Program-guided Transformer for Program-guided Tasks

Anonymous Author(s)
Affiliation
Address
email

Abstract

Since programs are structured in a formal language, they can play an important role in the communication between humans and agents. Therefore, learning to execute programs is an essential task for developing a program-guided agent. We notice that programs can provide guidance in both semantics and structures, and a good model should unify both signals. In this paper, we propose Program-guided Transformer (ProT), which uses attention mechanisms to leverage semantic guidance and leverage soft masks to follow structural guidance. ProT can serve as a general module for program-guided tasks such as visual reasoning and policy learning, and it establishes a new state-of-the-art on GQA and 2D Minecraft.

1 Introduction

Our contributions are summarized as follows:

1. We formulate and highlight program-guided tasks, which require exploiting the given program under the task specification.
2. We propose Program-guided Transformers (ProT) for program-guided tasks.
3. We conduct comprehensive experiments to demonstrate the power of ProT on several downstream tasks.

2 Related Work

Program Synthesis, Induction, and Interpretation Program synthesis methods aim at predicting programs from task specifications. In the domain of string transformations and digit operation, one line of work treat input/output pairs as task specifications [10, 3, 7, 6, 13]. In the vision domain, another line of work proposes to synthesis programs to represent compositional visual structures [12, 23]. Program induction refers to learning an underlying input-output mapping from given examples [16, 9, 15]. No explicit programs are used or predicted in program induction. Neural Program Interpreter and its following works [18, 25, 19] try to learn to compose lower-level programs to express higher-level programs. Different from the above literature, our paper studies learning to execute programs on given specifications. This task is first studied in [20] in the policy learning domain.

Visual Reasoning Visual standing requires joint modeling

Attention and Transformer

Program-guided Agent

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

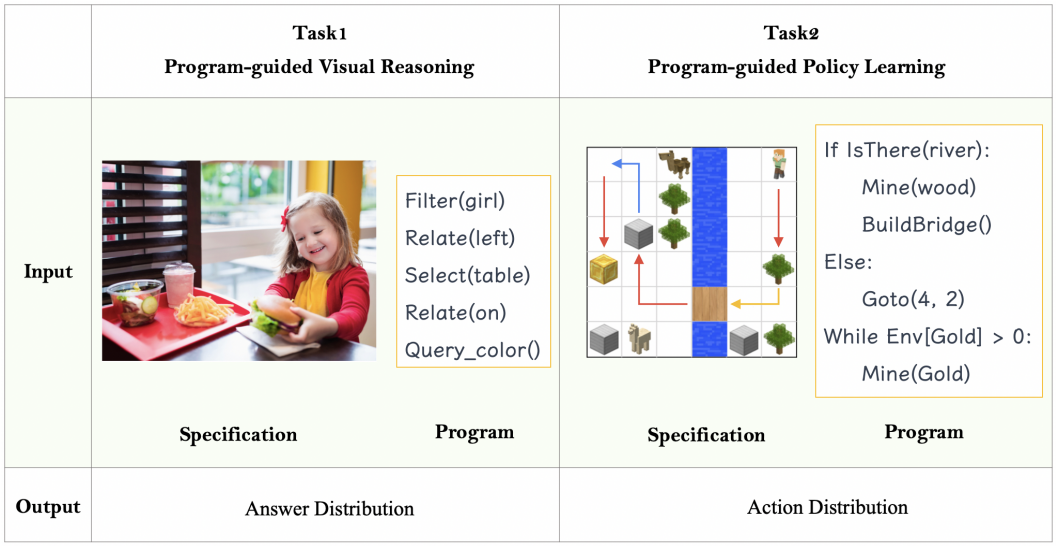

Figure 1: Illustration of two exemplar program-guided tasks. The first task is program-guided visual reasoning [14], where the model needs to learn to execute a visual program on the specification (image) to get a predicted answer. The second task is program-guided policy learning [20], where the agent needs to execute a program on the observed specification to predict an action at each timestep which should correspond to the program guidance.

3 Program-guided Tasks

Unlike natural language that is flexible, complex, and noisy, programs are structured, clean and formal [20]. Therefore, programs can serve as a powerful intermediate representation for human-computer interactions [11]. A lot of previous work studies *learning to infer (synthesis)* programs from data [17, 26, 23, 12]. However, we argue that it is important to study *learning to execute given* programs [20] based on three reasons. First, we found that in some tasks, the program inference problems are solved, and learning to execute programs conditioned on specifications becomes a bottleneck (Refer to Sec 5.1). Second, instructing machines via explicit programs instead of noisy natural languages enjoys efficiency and accuracy [20]. Third, we wish to propose unified models for different tasks that require learning to execute.

Formally, we denote Γ^α to be the space formed by a domain-specific language (DSL) for a task α. The design of domain-specific languages in our paper follows previous literature [14, 20]. We call a task α is a program-guided task if any instance $i \in \alpha$ consists of a tuple (S^i, P^i) where S^i is a specification and $P^i \in \Gamma^\alpha$ is a program. For convenience, we define a routine to be a minimum execution unit of a program (e.g. *Select(bag)* in Fig 2 is a routine). A specification S^i can be executed on a program P^i through an task-specific executor $\mathcal{E}^\alpha_\Phi$ to get outputs $\{O^i_q\}^Q_{q=1} = \mathcal{E}^\alpha_\Phi(S^i, P^i)$ where Q is the total number of routines in the program P^i. Note that different from hard-coded non-parametric executors used in some previous work [26, 10], the executor for a program-guided task contains learnable parameters Φ. The goal of a program-guided task is to maximizes a task-specific goal function $\mathcal{G}^\alpha$.

In Sec 4, we first present a uniform model, Program-guided Transformer, for program-guided tasks. Then we discuss the applications on two exemplar tasks in Sec 5 (refer to Fig 1).

4 Program-guided Transformer

A good model for a program-guided task should fulfill the following merits. First of all, it should leverage both the semantics and structures provided by the program. Second, it should parallel parts of the program's execution, as long as there is no results dependency [1]. Third, the model would better not leverage external memory for the efficiency of batched training. To this end, we present the Program-guided Transformer (ProT) shown in Fig2. ProT executes the given program on observed specifications and outputs latent result embeddings which could be decoded to explicit

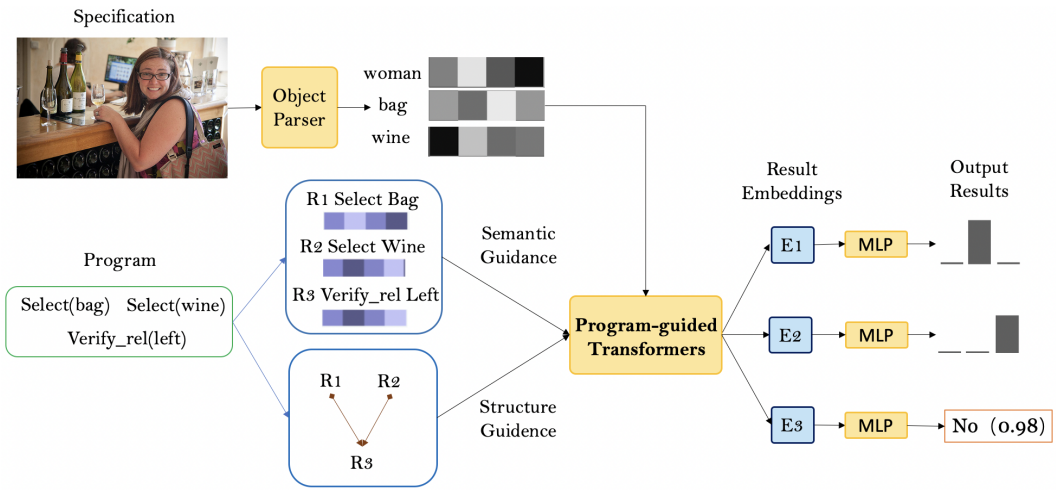

Figure 2: Program-guided Transformer used in program-guided tasks. Our Program-guided Transformer executes the given program on the input object-centric specification, and output latent results. The latent results are decoded via MLPs to get explicit output results.

results via MLPs. We detail two main components of ProT, the disentangled program representation, and transformer executors, in the following subsections.

4.1 Disentangled Program Representation

We adopt control flow analysis [1, 7] to formalize the execution of a program. A program can be represented as a directed connected acyclic graph $P = (R, E)$. The nodes R represent all routines in the program, and the edges E represent the result flow relations. An edge $e = (r_i, r_j) \in E$ exists if the input of r_j depends on the output of the routine r_i. In order to enable parallel computing, we layer the graph based on topological sorting [8] such that edges only exist between consecutive layers (refer to supplementary for a detailed algorithm). We denote $\widetilde{P} = (\{R_\tau\}_{\tau=0}^{T}, \{E_\tau\}_{\tau=0}^{T-1})$ as the layered program where R_τ is the set of routines in τth layer and E_τ is the set of edges that connect the τth layer and the $\tau + 1$th layer. The program is executed in a layer-by-layer manner, and all the nodes in one layer are executed in parallel via the transformer architecture. The total execution timestep is T, which can be smaller than the size of R due to parallel computing.

As shown in the left of Fig 3, we leverage both the semantics and structure of the program in our Program-guided Transformer. The semantic part of the program $P^{se} \in \mathbb{R}^{\|R\| \times D}$ is an semantic embedding matrix for all routines where $\|R\|$ is the size of R and D is the feature dimension (P^{se} is not changed for each timestep). In order to get P^{se}, we encode each routine's tokens via a learnable dictionary and concatenate each routine's token embeddings together (padded to D when needed). The structure part of the program $P^{st} \in \mathbb{R}^{T \times R \times R}$ is a tensor composed of T transition matrices which indicates how the execution results flow between layers. $P^{st}[\tau]$ denotes the transition matrix for the τ-th timestep, which is given by the following equation,

$$P^{st}[\tau][i][j] = \begin{cases} 1/\|E_\tau\| & \text{if } (e_j, e_i) \in E_\tau \text{ and } i \neq j \\ 1 & \text{if } \forall k, (e_k, e_i) \notin E_\tau \text{ and } i = j \\ 0 & \text{else,} \end{cases} \tag{1}$$

where $||E_\tau||$ is the size of E_τ.

4.2 Transformer Executors

Given the semantic guidance P^{se} and structure guidance P^{st}, we update the result embedding matrix $Z \in \mathbb{R}^{\|R\| \times D}$ for T timesteps via the transformer executors. The result embedding matrix is initialized as a zero matrix. To enable layer-by-layer execution, we only update part of the result embedding matrix at each timestep. Formally, denote $Z[R_\tau]$ to be the result embeddings selected

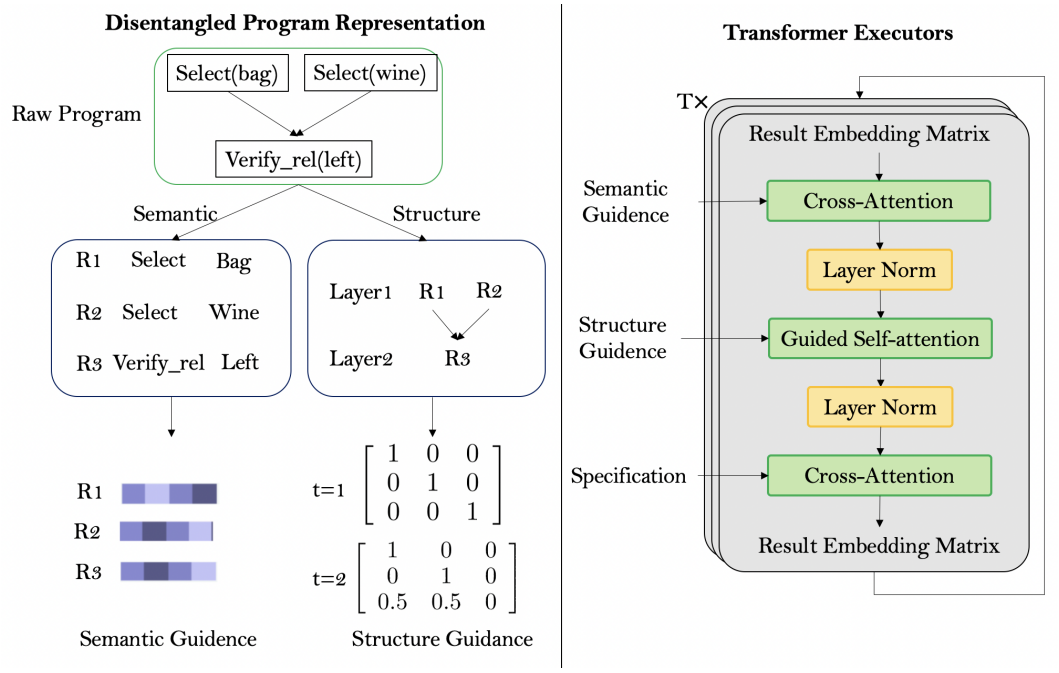

Figure 3: Model architecture of Program-guided Transformer (ProT). Left: we propose a disentangled program representation to leverage both semantics and structures of the program. Right: given structure and semantic guidance offered by a program, ProT updates result embedding by combining internal program execution features and object-centric features from specifications.

by the routine set R_τ, we only update $Z[R_\tau]$ at timestep τ and other entries of Z would be kept the same. After executing all timesteps, as shown in Fig 2, the latent result embeddings Z are decoded via MLPs to get explicit result distributions $\{O_q^i\}_{q=1}^{\|R\|} = \text{MLP}(Z)$.

We show the architecture of transformer executors in the right of Fig 3. Given the current result embedding matrix Z, we conduct standard cross-attention [22] to get hidden feature $H_1 = \text{CrossAtt}(P^{se}, Z)$. Then we propagate the information to a new layer via the guided self-attention proposed by us. The idea of guided self-attention is to use the structure guidance as a mask to force the connection relationship between layers. Formally,

$$H_2 = \text{GuidedSelfAtt}(H_1, P^{st}) = \text{softmax}(\frac{H_1 H_1^T \odot P^{st}}{\sqrt{D}})H_1. \tag{2}$$

In the above equation, we show only one head of self-attention for simplicity, while in experiments, we use multi-head self-attention [24]. After that, we apply cross-attention again to get updated result embedding matrix $Z = \text{CrossAtt}(S, H_2)$ where S is the specification.

5 Applications of Program-guided Transformer

In this section, we specifically consider two program-guided tasks and describe how we address them by our proposed Program-guided Transformer.

5.1 Program-guided Visual Reasoning

Task Description The typical visual question answering (VQA) task [2] aims at predicting an answer given an image and a question. We adopt an off-the-shelf program generation model to transfer each question to its corresponding program. In the GQA dataset, a simple seq2seq program

generation model achieves almost perfect accuracy.[1] In this way, the visual question answering task is reduced to a program-guided task, which requires executing the program on the image (specification) to get a predicted answer.

Model Details Follow the overall model shown in Fig 2, we parse the image by a pre-trained detection transformer [4] to get object-level visual features $S^v \in \mathbb{R}^{N \times D}$ where v denotes current instance, N is the number of objects and D is the hidden dimension. The language-aware visual object features S^v and the program P^v are sent into Program-guided Transformer $\mathcal{E}_\Phi^\alpha$. The execution results for the program P^v are $\{O_q^v\}_{q=1}^Q = \mathcal{E}_\Phi^v(S^v, P^v)$ where Q is the number of routines in P^v. We get the predicted answer distribution a from the final execution result O_Q^v through $a = \text{MLP}(O_Q^v)$. We get the ground true execution results $\{\widetilde{O}_q^v\}_{q=1}^Q$ by executing the program P^v on the provided ground true training scene graphs [14]. We detail how we encode the ground true execution results into continuous vectors in the supplementary.

Training Targets Given the ground true answer $\widetilde{a}$, the training loss for this instance is: $\mathcal{L}_{ALL}^v = \Sigma_{\tau=0}^T \mathcal{L}_{KL}(o_\tau^v, \widetilde{o}_\tau^v) + \eta^v \mathcal{L}_{CS}(a, \widetilde{a})$ where $\mathcal{L}_{KL}$ is the KL divergence, $\mathcal{L}_{CS}$ is the cross entropy loss, and η^v is a balancing factor.

5.2 Program-guided Policy Learning

Task Description Program-guided policy learning requires the agent to perform following a given program P^p where p denotes current task instance. [20]. At time-step t, the environment provides the agent with current state s_t. The agent takes an action a_t from a predefined action space $\mathcal{A}$ following his policy π and receives a reward r_t from the environment. When the agent successfully executes one instruction in P^p, the environment gives the agent an $r_t = +1$ reward. Otherwise, r_t is zero. The goal of program-guided policy learning is to learn a best policy to maximize the discounted return $\Sigma_{\tau=0}^T \gamma^\tau r_t$ where γ is the discounted factor given by the task.

Model Details Follow the overall model in Fig 2, we encode the state s_t via an MLP and denote $S_t^p \in \mathbb{R}^{N \times D}$ to be the encoded state (specification) where N is the number of object and D is the hidden dimension. In our experiments, S_t^p is the encoded object information for N grids. Both the encoded state S_t^p and the program P^p are sent to the Program-guided Transformer. The Program-guided Transformer outputs the execution result embeddings $\{O_\tau^p\}_{\tau=0}^T = \mathcal{E}_\Phi^p(S^p, P^p)$. The execution result embeddings implicitly encode the states during the execution of program. From the last execution result, we get the predicted action distribution through $a = \text{MLP}(O_T^p)$.

Training Targets Given the ground true states $\{\widetilde{O}_\tau^p\}_{\tau=0}$ after executing P_v (available at training time), we add a supervised loss on execution traces $\mathcal{L}_{res} = \Sigma_{\tau=0}^T \mathcal{L}_{KL}(o_\tau^p, \widetilde{o}_\tau^p)$. The policy is learned via the standard A2C [21] and we denote $\mathcal{L}_{A2C}$ as the policy learning loss (see supplementary for details). The overall training loss is a weighted average of the above losses: $\mathcal{L}_{ALL}^p = \mathcal{L}_{res} + \eta^p \mathcal{L}_{A2C}(\widetilde{a})$ where η^p is a balancing factor.

6 Experiments

The results on GQA comparing to previous baselines are in Table 1.

7 Conclusion

[1] In GQA dataset [14], we found that a simple seq2seq model can achieve 98.1% validation accuracy to convert a natural language question into a program. The concurrent work [5] also found similar facts.

Table 1: Comparison of ProT with previous models on the test split of GQA dataset.

Model	Binary	Open	Consistency	Validity	Plausibility	Distribution	Acc
Human	91.20	87.40	98.40	98.90	97.20	-	89.30
BottomUp	66.64	34.83	78.71	96.18	84.57	5.98	49.74
MAC	71.23	38.91	81.59	96.16	84.48	5.34	54.06
Dream	77.84	43.72	91.71	96.38	85.48	8.40	59.72
LXMERT	77.76	44.97	92.84	96.30	85.19	8.31	60.34
NSM	78.94	49.25	93.25	96.41	84.28	3.71	63.17
MMN	78.90	44.89	92.73	95.21	83.12	3.98	60.83
ProT(Ours)	**79.12**	**51.45**	**93.45**	**97.12**	**86.21**	**3.66**	**65.14**

References

[1] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[2] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C Lawrence Zitnick, and Devi Parikh. Vqa: Visual question answering. In *Proceedings of the IEEE international conference on computer vision*, pages 2425–2433, 2015.

[3] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

[4] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *European Conference on Computer Vision*, pages 213–229. Springer, 2020.

[5] Wenhu Chen, Zhe Gan, Linjie Li, Yu Cheng, William Wang, and Jingjing Liu. Meta module network for compositional visual reasoning. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 655–664, 2021.

[6] Xinyun Chen, Chang Liu, and Dawn Song. Towards synthesizing complex programs from input-output examples. *arXiv preprint arXiv:1706.01284*, 2017.

[7] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

[8] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[9] Jacob Devlin, Rudy Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural program meta-induction. *arXiv preprint arXiv:1710.04157*, 2017.

[10] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

[11] Alan Dix, Janet Finlay, Gregory D Abowd, and Russell Beale. Human-computer interaction. *Harlow ua*, 2000.

[12] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B Tenenbaum. Learning to infer graphics programs from hand-drawn images. *arXiv preprint arXiv:1707.09627*, 2017.

[13] Roy Fox, Richard Shin, Sanjay Krishnan, Ken Goldberg, Dawn Song, and Ion Stoica. Parametrized hierarchical procedures for neural programming. *ICLR 2018*, 2018.

[14] Drew A Hudson and Christopher D Manning. Gqa: A new dataset for real-world visual reasoning and compositional question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6700–6709, 2019.

[15] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

[16] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[17] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B. Tenenbaum, and Jiajun Wu. The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision. In *International Conference on Learning Representations*, 2019.

[18] Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

[19] Richard Shin, Illia Polosukhin, and Dawn Song. Towards specification-directed program repair. 2018.

[20] Shao-Hua Sun, Te-Lin Wu, and Joseph J Lim. Program guided agent. In *International Conference on Learning Representations*, 2019.

[21] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[22] Hao Tan and Mohit Bansal. Lxmert: Learning cross-modality encoder representations from transformers. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5103–5114, 2019.

[23] Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T Freeman, Joshua B Tenenbaum, and Jiajun Wu. Learning to infer and execute 3d shape programs. *arXiv preprint arXiv:1901.02875*, 2019.

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[25] Da Xiao, Jo-Yu Liao, and Xingyuan Yuan. Improving the universality and learnability of neural programmer-interpreters with combinator abstraction. *arXiv preprint arXiv:1802.02696*, 2018.

[26] Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Joshua B Tenenbaum. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. *arXiv preprint arXiv:1810.02338*, 2018.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [Yes]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [Yes]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [Yes]

5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [Yes]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [Yes]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [Yes]