSimilarity}(p, p')\right)$. This accounts for the fact that method names in $\mathbf{s}$ and $\widehat{\mathbf{s}}$ may not be the same yet have similar input-output (IO) behavior and thus can be considered equivalent. Symflower could correctly generate JUnit tests for $5{,}738$ Java codes in AVATAR-TC (Train) with an average of 6.34 tests per code. The training pipeline is implemented using Pytorch [10] and trained on a single compute node comprising four NVIDIA V100 GPUs with 32GB memory and six CPU cores/GPU. The low-rank adaptation matrices of the query/value layers have a rank r of 16 and a scaling factor α of 32. For optimization of LLM by CE loss, we use Adam [5] optimizer with a learning rate (lr) of 10^{-4}. For reinforcement learning (RL)-based optimization of LLMs by PPO algorithm, the output sequence is generated by pure sampling from the LLM distribution, with lr of $1.41 * 10^{-5}$. Further, the parameter-efficient approach of fine-tuning in Algorithm 1 (in the main paper) brings down the total training time by $\sim 75\%$ (keeping number of GPUs constant) as compared to jointly two LLMs without low-rank optimization.

Regarding the need for curation of a benchmark suite, most code translation datasets contain pairs of equivalent snippets/functions but not whole-programs, in two different languages. Further, the languages typically chosen are not very disparate e.g. Java and C#, two statically-typed languages with similar syntax. The AVATAR-TC dataset proposed in the paper has pairs of whole-programs in Java and Python (a statically- and dynamically-typed language, with different syntactic styles). And to the best of our knowledge, AVATAR-TC is the first such large-scale dataset where code compilability (syntactical correctness) is ensured, and code pairs have undergone thorough testing w.r.t. human-written TCs.

A.3 Non-RL Schemes of Incorporating CF and SF into Training

Note that, in the main paper we have proposed an RL-based scheme to incorporate compiler feedback (CF) and symbolic execution feedback (SF) during the LLM training process. However, we have also considered using non-RL schemes to incorporate this feedback during training. Instead of using CF and SF as rewards in an RL-based optimization framework as described in Algorithm 1 (in the main paper), another approach to incorporating feedback during LLM fine-tuning is to formulate loss as a combination of the CE loss and the feedback (CF, SF), and thereafter train the LLM by Supervised Fine-Tuning (SFT). The two alternatives for such an approach are:

a. **CoTran$^+$ (additive approach):** Minimize a linearly-weighted combination of CE loss and both the feedback.
b. **CoTran$^\times$ (multiplicative approach):** Minimize a weighted CE loss, where weights for samples in a mini-batch are a combination of both the feedback.

For CoTran$^+$, considering $\alpha_c, \alpha_s \in [0, 1]$ as the hyperparameters, the combined loss terms for LLM_f and LLM_b in a back-to-back (b2b) translation pipeline are thus respectively defined as:

$$\mathcal{L}_f^{\theta_f}\left(\mathbf{s}, \widehat{\mathbf{s}}, \mathbf{t}, \widehat{\mathbf{t}}\right) = \left[\mathcal{L}_{CE}^{\theta_f}\left(\mathbf{t}, \widehat{\mathbf{t}}\right) \quad \left[\omega_{CF}(\mathbf{t}, \widehat{\mathbf{t}}) \quad \omega_{SF}(\mathbf{s}, \widehat{\mathbf{s}})\right] \cdot \begin{bmatrix} 1-\alpha_s \\ \alpha_s \end{bmatrix}\right] \cdot \begin{bmatrix} 1-\alpha_c \\ \alpha_c \end{bmatrix}$$
$$\mathcal{L}_b^{\theta_b}\left(\mathbf{s}, \widehat{\mathbf{s}}\right) = \left[\mathcal{L}_{CE}^{\theta_b}\left(\mathbf{s}, \widehat{\mathbf{s}}\right) \quad \left[\omega_{CF}(\mathbf{s}, \widehat{\mathbf{s}}) \quad \omega_{SF}(\mathbf{s}, \widehat{\mathbf{s}})\right] \cdot \begin{bmatrix} 1-\alpha_s \\ \alpha_s \end{bmatrix}\right] \cdot \begin{bmatrix} 1-\alpha_c \\ \alpha_c \end{bmatrix} \tag{1}$$

For CoTran$^\times$, we train by weighing the CE loss with the reciprocal product of CF and SF. The respective combined loss for the backward and forward models is thus defined as:

$$\mathcal{L}_f^{\theta_f}\left(\mathbf{s}, \widehat{\mathbf{s}}, \mathbf{t}, \widehat{\mathbf{t}}\right) = \mathcal{L}_{CE}^{\theta_f}\left(\mathbf{t}, \widehat{\mathbf{t}}\right) / \left(\omega_{CF}(\mathbf{t}, \widehat{\mathbf{t}}) \times \omega_{SF}(\mathbf{s}, \widehat{\mathbf{s}})\right)$$
$$\mathcal{L}_b^{\theta_b}\left(\mathbf{s}, \widehat{\mathbf{s}}\right) = \mathcal{L}_{CE}^{\theta_b}\left(\mathbf{s}, \widehat{\mathbf{s}}\right) / \left(\omega_{CF}(\mathbf{s}, \widehat{\mathbf{s}}) \times \omega_{SF}(\mathbf{s}, \widehat{\mathbf{s}})\right) \tag{2}$$

which are normalized by the sum of weights per mini-batch.

The issue here is that CF and SF are both obtained by non-differentiable operations. As such, in CoTran$^+$ the feedback when added to the CE loss, do not play a role in the optimization process. CoTran$^\times$ on the other hand, is more theoretically grounded where a weighted CE loss is optimized. In Table A1, along with all the previously-reported results of Table 1 (in the main paper), we tabulate the performance of CoTran$^+$ and CoTran$^\times$ when using only CF

Table A1: **Code Translation Results (Full):** Performance Comparison of CoTran for Java-Python (J2P) and Python-Java (P2J) translation. (In each column, the highest value is marked in **bold**, second-highest underlined.)

Method / Tool	Model	Java → Python (J2P)						Python → Java (P2J)					
		FEqAcc	*CompAcc*	$\text{errPos}_{1\text{st}}$	CodeBLEU	BLEU	EM	*FEqAcc*	*CompAcc*	$\text{errPos}_{1\text{st}}$	CodeBLEU	BLEU	EM
Transpilers	java2python [8]	3.32	41.46	28.62	20.31	17.54	0	-	-	-	-	-	-
	TSS CodeConv [14]	0.46	58.30	54.26	41.87	24.44	0	-	-	-	-	-	-
	py2java [3]	-	-	-	-	-	-	0	0	1.61	41.56	48.59	0
Recent competing tools *(unsupervised training)*	TransCoder [11]	0.46	88.09	63.57	35.07	32.07	0	0	0	4.57	35.02	35.06	0
	TransCoder-DOBF [6]	0.46	63.00	47.10	39.98	33.84	0	0	0	3.11	33.33	32.72	0
	TransCoder-ST [12]	0.46	91.58	74.68	40.04	37.30	0	0	0	4.67	29.88	28.15	0
ChatGPT	GPT-3.5-turbo [9]	**76.06**	95.36	90.88	52.11	53.19	0.29	21.65	24.97	30.86	54.08	55.58	0
Recent competing tools *(supervised training on AVATAR-TC)*	CodeBERT [2]	12.31	84.77	79.57	46.00	48.10	0.46	0.74	**96.79**	**99.51**	26.10	19.62	0
	GraphCodeBERT [4]	10.88	85.05	79.78	45.53	47.26	0.57	0.46	89.75	98.05	23.72	16.21	0
	CodeGPT [7]	24.86	78.92	89.21	38.38	38.64	1.49	13.40	45.13	94.50	40.51	37.96	0.52
	CodeGPT-adapted [7]	24.17	76.75	89.31	36.84	37.36	1.55	20.50	52.00	97.60	41.46	38.15	1.03
	PLBART-base [1]	38.55	91.47	90.79	54.77	59.34	1.32	38.26	75.77	96.64	55.96	59.24	0.97
	CodeT5-base [15]	40.95	92.84	**93.76**	55.34	60.03	2.41	33.79	68.84	98.02	57.64	60.16	0.86
	PPOCoder [13]	44.27	93.47	91.44	55.16	59.51	1.89	37.11	59.62	96.77	55.04	58.52	0.52
Our tool	**CoTran (baseline)**	44.52	96.12	92.07	55.44	58.71	2.11	40.41	73.63	92.16	**59.11**	61.12	1.66
Our tool with compiler feedback only	CoTran$^+$ ($\alpha_c = 0.25$)	46.06	97.08	93.26	55.18	58.48	**2.68**	39.95	74.89	93.94	58.85	60.64	**1.71**
	CoTran$^+$ ($\alpha_c = 0.50$)	44.52	96.35	91.95	55.86	58.27	2.17	40.64	73.57	91.59	58.58	60.42	1.37
	CoTran$^+$ ($\alpha_c = 0.75$)	46.33	96.62	92.89	55.46	59.68	2.64	40.01	74.26	90.66	57.80	59.48	0.91
	CoTran$^\times$	44.62	96.91	93.39	55.22	58.50	2.35	38.03	71.48	90.69	54.93	59.82	1.60
	CoTran + CF (RL-based training)	47.02	96.56	91.58	56.10	60.59	2.23	42.78	74.80	96.91	58.55	61.26	1.60
	CoTran + CF (RL+SFT interleaved training)	49.83	96.79	92.08	56.07	60.61	2.23	45.93	75.77	96.89	58.28	61.21	1.60
Our tool (b2b) with compiler and symexec feedback	CoTran$^+$ ($\alpha_c = 0.5$, $\alpha_s = 0.01$)	45.68	96.58	92.11	55.72	58.92	2.17	40.92	75.27	94.89	58.97	60.34	1.08
	CoTran$^+$ ($\alpha_c = 0.5$, $\alpha_s = 0.5$)	46.14	96.29	92.02	55.95	58.22	2.23	41.84	74.89	93.94	58.89	60.10	1.31
	CoTran$^\times$	46.45	96.56	91.58	56.08	60.58	2.23	41.63	74.34	96.91	58.53	61.27	1.60
	CoTran + CF + SF (RL-based training)	50.45	96.79	92.15	56.17	60.60	2.23	43.92	75.14	96.93	58.59	**61.28**	1.60
	CoTran + CF + SF (RL+SFT interleaved training)	53.89	**97.14**	92.73	**56.24**	**60.69**	2.29	**48.68**	76.98	96.93	58.38	61.19	1.60

and when using both CF, SF. For training with CF as the sole feedback, only the forward LLM is trained (there is no SF and no need for b2b LLMs). As such, for CoTran$^+$, the loss of LLM_f in Eqn. 1 is computed as $\mathcal{L}_f^{\theta_f}\left(\mathbf{t},\widehat{\mathbf{t}}\right) = \left[\mathcal{L}_{CE}^{\theta_f}\left(\mathbf{t},\widehat{\mathbf{t}}\right) \quad \omega_{CF}(\mathbf{t},\widehat{\mathbf{t}})\right] \cdot \begin{bmatrix} 1-\alpha_c \\ \alpha_c \end{bmatrix}$. And for CoTran$^\times$, the loss of LLM_f in Eqn. 2 is computed as $\mathcal{L}_f^{\theta_f}\left(\mathbf{t},\widehat{\mathbf{t}}\right) = \mathcal{L}_{CE}^{\theta_f}\left(\mathbf{t},\widehat{\mathbf{t}}\right)/\omega_{CF}(\mathbf{t},\widehat{\mathbf{t}})$, normalized by the sum of weights per mini-batch.

However, it is observed that the LLMs are not able to learn from the feedback well, from either the additive or the multiplicative approach. That is, the translation performance is minimally better than the baseline and, does not compare well with the RL-based schemes. Also, CoTran$^+$ is sensitive to hyperparameters α_c, α_s.

A.4 Results: Surprising Finding w.r.t. Transpilers

In Table A1, we expected the human-written transpilers to outperform LLM-based methods. However, they do not perform well on any of the listed metrics. The `java2python` [8] transpiler was last updated 7 years back, and as such does not support the latest versions of Java and Python3. In fact, any hand-crafted set of rules faces this issue, it needs to be updated manually after every major version release of the PLs. The commercial transpiler TSS CodeConv [14] performs better but fails for scenarios where the specific conversion rules are not defined. It copies verbatim portions from the source-language code to the generated translation. Thus, the transpilers are excellent choices for a translation head-start from where a human developer can take over, but are not suitable for translating whole-programs and producing readily-compilable solutions.

References

[1] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. Unified Pre-training for Program Understanding and Generation. In *Proc. 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online, June 2021. ACL. doi: 10.18653/v1/2021.naacl-main.211. URL https://aclanthology.org/2021.naacl-main.211.

[2] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, Nov. 2020. ACL. doi: 10.18653/v1/2020.findings-emnlp.139.

[3] N. Fomin. `py2java`: Python to Java Language Translator, 2019. https://pypi.org/project/py2java/.

[4] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, et al. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations (ICLR)*, 2021.

[5] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations (ICLR)*, 2015.

[6] M.-A. Lachaux, B. Roziere, M. Szafraniec, and G. Lample. DOBF: A Deobfuscation Pre-training Objective for Programming Languages. *Advances in Neural Information Processing Systems (NeurIPS)*, 34: 14967–14979, 2021.

[7] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, et al. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv preprint arXiv:2102.04664*, 2021.

[8] T. Melhase, B. Kearns, L. Li, I. Curt, and S. Saladi. `java2python`: Simple but Effective Tool to Translate Java Source Code into Python, 2016. https://github.com/natural/java2python.

[9] OpenAI. ChatGPT [Large Language Model], 2023. https://chat.openai.com.

[10] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems*, 32, 2019.

[11] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample. Unsupervised Translation of Programming Languages. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:20601–20611, 2020.

[12] B. Roziere, J. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample. TransCoder-ST: Leveraging Automated Unit Tests for Unsupervised Code Translation. In *International Conference on Learning Representations (ICLR)*, 2022. URL https://openreview.net/forum?id=cmt-6KtR4c4.

[13] P. Shojaee, A. Jain, S. Tipirneni, and C. K. Reddy. Execution-based Code Generation using Deep Reinforcement Learning. *arXiv preprint arXiv:2301.13816*, 2023.

[14] TSS. The Most Accurate and Reliable Source Code Converters, 2023. (Tangible Software Solutions): https://www.tangiblesoftwaresolutions.com/.

[15] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proc. 2021 Conference on Empirical Methods in NLP*, pages 8696–8708. ACL, 2021. doi: 10.18653/v1/2021.emnlp-main.685. URL https://aclanthology.org/2021.emnlp-main.685.