分层神经程序合成

Linghan Zhong[1], Ryan Lindeborg[1],Jesse Zhang[1],Joseph J. Lim[2 3],Shao-Hua Sun[4]南加州大学， [2]韩国科学技术院， [3]Naver AI Lab 的 AI 顾问， [4]国立台湾大学

arXiv:2303.06018v1 [cs.SE] 9 Mar 2023

摘要

程序合成旨在自动构建满足给定任务规范的可读程序，例如输入/输出对或示例。近期研究已在多个领域取得令人鼓舞的成果，例如字符串转换、张量操作，以及描述具身智能体的行为。大多数现有程序合成方法旨在从零开始合成程序，逐词元、逐行生成程序。这从根本上限制了这些方法扩展到合成更长或更复杂程序的能力。在这项工作中，我们提出了一种可扩展的程序合成框架，通过分层组合程序来合成程序。具体而言，我们首先学习一个任务嵌入空间和一个程序解码器，后者能够将任务嵌入解码为程序。随后，我们训练一个高层模块，从长程序中理解任务规范（例.如，输入/输出对或示例），并生成一系列任务嵌入，再由程序解码器对其进行解码并组合，从而得到合成程序。我们在一个使用输入/输出对的字符串转换领域中，对所提出的框架进行了广泛评估。实验结果表明，与先前程序合成研究所考虑的程序相比，所提出的框架能够合成明显更长、更复杂的程序。网站：https://thoughtp0lice.github.io/hnps_web/

1 引言

程序合成旨在自动合成以领域特定语言（DSL）编写、满足给定任务规范的程序。近年来，在字符串和张量操作、计算机命令、图形程序以及描述具身智能体行为等多个领域都取得了令人鼓舞的成果（Devlin 等人，2017b；Balog 等人，2017；Bunel 等人，2018；Wu 等人，2017；Lin 等人，2018；Sun 等人，2018；Trivedi 等人，2021）。
大多数现有的程序合成方法都旨在按顺序合成程序——根据先前生成的词元逐个生成程序词元，直至合成过程结束。然而，如果对程序空间没有任何归纳偏置，那么当我们从行为简单的短程序扩展到行为更复杂的长程序时，逐个词元合成目标程序就会变得困难。此外，这些方法通常会通过损失函数来学习最大化每个词元的似然，而无论这些词元会如何影响程序行为，其重要性在损失函数中都一视同仁。这种训练方案存在程序别名问题——不同词元会产生相同的程序行为——以及对关键词元的弱监督问题：替换单个词元就会极大地改变程序输出。例如，将非关键令牌替换为另一个令牌（如在 C++ 中大多可以互换的“nullptr”和“NULL”）时，程序输出可能不会显著变化；但替换关键函数的输入值（即数组的位置索引词元从 a[0] 变为 a[1]）时，输出则会完全改变。因此，我们认为，这种逐词元生成和训练范式从根本上限制了当前程序合成方法扩展到合成更复杂程序的能力。

在这项工作中，我们旨在开发一种程序合成框架，使其能够扩展到更长的程序，从而产生更复杂的行为。我们的核心思想是，将具有简单行为的短程序组合起来，形成具有更复杂行为的长程序。然而，在大多数情况下，有意义的短程序空间难以穷尽——搜索正确的短程序序列

以组合成长程序仍然颇具挑战。因此，我们不仅提出了一种能够学习分层组合程序的架构，并将其称为分层神经程序合成器（HNPS），还设计了一种训练方案，以高效地学习这一分层合成器。

为此，我们首先构建一个短程序数据集，其中包含短程序及其对应的输入/输出对。然后，我们学习一个任务嵌入空间，用以表示短程序行为的空间，并训练一个程序解码器，将任务嵌入解码为短程序。接下来，我们创建一个组合程序数据集，其中的组合程序由短程序按顺序组合而成。随后，我们在该数据集上训练一个“程序组合器”，以生成一系列任务嵌入，每个嵌入代表一个潜在短程序；程序解码器将这些嵌入解码为短程序，再按顺序组合这些短程序，从而得到合成程序。该训练方案使我们能够通过逐程序组合而非逐词元组合来生成程序，因此可以利用任务嵌入提供额外监督，而不只是最大化基于生成的程序词元的似然。

为了评估所提出的框架，我们在字符串转换领域研究程序合成，其中任务规范由输入/输出字符串组成（例如，[John-01/John、Mary-02/Mary]），程序则由一系列字符串操作组成（例如，SubStr(0, Regex("-", 0, Start))），与 Devlin 等人（2017b）和 Hong 等人（2021）的研究类似。实验表明，我们的框架在合成更长、能够实现更复杂字符串操作的程序方面表现更优。此外，我们还进行了广泛的消融研究，验证了层次化架构以及由所提出的训练方案和组合程序数据集所支持的潜在监督的有效性。

2 相关工作

神经程序归纳。程序归纳方法（Graves 等，2014；Neelakantan 等，2015年；Kaiser 和 Sutskever，2016年；Reed 和 De Freitas，2016年；Gaunt 等，2017年；Burke 等，2019年；Xiao 等，2018年；Lázaro-Gredilla 等，2019年；Pierrot 等，2019年；Sun 等，2020年）旨在隐式归纳底层程序，以模仿给定任务规范中展示的行为。为此，大多数方法采用外部记忆（Graves 等，2014；Zaremba 等，2016年）、带有模块化监督的模块化框架（Reed 和 De Freitas，2016年；Cai 等，2017年；Xu 等，2018年），或复杂的注意力机制（Devlin 等，2017a）来获取程序化行为。与此不同，我们关注的是显式合成可读程序。

神经程序合成。程序合成方法（Balog 等人，2017年；Bošnjak 等人，2017年；Parisotto 等人，2017年；Devlin 等人，2017b；Bunel 等人，2018年；Shin 等人，2018年；Chen 等人，2019年；Liu 等人，2019年；Sun 等人，2018年；Lin 等人，2018年；Abolafia 等人，2018年；Verma 等人，2018年；Liao 等人，2019年；Ellis 等人，2019年；2020年；Liskowski 等人，2020年；Hong 等人，2021年；Silver 等人，2020年；Trivedi 等人，2021；Liu 等人，2023年）会显式生成可执行的程序，以根据给定的任务规范完成任务。这些任务规范的形式多种多样，既可以是程序输入/输出对，也可以是智能体行为的示范。与这些研究相比，我们尤其关注开发一种能够合成更长、具有复杂行为的程序的框架。其他方法，例如 BUSTLE（Odena 等人，2021年），则着重于将神经方法融入自底向上搜索，以组合方式迭代遍历大量候选程序。相比之下，HNPS 只合成一个程序预测。

近年来，预训练大语言模型（大语言模型）已被用于根据任务规范进行程序合成，例如描述期望程序行为的文本或直接的输入/输出示例（Chen 等人，2021；Nijkamp 等人，2022；Li 等人，2022）。这些大型模型在海量代码语料上进行训练，因此在合成程序方面表现出色。然而，它们仍然按顺序生成词元，因此可能遇到与领域特定的逐词元合成方法相同的问题。此外，基于 Transformer 的模型（Vaswani 等人，2017）已通过新颖的、针对程序合成定制的注意力机制进行了改进（Shi 等人，2022）。我们的方法有所不同，它除了对单个程序词元进行监督外，还针对潜在嵌入向量进行监督。我们提出的训练框架与大语言模型取得的进展相辅相成，未来可与此类架构及海量数据集结合使用。

3 问题定义

我们的目标是根据任务规范合成程序。本节将正式阐述我们对程序和任务规范的定义，以及具体的问题定义。

程序与领域特定语言。 本文所研究的程序基于一种用于字符串转换任务的领域特定语言（DSL）构建，与（Devlin 等人（2017b）；Hong 等人（2021））中使用的字符串转换任务 DSL 类似。该 DSL 界定了有效程序的可能空间，并支持对程序进行采样。图 1 展示了我们的 DSL 示例。该 DSL 定义了一组操作（例如，SubStr(k_1，k_2)、Concat(e_1,e_2, ...)、ConstStr(s)）及其参数，例如数字（例如， k_1, k_2）和字符串 (*e.g.*， s)。程序由一系列表达式组成，这些表达式会输出输入字符串的一个子字符串或一个常量字符串。随后，这些表达式的输出会拼接起来，生成程序的最终输出字符串。例如，程序 Concat(SubStr(0, 1), ConstStr("!")) 接收输入字符串 John-01，并生成输出字符串 J!。

程序 $\rho :=$ 拼接 (e_1 e_2 e_3 ...) 表达式 $e :=$ 常量字符串(s) | 子字符串 (q_1 q_2) 位置 $q :=$ 常量位置 (k) | 正则表达式 (x n b) 边界 b $:=$ 起始位置 | 结束 正则表达式 $x := s$ | 单词 | 数字 | 字母数字 | ... | 字符 位置索引 $k := -20$ | -19| ... | 19 | 20正则表达式索引 $n := -3$ | -2| ... | 2 | 3字符 $s :=$ "空格" | "." | "," | ... |"?"

图 1：用于构造字符串操作程序的领域特定语言（DSL）。

任务规范。任务规范描述了用户期望的意图，其中包含输入和期望的程序执行结果。例如，[输入：October，输出：oct] 可以作为字符串转换程序的输入/输出对任务规范。请注意，多个不同的程序都可能满足这个示例规范，因此在实践中，通常会向程序合成网络提供多个输入/输出对。重要的是，任务规范与程序之间是多对多的映射关系——多个任务规范可以用来描述同一个期望的程序，而许多程序也可能满足同一个任务规范，这表明程序合成具有一定难度。一般而言，提供任务规范应该比要求非专家用户编写程序更容易。任务规范的常用表示形式包括输入/输出（I/O）对Devlin 等人（2017b）；Hong 等人（2021）、示范Sun 等人（2018）；Trivedi 等人（2021）；Burke 等人（2019），或自然语言指令 Li 等人（2022）；Nijkamp 等人（2022）。

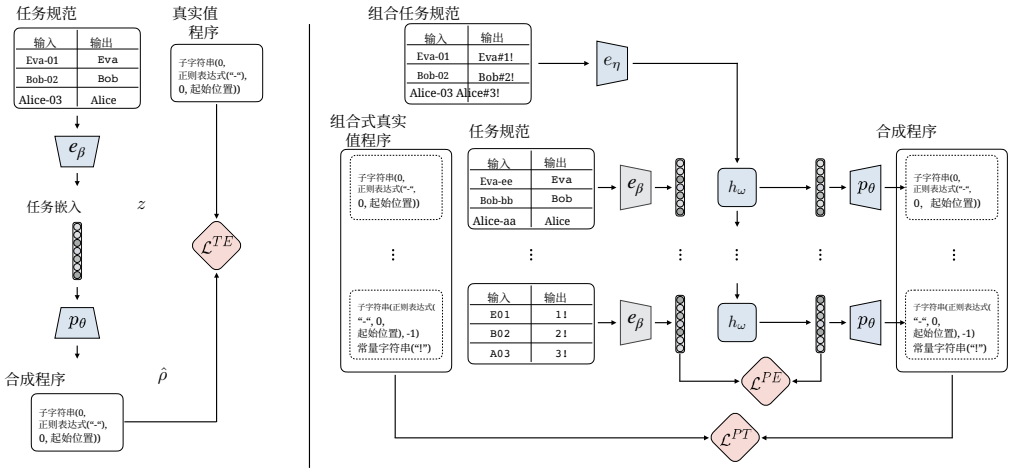
问题定义。 我们开发了一种框架，能够根据上述任务规范合成程序。具体而言，我们关注能够产生更复杂执行结果的长程序。具体来说，我们考虑字符串转换领域，其中任务规范由一组输入/输出字符串对组成（例如， [John-`01`/John，Mary-`02`/Mary，Bob-`03`/Bob]），而程序由一系列字符串操作组成（例如，SubStr(0, Regex("-", 0, 起始位置))）。我们考虑的长程序通常最长可达 63 个词元，并包含 2 至 4 个常量字符串或子字符串表达式。

4 方法

我们的目标是设计一种框架，使其能够根据任务规范合成长程序。为此，我们提出了分层神经程序合成器（HNPS），这是一种能够学习组合程序以构成长程序的框架，如图 2 所示。接下来，我们将介绍如何在第 4.1 节中通过短程序和较简单的任务规范学习程序嵌入空间。然后，我们将在第 4.2 节中介绍如何构建用于学习组合程序的数据集。最后，我们将在第 4.3节中介绍 HNPS 如何通过组合程序，学习根据任务规范合成程序。

4.1 学习任务嵌入空间

为了学习任务嵌入空间，我们构建了一个短程序数据集 D_{short}，方法是从领域特定语言中随机采样程序，并通过使用随机生成的输入运行采样到的程序来生成任务规范（例如，输入/输出对）。然后，我们从 D_{short} 中采样程序 ρ 及其对应的任务规范 σ，据此训练一个神经程序合成模型，该模型由一个任务



(a) 学习任务嵌入空间　　(b) 分层神经程序合成

图 2：**框架概述**。（**a**）学习任务嵌入空间：我们首先通过训练任务编码器 e_β 来学习任务嵌入空间，将短程序的任务规范（例如输入/输出对） σ 编码为任务嵌入，再由 p_θ 将其解码为程序 $\hat{\rho}$。 e_β 和 p_θ 通过任务嵌入损失 $\mathcal{L}^{\text{TE}}_{\theta,\beta}$进行训练。（**b**）分层神经程序合成：随后，我们训练一个组合后的任务编码器 e_η，并使用任务编码器 e_β 的权重对其进行初始化，以将给定的长程序输入/输出对编码为嵌入。程序组合器 h_ω 使用该编码输入，持续预测任务嵌入，并由预训练的 p_θ 解码器进行解码。最后，将这些解码后的程序组合起来，形成完整的合成程序。模型通过程序令牌损失 $\mathcal{L}^{\text{PT}}$ 进行训练以合成长程序，并通过程序嵌入损失 $\mathcal{L}^{\text{PE}}$使程序嵌入与真实程序嵌入相匹配。

编码器 e_β（将任务规范 σ 编码为任务嵌入）和一个程序解码器 p_θ（根据嵌入合成程序）组成。任务编码器 e_β 和程序解码器 p_θ都是循环网络，通过优化逐词元交叉熵损失进行训练，以最大化真实程序词元的似然。我们将其称为任务嵌入损失，$\mathcal{L}^{\text{TE}}_{\theta,\beta}$:

$$\mathcal{L}^{\text{TE}}_{\theta,\beta}(\boldsymbol{\rho}) = -\mathbb{E}_{(\boldsymbol{\sigma},\boldsymbol{\rho})\sim D_{\text{short}}}\left[\log p_\theta\left(\boldsymbol{\rho}|e_\beta\left(\boldsymbol{\sigma}\right)\right)\right]. \tag{1}$$

理想情况下，这个学习得到的嵌入空间可以作为表示所需程序行为的良好空间，将相似的程序及其任务规范聚集得更近，同时让不相似的程序及其任务规范彼此相距更远。

4.2 创建组合长程序数据集

我们的最终目标是学习如何组合短程序，形成更长的程序。因此，我们创建了一个数据集 D_{composed}，其中包含通过组合程序数据集 D_{short}中的较短程序得到的长程序。我们将这些组合长程序记为 $\bar{\rho} = [\rho_1, ..., \rho_N]$，其中每个 ρ_i 都是一个短程序。每个组合程序的任务规范均采用 5.1 节详述的技术随机采样。这样的组合程序数据集将使我们能够获取每个长程序如何组合而成的真实值信息，并让模型学习程序组合。对于 D_{composed}中的所有长程序，我们还会纳入其子程序的任务嵌入，这些嵌入由任务编码器 e_β 根据 D_{short}中短程序的任务规范计算得出。任务嵌入中的信息将使我们能够训练模型，充分利用所学习的任务嵌入空间。

4.3 分层神经程序合成　　ESIS

为了按程序而非按词元逐个合成程序，我们提议利用在第 4.1 节中学习到的程序解码器 p_θ 作为低层模块，以生成短程序。

然后，我们采用一个程序组合器 h_ω，它会生成一系列任务嵌入，这些嵌入可被解码为短程序，并按顺序组合成一个长程序。[1]

具体而言，我们的分层神经程序合成器由以下三个组件构成。组合任务编码器 e_η 学习将给定的长程序任务规范编码为嵌入（即，字符串输入/输出对）。由于来自 D_{short} 和 D_{composed} 的任务规范相似，因此可以将任务编码器 e_β 的参数用作组合任务编码器e_η的初始化参数。然后，程序组合器 h_ω 以组合任务编码器的输出为输入，依次生成任务嵌入，直至生成满足任务规范的长程序（对于字符串转换任务，即程序能够根据所有给定的输入字符串生成正确的输出），或者达到最大程序长度。最后，在构建任务嵌入空间期间训练的程序解码器p_θ 将生成相应的短程序。由于难以准确预测任务嵌入，因此程序解码器 p_θ 会与其他组件一起进行微调，并允许任务嵌入空间发生漂移。为了训练分层神经程序合成器，我们提出优化以下目标。

程序词元损失。 程序词元损失 $\mathcal{L}^{\text{PT}}$ 旨在通过交叉熵损失，最大化合成程序中真实值程序词元的对数似然。对组合程序 $\bar{\rho}$ 及其任务规范 σ进行采样后，我们应用交叉熵损失，最大化构成 $\bar{\rho}$的每个独立短程序 $\rho_1 ... \rho_N$ 的对数似然。该损失会在全部三个网络中传播。请注意，h_ω 是一个循环网络，因此其预测取决于先前的短程序；因此，来自后续短程序的损失将通过时间反向传播，以更新 e_η 和 h_ω，从而改善对整个长程序的整体重建：

$$\mathcal{L}^{\text{PT}}_{\theta,\omega,\eta} = -\mathbb{E}_{(\boldsymbol{\sigma},\bar{\boldsymbol{\rho}})\sim D_{\text{composed}}}\left[\frac{1}{N}\sum_{i=1}^{N}\log p_\theta\left(\boldsymbol{\rho_i}|h_\omega\left(e_\eta\left(\boldsymbol{\sigma}\right)\right)\right)\right]. \tag{2}$$

程序嵌入损失。 尽管任务嵌入空间可能会随着 p_θ 的微调而变化，但初始任务嵌入空间中的嵌入仍能为程序组合器提供有用的依据，使其输出合理的任务嵌入。为了对程序组合器的输出空间进行正则化，我们使用原始的固定任务编码器 e_β，对 D_{composed} 中构成长程序的每个短程序的任务规范 ρ_i 进行嵌入。随后，我们应用程序嵌入损失 $\mathcal{L}^{\text{PE}}$，旨在最小化真实值（即编码后的）程序嵌入与程序组合器生成的程序嵌入之间的欧几里得距离，该损失适用于组合程序中的所有 N 子程序：

$$\mathcal{L}^{\text{PE}}_{\eta,\omega} = \mathbb{E}_{(\boldsymbol{\sigma},\boldsymbol{\sigma_1},\ldots,\boldsymbol{\sigma_N})\sim D_{\text{composed}}}\left[\frac{1}{N}\sum_{i=1}^{N}||h_\omega\left(e_\eta\left(\boldsymbol{\sigma}\right)\right) - e_\beta\left(\boldsymbol{\sigma_i}\right)||_2\right]. \tag{3}$$

在第5.3节的实验中，我们发现，对于训练数据较少的情形，这种损失尤其有用，能够引导程序组合器学习合理的程序嵌入空间。

总之，我们提出以下目标函数：通过处理给定的任务规范并组合程序，学习合成程序：$\min_{\eta,\ \omega,\ \theta}\mathcal{L}^{\text{PT}} + \lambda\mathcal{L}^{\text{PE}}$，其中 λ 用于控制各项损失的重要性。请注意，程序嵌入损失源于我们的一项贡献：组合由较短程序构成的数据集，以构建包含可分解长程序的数据集。

5 实验

我们在上述字符串转换任务所生成的数据集上评估框架，详见第 5.1 节。随后，我们通过与第 5.2 节详述的基线方法和消融实验进行比较，说明 HNPS 的设计决策。最后，我们在第 5.3 节展示实验结果。

[1]我们承认，我们的模型仅限于对子程序进行线性组合。然而，我们认为，具体的组合模式与我们的主要关注点正交；我们的主要关注点是引入层次化组合，以此作为实现可扩展程序合成的一种方案。我们希望在未来的工作中将其扩展到更通用的组合模型。

5.1 数据集

我们创建了两个数据集，D_{short} 和 D_{composed}，其中 D_{composed} 由长程序组成，这些长程序是通过组合 D_{short}中的短程序得到的。生成过程中会同时创建字符串输入/输出对及其对应的程序。D_{short} 包含 100,000 个程序，每个程序仅包含一个常量字符串或子字符串表达式。我们采用与先前工作（Trivedi 等人，2021）类似的方式，从我们的领域特定语言中随机采样，以创建这些短程序。对于每个采样得到的程序，我们随机生成 1,000 个字符串作为潜在的输入字符串，并将其中前 20 个能够成功执行的输入字符串及其对应的输出字符串存入数据集。如果生成的 1,000 个输入字符串中，不足 20 个能在某个程序上成功执行，则丢弃该程序。我们重复这一过程，直到生成目标数量的程序，且每个程序都包含 20 个输入/输出对作为任务规范。对于 D_{composed}，我们利用领域特定语言的特性，将短程序中的表达式线性组合成一个长程序。我们通过组合从 D_{short}中随机采样的 2–4 个程序，生成 200,000 个组合程序。程序组合过程详见附录。对于 D_{composed}中的每个组合程序，我们使用相同的方法生成输入/输出对。我们将每个数据集划分为训练集和验证集，以便在训练过程中评估模型泛化性能。

最后，我们生成了一个长程序数据集 D_{long}，其中包含 30,000 个程序，每个程序包含 2-4 个表达式（因此程序最长可达 63 个词元），这些程序是通过直接从我们的领域特定语言中采样得到的。值得注意的是，D_{long} 并非通过组合短程序构建而成。我们在这个测试数据集上评估 HNPS 的性能，以检验其合成未见过的长程序的能力。

5.2 基线与消融实验

我们将 HNPS 与基线方法及其变体进行比较。以下基线方法代表了神经合成方法这一类方法，它们旨在以逐词元的方式合成程序。

- **Naïve**：一种朴素的合成基线方法，在 $D_{\text{组合}}$上训练；该方法将输入/输出对的词元编码为嵌入，并利用该嵌入合成完整长度的程序。预计这种基线方法难以从头开始学习合成具有复杂行为的长程序。
- **朴素短版**：朴素基线，但模型仅在 $D_{\text{短}}$任务上训练。预期该基线能从 $D_{\text{短}}$ 任务中充分学习，但零样本泛化到 $D_{\text{长}}$任务的表现较差。
- **Naïve-short-finetune**：Naïve-short 基线，但模型在 $D_{\text{组合}}$任务上进行微调。该基线以与我们提出的方法相同的顺序，同时利用 $D_{\text{短}}$ 任务和 $D_{\text{组合}}$ 任务。区别在于，它采用扁平架构，没有像我们的方法那样使用层次结构。该基线与我们方法之间的性能差距应能证明我们所提出框架的层次化设计是合理的。

我们还纳入了一个最先进的基于搜索的程序合成基线（Odena 等人，2021年）。

- **BUSTLE**：一种自底向上的搜索技术，专为字符串操作等任务设计。该技术通过组合式地遍历程序领域特定语言，合成满足任务规范的程序。为使搜索更有针对性，该方法会对某些子表达式进行排序，使其优先于其他子表达式接受搜索。运行时间限制设为 500,000 个表达式。

为论证我们的设计选择，我们在消融研究中考虑了以下 HNPS 变体 s.

- **H-朴素-PT**：一种朴素的分层合成基线，其中解码器从头开始训练（即，我们跳过嵌入空间学习阶段），且仅应用程序令牌损失 $\mathcal{L}_{\text{PT}}$。该消融实验学习组合短程序以创建更长的程序，但不采用两阶段学习方式。该消融实验与我们提出的框架之间的性能差距，应能证明学习任务嵌入空间的重要性。
- **HNPS-PT**：一种仅应用程序令牌损失 $\mathcal{L}_{\text{PT}}$的消融实验；该损失仍能为相对较短的程序提供强有力的监督。该消融实验首先从 $D_{\text{短程序}}$ 中学习任务嵌入空间，然后在优化 $\mathcal{L}_{\text{PT}}$的同时从 $D_{\text{组合程序}}$ 中学习。鉴于真实任务嵌入可能存在噪声，我们在此旨在分析，缺少程序嵌入损失会在何种情况下对执行准确率造成更大或更小的不利影响。
- **HNPS（HNPS-PT+PE）**：完整的 HNPS 方法。首先，它从$D_{\text{短程序}}$中学习任务嵌入空间。然后，在从 $D_{\text{组合程序}}$中学习的同时，优化两种损失（程序令牌损失 $\mathcal{L}_{\text{PT}}$ 和程序嵌入损失 $\mathcal{L}_{\text{PE}}$）。

5.3 结果

我们展示了 HNPS 和对比方法在合成程序方面的结果，涵盖未见过的、非组合的长程序数据集，以及 D_{composed} 中测试集里的未见程序（见第5.3.1节）。随后，我们分析 HNPS 学到的任务嵌入空间，以及它如何帮助合成长程序（见5.3.2节）。最后，我们仔细考察程序嵌入损失对合成不同长度程序以及不同数据集大小的影响（见第5.3.3节）。

5.3.1 长程序的执行准确率

表1：**未见测试程序上的性能**。 我们在数据集 D_{long} 的未见测试程序上评估执行准确率，其中执行准确率指合成程序对所有给定输入字符串均产生正确输出的百分比。横轴表示真实程序的长度（程序词元数量）。

	程序词元数量			
方法	**10-25**	**25-40**	**40-55**	**55-70**
Naïve	42.99%	11.88%	0.61%	0.01%
朴素短版	0.00%	0.00%	0.00%	0.00%
Naïve-short-finetune	42.68%	16.75%	1.93%	0.02%
BUSTLE	10.72%	0.13%	0.00%	0.00%
H-朴素-PT	60.78%	27.12%	5.99%	0.41%
HNPS-PT	**65.76%**	**32.49%**	9.4%	0.78%
HNPS（我们的完整版本）	64.25%	32.22%	**9.71%**	**0.87%**

非组合长程序上的评估。 我们在数据集 D_{long} 中的未见过的测试程序上评估执行准确率，其中执行准确率是指对所有给定输入字符串都能生成正确输出的合成程序所占的百分比。

我们的实验结果如表1所示。此外，我们在图3中可视化了不同程序长度下的相对性能，并将 HNPS 的执行准确率缩放至 1.0，据此对每种方法的执行准确率进行归一化。对于长度超过 40 个词元的程序，我们的方法优于基线方法和消融变体：在长度为 40–55 个词元的程序上，执行准确率为 9.71%；在长度为 55–70 个词元的程序上，执行准确率为 0.87%。对于长度短于 40 个词元的程序，不应用程序嵌入损失的 HNPS-PT 性能最佳：在长度为 10–25 个词元的程序上，执行准确率为 65.76%；在长度为 25–40 个词元的程序上，执行准确率为 32.49%。我们将在后续章节进一步分析 HNPS-PT 与 HNPS 在较短程序上的性能差异。

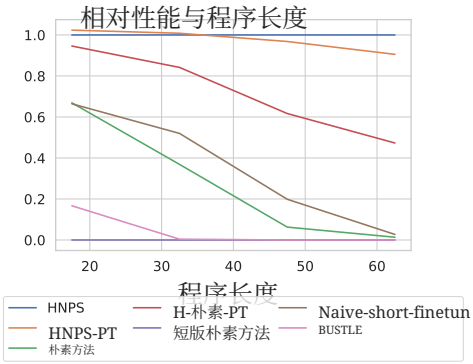


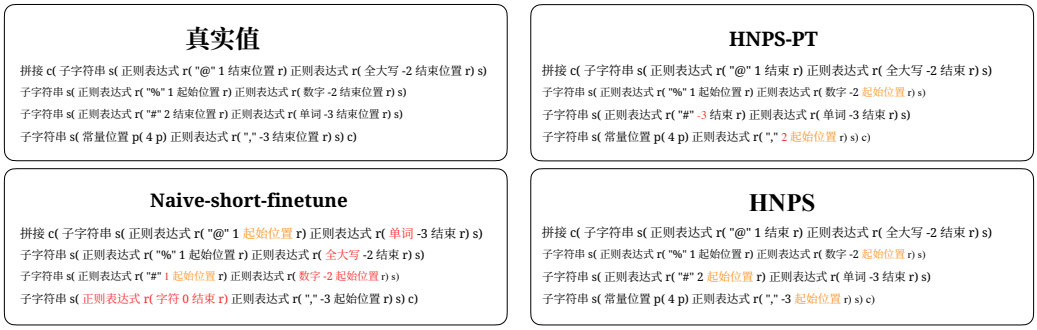
图3：在 D_{long} 上评估的所有基线方法与 HNPS 的相对性能。我们将 HNPS 的执行准确率缩放至 1.0，并据此对每种方法的执行准确率进行归一化。

仅采用层次化架构，不使用
在预训练任务嵌入空间中，H-朴素-PT 仍然能够胜过其他非层次化方法。然而，随着程序长度增加，其相对于 HNPS 的性能显著下降；在包含 55–70 个词元的程序上，其执行准确率不到 HNPS 的 50%。这表明任务嵌入空间对于训练层次化模型至关重要。

非层次化基线方法 Naïve、朴素短版和 Naïve-short-finetune 的表现显著逊于我们的方法。在包含 55–70 个词元的程序上，这三种基线方法的执行准确率都接近于零，即使是 Naïve-short-finetune 也不例外；它在相同数据集上对与 HNPS 相同的任务编码器和程序解码器进行了微调。这进一步支持了我们的观点：逐词元生成方案从根本上限制了程序合成模型扩展到更长程序的能力。

基于搜索的方法 BUSTLE 表现也不佳，在包含 10-25 个词元和 25-40 个词元的程序上，执行准确率分别仅为 10.72% 和 0.13%。对于长度超过 40 个词元的程序，BUSTLE 从未能合成出完全符合任务规范的程序。像我们评估的这种较长程序合成任务，似乎对自底向上

图4： **定性示例**。 我们展示了由 HNPS、HNPS-PT 和 Naïve-short-finetune 生成的程序示例，并与真实程序进行比较。词元变动不影响程序行为的非关键错误以橙色标记。词元变动导致程序行为不同的关键错误以红色标记。



搜索方法来说过于复杂。尽管 BUSTLE 能够优先搜索更有可能出现在最终解中的子表达式，但它无法高效地搜索庞大的可能程序空间并找到解。

组合长程序评估。 我们在训练程序和未见过的 测试程序上评估所有方法的执行准确率，使用我们的 D_{composed} 数据集。具体结果详见附录D.2节。与对 D_{long} 的评估结果类似，HNPS 和 HNPS-PT 的表现优于 H-朴素-PT，这进一步凸显了构建任务嵌入空间的重要性。我们还观察到，非层次化基线（Naïve-*）在训练程序和测试程序上的性能均显著低于层次化方法。这支持了我们的观点：缺少层次化架构时，模型拟合长程序分布的能力从根本上受到限制。

我们还分析了 D_{composed}上的“泛化差距”，即，训练集与测试集性能之间的差异。HNPS的泛化差距小于HNPS-PT和H-朴素-PT。这表明，程序嵌入损失有助于防止层次化模型过拟合。这可能是因为在 D_{short} 训练期间，程序解码器学会了合成一组多样化的短程序。在 D_{composed} 上训练时，嵌入损失会将解码器锚定到原始任务嵌入空间，促使其不只对D_{composed} 训练集所需的短程序过拟合。因此，程序嵌入损失不仅有助于组合器创建更长的程序（我们将在5.3.2节中进一步探讨），也有助于确保模型不会对用于组合 D_{composed}的短程序子集过拟合。

定性示例。图4展示了HNPS、HNPS-PT和Naïve-short-finetune生成的程序与真实程序的对比示例。为了评估我们的方法构建适用于程序组合的有效嵌入空间的能力，我们首先介绍两个概念：关键/非关键词元和关键/非关键错误。我们将关键词元定义为这样一类程序词元：如果发生改变，就会改变程序行为，从而针对给定的字符串输入产生不同的程序输出。相应地，程序合成模块在关键词元上出现的错误称为关键错误。相反，非关键词元是指可以替换而不影响程序行为的词元；非关键错误则是指在非关键词元上的错误。在图中，我们用橙色标出所有非关键错误，用红色标出所有关键错误。
在 HNPS 生成的程序中，有 3 个词元与真实程序不同，它们都将“结束”词元与“起始位置”词元混淆了。然而，即使存在这些错误词元，HNPS 生成的程序仍能正确运行。这是因为 DSL 语法规定，只捕获一个字符串令牌的“正则表达式”表达式，无论指定“起始位置”还是“结束”，都会指向相同的索引位置。因此，即使有三个词元错误，HNPS 的输出仍能满足输入/输出规范。

HNPS-PT 的生成程序有 4 个词元错误，只比 HNPS 多一个错误。然而，HNPS-PT 的 4 个错误中有 2 个出现在“正则表达式”函数的关键参数中，彻底改变了该函数的输出。这个例子对应于我们在生成的程序中观察到的一种趋势：与 HNPS-PT 相比，HNPS 往往会出现相同数量或更多的词元错误，但 HNPS

在关键词元上出现的错误较少，因此执行准确率更高。我们推测，这是因为缺少程序嵌入损失时，HNPS-PT 在关键词元上的监督较弱，导致出现更多关键错误。

对于 Naïve-short-finetune，生成程序的正则表达式模式参数存在若干错误，甚至使用了错误的函数。这表明，尽管 Naïve-short-finetune 在相同的数据集上进行训练，其性能仍受逐词元预测方案的限制，因而无法识别正确的程序行为并合成准确的程序。

5.3.2 分析任务嵌入空间

现在，我们分析学习任务嵌入空间如何助力分层程序合成。在图5中，我们展示了从 D_{short} 测试集中随机抽取的 5 个程序的任务嵌入可视化结果。我们使用主成分分析（PCA）将任务嵌入投影到二维空间，从而实现降维。图中的每个点代表一个输入/输出对的嵌入，不同颜色则代表不同的程序。如图所示，来自同一程序的输入/输出对聚集在一起，而来自不同程序的输入/输出对彼此可分，这表明我们能够学习到有意义的嵌入空间，并在训练流程中加以利用。

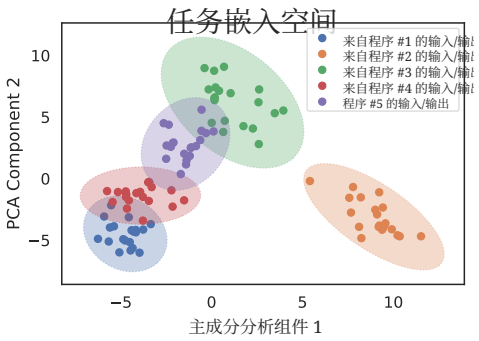


图5：我们对从 D_{short} 测试集中随机抽取的 5 个程序的任务嵌入进行主成分分析降维。每个点代表一个输入/输出对的嵌入，每种颜色代表一个不同的程序。

我们还关注嵌入空间是否能够识别“关键词元”，即替换后会显著改变程序行为的词元。如果只有细微关键变化、几乎完全相同的程序在嵌入空间中的距离更远，那么我们的程序组合器就能更好地抵御影响标准神经合成方法的关键令牌问题。为此，我们从 D_{short} 中抽取了 100 个程序。对于每个抽取的程序 ρ，我们要么更改一个非关键令牌，生成 $\rho_{\text{non-critical}}$，要么更改一个关键令牌，生成 ρ_{critical}。为了获取 $\rho_{\text{non-critical}}$ 和 ρ_{critical} 的任务嵌入，我们为每个程序生成 20 个输入/输出对，并将它们输入任务编码器 e_β。表 2 显示，关键令牌不同的程序之间的嵌入距离，显著大于仅有一个非关键令牌不同的程序之间的距离。因此，我们确认嵌入空间能够捕捉程序语义：与非关键令牌的变化相比，关键令牌的变化对程序行为影响更大，相应地，嵌入空间中 ρ_{critical} 的嵌入与 ρ 的距离也比 $\rho_{\text{non-critical}}$ 的嵌入更远。我们推测，这种平滑的嵌入空间使程序组合器能够更轻松地合成符合给定任务规范的程序。

5.3.3 消融程序嵌入损失

现在，我们具体分析程序嵌入损失的影响。在表 3 中，我们比较了完整方法 HNPS 与不包含程序嵌入损失$\mathcal{L}^{\text{PE}}$的方法 HNPS-PT，在 D_{long} 上的表现，并考察其与组合程序数据集规模的关系。我们在以下数据集上评估这两种方法：（1）包含 200,000 个程序的完整 D_{composed}，（2）D_{composed}的 50%，以及（3）D_{composed}的 25%。在使用完整 D_{composed}进行训练时，只有当程序长度超过 40 个词元时，HNPS 的表现才优于 HNPS-PT。然而，当使用 50% 或 25% 的数据集上，HNPS 的性能在所有程序长度下都显著优于 HNPS-PT。尤其是在 10 万数据集上，对于长度超过 55 个词元的程序，HNPS 的执行准确率为 0.16%，是 HNPS-PT 性能的 8 倍。这表明，来自

表 2：**词元变化时的嵌入距离**。我们从 D_{short} 的训练集和测试集中各抽取了 100 个程序，用于比较两类程序的任务嵌入之间的平均欧几里得距离：一类程序仅在一个关键词元上存在差异，另一类程序仅在一个非关键令牌上存在差异。

差异类型	训练	test
非关键令牌	2.92 ± 0.58	2.99 ± 0.63
关键词元	6.71 ± 1.98	10.50 ± 1.79

程序嵌入损失的监督在数据稀缺时尤其有效。我们推测，这是因为在实际应用中，由于任务编码器并不完美，真实任务嵌入往往带有噪声。当训练数据为程序空间提供了良好的支撑时，模型仅凭程序词元损失就能学会组合短程序，因此噪声较大的程序嵌入损失作为监督的作用较小。然而，当训练数据稀缺时，仅从词元损失中学习会很困难，模型需要程序嵌入损失提供的信息，才能在嵌入损失带有噪声的情况下输出合理的任务嵌入。

表 3：**数据集大小对程序嵌入损失的影响**。我们在以下数据集上评估 HNPS 和 HNPS-PT：(1) 包含 200,000 个程序的完整 D_{composed}，(2) D_{composed} 的 50%（10万），以及 (3) D_{composed} 的 25%（50k）。我们报告它们在 D_{long} 上的执行准确率。

		程序词元数量			
数据集大小	**方法**	**10-25**	**25-40**	**40-55**	**55-70**
200k	HNPS	64.25%	32.22%	**9.71%**	**0.87%**
	HNPS-PT	**65.76%**	**32.49%**	9.4%	0.78%
100k	HNPS	**55.20%**	**20.06%**	**2.32%**	**0.05%**
	HNPS-PT	50.5%	14.55%	0.67%	0.01%
50k	HNPS	**42.23%**	**7.71%**	**0.12%**	0.00%
	HNPS-PT	40.72%	4.5%	0.01%	0.00%

6 讨论

在这项工作中，我们研究如何从任务规范（例如输入/输出对）中学习合成程序。具体而言，我们关注如何扩展当前的神经程序合成方法，以合成具有更复杂行为的长程序。为此，我们提出了分层神经程序合成（HNPS）框架；不同于大多数现有方法逐个生成程序词元，我们提出逐个子程序地合成程序。具体来说，我们首先从短程序及其输入/输出对中学习任务嵌入空间，以连续地参数化多样的程序行为。随后，我们构建一个组合程序数据集，为学习程序组合器提供中间监督，使其能够高效地以分层方式组合短程序，从而形成用于解决任务的长而复杂的程序。在字符串转换领域进行的实验结果表明，我们提出的框架有效。消融研究详细分析了学到的任务嵌入空间，并进一步验证了所提出的训练方案，该方案利用了组合程序数据集。

参考文献

Daniel A Abolafia、Mohammad Norouzi、Jonathan Shen、Rui Zhao 和 Quoc V Le。《采用优先队列训练的神经程序合成》。*arXiv* 预印本 *arXiv:1801.03526*，2018年。

Matej Balog、Alexander L Gaunt、Marc Brockschmidt、Sebastian Nowozin 和 Daniel Tarlow。Deepcoder：学习编写程序。发表于国际学习表征会议，2017年。

Matko Bošnjak、Tim Rocktäschel、Jason Naradowsky 和 Sebastian Riedel。使用可微 Forth 解释器进行编程。发表于国际机器学习会议，2017年。

Rudy R Bunel、Matthew Hausknecht、Jacob Devlin、Rishabh Singh 和 Pushmeet Kohli。利用语法和强化学习进行神经程序合成。发表于国际学习表征会议，2018年。

Michael Burke、Svetlin Penkov 和 SubramanianRamamoorthy。从解释到合成：用于从示范中学习的组合式程序归纳。*arXiv* 预印本 *arXiv:1902.10657*，2019年。

Jonathon Cai、Richard Shin 和 Dawn Song。通过递归使神经编程架构实现泛化。《国际学习表征会议》，2017年。

Mark Chen、Jerry Tworek、Heewoo Jun、Qiming Yuan、Henrique Ponde de Oliveira Pinto、Jared Kaplan、Harri Edwards、Yuri Burda、Nicholas Joseph、Greg Brockman、Alex Ray、Raul Puri、Gretchen Krueger、Michael Petrov、Heidy Khlaaf、Girish Sastry、Pamela Mishkin、Brooke Chan、Scott Gray、Nick Ryder、Mikhail Pavlov、Alethea Power、Lukasz Kaiser、Mohammad Bavarian、Clemens Winter、Philippe Tillet、Felipe Petroski Such、Dave Cummings、Matthias Plappert、Fotios Chantzis、Elizabeth Barnes、Ariel Herbert-Voss、William Hebgen Guss、Alex Nichol、Alex Paino、Nikolas Tezak、Jie Tang、Igor Babuschkin、Suchir Balaji、Shantanu Jain、William Saunders、Christopher Hesse、Andrew N. Carr、Jan Leike、Josh Achiam、Vedant Misra、Evan Morikawa、Alec Radford、Matthew Knight、Miles Brundage、Mira Murati、Katie Mayer、Peter Welinder、Bob McGrew、Dario Amodei、Sam McCandlish、Ilya Sutskever 和 Wojciech Zaremba。评估基于代码训练的大语言模型。《arXiv 预印本 arXiv:2107.03374》，2021年。

Xinyun Chen、Chang Liu 和 Dawn Song。执行引导的神经程序合成。载于《国际学习表征会议》，2019年。

Jacob Devlin、Rudy R Bunel、Rishabh Singh、Matthew Hausknecht 和 Pushmeet Kohli。神经程序元归纳。载于《神经信息处理系统进展》，2017a。

Jacob Devlin、Jonathan Uesato、Surya Bhupatiraju、Rishabh Singh、Abdel-rahman Mohamed 和 Pushmeet Kohli。Robustfill：噪声输入 / 输出下的神经程序学习。载于《国际机器学习会议》，2017b。

Kevin Ellis、Maxwell Nye、Yewen Pu、Felix Sosa、Josh Tenenbaum 和 Armando Solar-Lezama。编写、执行、评估：使用 REPL 进行程序合成。载于《神经信息处理系统》，2019年。

Kevin Ellis、Catherine Wong、Maxwell Nye、Mathias Sable-Meyer、Luc Cary、Lucas Morales、Luke Hewitt、Armando Solar-Lezama 和 Joshua B Tenenbaum。Dreamcoder：通过清醒-睡眠贝叶斯程序学习，构建通用且可解释的知识。*arXiv* 预印本 *arXiv:2006.08381*，2020年。

Alexander L. Gaunt、Marc Brockschmidt、Nate Kushman 和 Daniel Tarlow。结合神经库的可微程序。载于国际机器学习会议（*ICML*）论文集，2017年。

Alex Graves、Greg Wayne 和 Ivo Danihelka。神经图灵机。*arXiv* 预印本 *arXiv:1410.5401*，2014年。

Joey Hong、David Dohan、Rishabh Singh、Charles Sutton 和 Manzil Zaheer。潜在程序员：用于程序合成的离散潜在代码。载于国际机器学习会议，2021年。

Łukasz Kaiser 和 Ilya Sutskever。神经图形处理器学习算法。载于国际学习表征会议，2016年。

Miguel Lázaro-Gredilla、Dianhuan Lin、J Swaroop Guntupalli 和 Dileep George。超越模仿：通过将概念作为认知程序来学习，实现机器人上的零样本任务迁移。《科学·机器人学》，2019年。

Yujia Li、David Choi、Junyoung Chung、Nate Kushman、Julian Schrittwieser、Rémi Leblond、Tom Eccles、James Keeling、Felix Gimeno、Agustin Dal Lago 等。使用 AlphaCode 实现竞赛级代码生成。*arXiv* 预印本 *arXiv:2203.07814*，2022年。

Yuan-Hong Liao、Xavier Puig、Marko Boben、Antonio Torralba 和 Sanja Fidler。通过活动草图合成环境感知活动。发表于*IEEE*计算机视觉与模式识别会议，2019年。

Xi Victoria Lin、Chenglong Wang、Luke Zettlemoyer 和 Michael D Ernst。NL2Bash：面向Linux操作系统自然语言接口的语料库和语义解析器。发表于国际语言资源与评测会议，2018年。

Paweł Liskowski、Krzysztof Krawiec、Nihat Engin Toklu 和 Jerry Swan。《程序合成作为潜在连续优化：神经嵌入中的进化搜索》。发表于遗传与进化计算会议，2020年。

Guan-Ting Liu、En-Pei Hu、Pu-Jen Cheng、Hung-Yi Lee 和 Shao-Hua Sun。《通过学习组合程序实现分层程序化强化学习》。《style id='1'>arXiv 预印本 arXiv:2301.12950，2023年。

Yunchao Liu、Jiajun Wu、Zheng Wu、Daniel Ritchie、William T. Freeman 和 Joshua B. Tenenbaum。《学习用程序描述场景》。发表于国际学习表征会议，2019年。

Arvind Neelakantan、Quoc V Le 和 Ilya Sutskever。《神经程序员：通过梯度下降诱导潜在程序》。发表于国际学习表征会议，2015年。

Erik Nijkamp、Bo Pang、Hiroaki Hayashi、Lifu Tu、Huan Wang、Yingbo Zhou、Silvio Savarese 和 Caiming Xiong。用于程序合成的对话范式。*arXiv* 预印本，2022年。

Augustus Odena、Kensen Shi、David Bieber、Rishabh Singh、Charles Sutton 和 Hanjun Dai。BUSTLE：通过学习引导的探索实现自底向上程序合成。发表于国际学习表征会议，2021年。

Emilio Parisotto、Abdel-rahman Mohamed、Rishabh Singh、Lihong Li、Dengyong Zhou 和 Pushmeet Kohli。神经符号程序合成。发表于国际学习表征会议，2017年。

Thomas Pierrot、Guillaume Ligner、Scott E Reed、Olivier Sigaud、Nicolas Perrin、Alexandre Laterre、David Kas、Karim Beguir 和 Nando de Freitas。通过递归树搜索和规划学习组合式神经程序。发表于神经信息处理系统，2019年。

Scott Reed 和 Nando De Freitas。神经程序员解释器。发表于国际学习表征会议，2016年。

Kensen Shi、Joey Hong、Manzil Zaheer、Pengcheng Yin 和 Charles Sutton。神经程序合成中的组合泛化与分解。发表于*ICLR* 代码深度学习（*DL4C*）研讨会，2022年。

Eui Chul Shin、Illia Polosukhin 和 Dawn Song。利用推断出的执行轨迹改进神经程序合成。发表于神经信息处理系统，2018年。

Tom Silver、Kelsey R Allen、Alex K Lew、Leslie Pack Kaelbling 和 Josh Tenenbaum。基于逻辑程序策略的少样本贝叶斯模仿学习。发表于人工智能促进协会，2020年。

Shao-Hua Sun、Hyeonwoo Noh、Sriram Somasundaram 和 Joseph Lim。通过多样化演示视频进行神经程序合成。发表于国际机器学习会议，2018年。

Shao-Hua Sun、Te-Lin Wu 和 Joseph J. Lim。程序引导智能体。载于国际学习表征会议，2020年。

Dweep Trivedi、Jesse Zhang、Shao-Hua Sun 和 Joseph J Lim。学习合成程序，将其作为可解释且可泛化的策略。载于神经信息处理系统进展，2021年。

Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N. Gomez、Lukasz Kaiser 和 Illia Polosukhin。注意力就是一切。载于神经信息处理系统会议论文集（*NeurIPS*），2017年。

Abhinav Verma、Vijayaraghavan Murali、Rishabh Singh、Pushmeet Kohli 和 Swarat Chaudhuri。可编程解释的强化学习。载于国际机器学习会议，2018年。

Jiajun Wu、Joshua B Tenenbaum 和 Pushmeet Kohli。神经场景反渲染。载于*IEEE*计算机视觉与模式识别会议，2017年。

Da Xiao、Jo-Yu Liao 和 Xingyuan Yuan。通过组合子抽象提升神经程序员解释器的普适性和可学习性。发表于国际学习表征会议，2018年。

Danfei Xu、Suraj Nair、Yuke Zhu、Julian Gao、Animesh Garg、Li Fei-Fei 和 Silvio Savarese。神经任务编程：学习在层级任务之间进行泛化。发表于国际机器人与自动化会议，2018年。

Wojciech Zaremba、Tomas Mikolov、Armand Joulin 和 Rob Fergus。从示例中学习简单算法。发表于国际机器学习会议，2016年。

附录

目录

A 架构 **14**

A. 1 任务编码器.14 A.2 程序解码器.14 A.3 程序组合器.14

B 超参数与训练细节 15

B.1 学习程序嵌入空间 .
15 B.2 分层神经程序合成.15 B.3 基线实现细节.15 B.4 超参数.15 B.5 语法检查器.
16

C 程序数据集 16

C.1 程序与规范生成细节 .
16 C.2 数据集统计.17 C.3 程序与输入输出对样例.17

D 扩展结果 19

D.1 生成的程序 .
19 D.2 对 D_{composed} 的评估.20

A 架构

A.1 任务编码器

任务编码器 e_β 是一个循环神经网络，用于将输入/输出任务规范 σ 编码为任务嵌入。随后，程序组合器 h_ω 使用该编码表示来合成程序。具体而言，该编码器是一个门控循环单元（GRU）模块，输入特征维度为 256，隐藏状态维度为 512。分别为输入字符串和输出字符串计算嵌入向量。这些嵌入会传入联合编码模块，该模块由一组线性层、比率为 0.2 的随机失活和线性整流单元（ReLU）激活函数组成，随后进行批归一化。

A.2 程序解码器

程序解码器 p_θ 首先在嵌入空间学习阶段进行训练，它是一个循环神经网络，用于将任务编码器 e_β 生成的任务嵌入解码为程序 $\hat{}\rho$。训练该模型的目标是使重构程序尽可能接近真实程序，而该真实程序的任务规范正是任务编码器的输入。在 HNPS 中，程序解码器接收来自程序组合器模块的程序嵌入作为输入，并输出程序词元。从内部结构来看，解码器是一个门控循环单元（GRU）模块，输入特征维度为 343，隐藏状态维度为 256。先对先前生成的程序词元进行编码，并与程序组合器模块的输出拼接，再输入 GRU；随后通过两个线性层和双曲正切激活函数生成输出 logits。最后，对这些 logits 进行 softmax 运算，以生成程序词元。

A.3 程序组合器

程序组合器 h_ω 是一个循环神经网络，以任务解释器的输出作为输入，并依次生成任务嵌入。具体来说，在每个步骤中，h_ω 将先前生成的任务嵌入和程序解码器的最后一个隐藏状态作为输入，以预测下一个任务

嵌入。它由一个 GRU 模块组成，输入特征维度为 768，隐藏状态维度为 256。随后接一个由线性层和带泄漏的修正线性单元激活组成的单块结构。

B 超参数与训练细节

对于用于训练模型的数据集（D_{short}、 D_{composed}），我们采用 70%/15%/15% 的训练/验证/测试集划分。每个程序的数据集包含 20 组输入/输出字符串对。训练期间，我们会从这 20 组输入/输出对中随机抽取 10 组作为任务规范。
我们使用 PyTorch 框架实现 HNPS。

B.1 学习程序嵌入空间

在此阶段，我们的模型旨在生成一个程序，使其尽可能接近其行为由输入任务规范描述的程序。为此，我们对生成程序词元与真实程序词元使用交叉熵损失。我们采用具有教师强制行为的训练方案，即用真实程序词元替换先前预测的程序词元。

B.2 分层神经程序合成

在这里，除了针对单个程序词元的交叉熵损失外，我们还对程序嵌入使用均方误差损失。在微调期间，我们使用真实程序嵌入来计算程序嵌入损失，但允许嵌入空间发生漂移。尽管真实程序嵌入可能存在噪声，但它们仍然为程序组合提供了宝贵的监督信号。

为了确定合成模块何时应停止输出更多程序词元，我们最初尝试学习停止词元的预测。然而，我们发现这种方法不够稳健。作为替代方案，我们利用了程序合成领域的一项特点：我们可以随时验证程序的执行情况。因此，在组合程序时，如果添加另一个子程序会导致执行准确率下降，我们就会在此时终止程序——我们发现这一启发式方法效果良好。

B.3 基线实现细节

Naïve、朴素短版和 Naïve-short-finetune 均采用 GRU 编码器-解码器模型实现。编码器采用与 HNPS 任务编码器相同的架构和超参数，解码器则采用与 HNPS 程序解码器相同的架构和超参数。

BUSTLE 自底向上搜索基线的运行时阈值为 500,000 个表达式。我们使用的该基线结果来自这一版本的 BUSTLE：它利用字符串启发式来优先进行子表达式搜索。我们尝试了加入神经网络的变体，但发现其在长程序上的执行准确率并未提升，因此这里只报告启发式实现的结果。

B.4 超参数

训练使用 Adam 优化器，初始学习率为 0.0001，并通过学习率调度器以 0.95 的伽马值衰减学习率。权重衰减设为 0.00001。模型在该数据集上训练 150 轮。

我们实验中使用的超参数如下：

- 任务规范中的输入/输出字符串对数量：10
- 优化器：Adam
- 学习率：0.0001
- 学习率调度器伽马值：0.95
- 权重衰减：0.00001

- 训练轮数：150

任务编码器：

- 模型类型：门控循环单元
- 模型类型：门控循环单元
- 输入维度：256
- 隐藏状态维度：512
- 随机失活：0.2
- 激活函数：ReLU
- 线性层块数量：1
- 输入/输出嵌入池化函数：均值

程序解码器：

- 模型类型：门控循环单元
- 输入维度：256
- 隐藏状态维度：256
- 激活函数：双曲正切
- 线性层数量：2

程序组合器：

- 模型类型：门控循环单元
- 输入维度：768
- 隐藏状态维度：256
- 激活函数：带泄漏的修正线性单元
- 线性层块数量：1

HNPS 损失系数：

- λ：1.0

B.5 语法检查器

在我们的实验中，对于所有方法，我们都使用语法检查器来确保合成模块按照领域特定语言输出有效的程序词元。这有助于生成能够实际编译和执行的程序。例如，如果生成了一个左括号词元，语法检查器会确保括号中的表达式最终以右括号词元闭合。语法检查器的另一个用途是在合成过程中确保正则表达式格式正确（*i.e.*，正则表达式函数类型必须是以下之一：单词、数字、字母数字、全大写、首字母大写、小写、数字、字符。

C 程序数据集

训练好嵌入空间后，我们使用训练好的任务编码器获取每个短程序的嵌入。这些短程序组合在一起构成数据集 D_{composed}。我们将短程序嵌入与这些短程序组合后形成的程序一并存储，以便计算程序嵌入损失 $\mathcal{L}^{\text{PE}}$。

C.1 程序和规范生成细节

在生成过程中，我们使用图 1 中提供的领域特定语言语法递归地采样每个程序。具体而言，我们从一个含有空位的程序草图开始，拼接 c(e c)，然后根据语法规则将空位 e 转换为常量字符串 k(s k)、子字符串 s(q_1 q_2 s) 或两个表达式e_1e_2。此过程递归重复，直到程序中不再有空位。生成过程中各组件所用的概率列于表 4。

给定一个生成程序，我们使用概率采样器来生成候选输入字符串。为降低程序执行时出现异常的概率，我们采用一套简单的启发式方法来生成输入字符串。例如，如果程序包含正则表达式 r(数字 3 起始位置 r)，用于计算

表5：我们各个数据集中重要领域特定语言标记的频率。我们通过将每个词元的总出现次数除以数据集中的程序数量来计算标记频率。

词元	D_{short}	D_{composed}	D_{long}
拼接	1.00	1.00	1.00
常量字符串	0.06	0.23	0.11
子字符串	0.94	2.82	2.89
正则表达式	1.64	4.92	5.21
常量位置	0.23	0.72	0.57
Word	0.09	0.25	0.32
Num	0.04	0.11	0.34
字母数字	0.09	0.25	0.32
全大写	0.08	0.25	0.33
首字母大写	0.09	0.26	0.33
小写	0.09	0.25	0.34
数字	0.09	0.26	0.32
Char	0.06	0.21	0.31
所有数字词元	1.87	5.66	5.78
所有字符词元	1.08	3.31	2.71
起始位置	0.82	2.46	2.60
End	0.82	2.46	2.61

输入字符串中第三个数字的位置，我们会限制采样器，使生成的输入字符串至少包含 3 个数字。

C.2 数据集统计

D_{short} 包含 100,000 个程序，这些程序是通过从我们的领域特定语言中随机采样长度为一个表达式的程序创建的。对于每个采样得到的程序，我们随机生成 1,000 个包含 50–70 个字符的字符串作为潜在输入字符串，并将其中能够无异常执行的前 20 个输入字符串及其对应的输出字符串存入数据集。如果某个程序对应的 1,000 个生成字符串中，不到 20 个输入字符串能够无异常执行，则舍弃该程序。对于 D_{composed}，我们还通过组合从 D_{short} 中采样得到的程序，生成 200,000 个程序。在 D_{composed}中，60,000 个程序由 2 个短程序组合而成，70,000 个程序由 3 个短程序组合而成，另有 70,000 个程序由 4 个短程序组合而成。对于 D_{composed}中的每个组合程序，均采用相同的方法生成输入/输出对。

表 4：生成每个程序时，抽取各组件以填充空位的概率。我们采用递归生成过程，用 DSL 语法规则指定的组件填充当前程序中的空位。

空位类型	**组件**	**概率**
表达式 e	常量字符串 k(sk)	0.1
	子字符串 s(q_1 q_2 s)	0.35
	e_1 e_2	0.55
位置 q	常量位置 p(k p)	0.1
	正则表达式 r(x n b r)	0.35
边界 b	起始位置，结束	0.5
正则表达式 x	单词、数字.....字符	0.0625
位置索引 k	-20, -19, ..., 20	0.0244
位置索引 k	-3, -2, ..., -3	0.143
字符 k	“空格” , ".", ..."? "	0.05

我们生成了一个长程序数据集 D_{long}，其中包含 30,000 个程序，每个程序都由.....组成通过直接从我们的领域特定语言中采样来训练 2–4 个表达式。D_{long} 包含 10,000 个由 2 个表达式组成的程序、10,000 个由 3 个表达式组成的程序，以及 10,000 个由 4 个表达式组成的程序。

为了展示这 3 个数据集之间的分布差异，我们在表5中列出了每个数据集中重要领域特定语言标记的频率。我们通过将每个词元的出现总次数除以数据集中的程序数量来计算标记频率。

C.3 程序和输入/输出对样例

我们列出来自 D_{composed} 数据集的程序示例，并展示为每个程序采样的 20 个输入/输出对中的 5 个。

输入：ZL%@r#ENc#HMONNRZLEBHVZMYIDUYBZJWQPOQLVLXGLMREUJDXQI’aP@Q(dR(#/E@7#//
输出：@7ZL%@r#ENc#HMONNRZLEBHVZMYIDUYBZJWQPOQLVLXGLMREUJDXQI’aP
@Q(dR(#/E@7#/

输入：##/@@@KA?/WYYESBKMODJKGDBBYWPVXFQINPRGVGYOHRQWDEQZIOCYVGPJh/#
输出：@KA?/WYYESBKMODJKGDBBYWPVXFQINPRGVGYOHRQWDEQZIOCYVGPJh/KA?/W
YYESBKMODJKGDBBYWPVXFQINPRGVGYOHRQWDEQZIOCYVGPJh

输入：?!b:{#M@T@$BRKOCJZARAAHMHPZWAKAUy:YSLDHLLVOPHFMPYTZCFr#/X@///TLa#
输出：@///TLaBRKOCJZARAAHMHPZWAKAUy:YSLDHLLVOPHFMPYTZCFr#/X@//

输入：@/EOJHGAWTNYWAETCPPFNTTECQVPFYSb@#/#@/#EPGDIKWHJMVQNTVM3
输出：@/EOJHGAWTNYWAETCPPFNTTECQVPFYSb@#/#@

输入：MG#i/ZMTEMXUVXDKEMIDAFAUV8@QOOXGRGGFVQTSOZUAURSv@@7#%#kX?/L[W’@z{a#/(
输出：@z{aMG#i/ZMTEMXUVXDKEMIDAFAUV8@QOOXGRGGFVQTSOZUAURSv@@7#
%#kX?/L[W’@z{a#

真实程序：

```
拼接 c( 子字符串 s( 常量位置 p( 7 p) 正则表达式 r( "#" 1 结束 r) s)
子字符串 s( 正则表达式 r( "(" -2 起始位置 r) 正则表达式 r( "." -2 结束 r) s) c)
```

输入：G#PZoez.Xo()$D(AJW3Av#q’@Xi?#P..TlY(O(N&XI#G: Q.hQT#dk)0Y(jSF
输出：.Xo()$D(AJW3Av(N&XI#G: Q.hQT#dk)0Y(j

输入：xN#iiu/HHfa4(!pEDz4{q(,(nlw.{JsGN#SK;h./CjDt}u.n0DgMuj#KBR,H#.(F;SR;R
输出：HHfa4(!pEDz4{q(,(nlw.{JsGN(nlw.{JsGN#SK;h./CjDt}u.n0DgMuj#KBR,H#.(F;SR

输入：ncL9.(Cbz3q[y9CXHkjjI84RA!sRX.T’;%G(kdk.@zb#j(D#X]#.&CHSwcdB([D}4 1
输出：bz3q[y9CXHkjjI84RA!sRX.T’;%G(kdk.@zb#j(D(D#X]#.&CHSwcdB([D}4

输入：78@aQ0(I(:#[h]#)8F3C.9(.5h9UXz#&L(2gZCR8d5iK33.Z.e[8
输出：I(:#[h](.5h9UXz#&L(2gZCR8d5iK33.Z.e

输入：:Jmq?T(R1hD.#;,.(Q9X?q7Dm}1Vq6e’0.J(##C8]s]5.X/a0?mC9(5k
输出：R1hD.#;,.(Q9X?q7Dm}1Vq6e’0.J((##C8]s]5.X/a0?mC9(

真实程序：

```
拼接 c( 子字符串 s( 正则表达式 r( "@" -1 结束 r) 正则表达式 r( "#" -1 结束 r) s)
子字符串 s( 正则表达式 r( 全大写 0 起始位置 r) 正则表达式 r( "/" -1 结束 r) s) c)
```

输入：!&]dr]!rqFpih3CdBEEICIvPaNkuHa4$0lX!{IA&eFkAUtbDBAs5GusY’&&
输出：!&]dr]!rqFpih3CdBEEICIvPaNkuHa4$0lX!{IA&eFkAUtbDBAs5Gus
$]dr]!rqFpih3CdBEEICIvPaNkuHa4$0lX!{IA&eFkAUtbDBAs5GusY’

输入：a}&]&EqZWNGObvBcuKmtUvUzEtupBTYdHjfyMOj3R!f&wkhWgLeLk5!tShuzGfo4&
输出：!f&wkhWgLeLk5!tShuzGf$]&}&EqZWNGObvBcuKmtUvUzEtupBTYdHjf
yMOj3R!f

输入：evFjtgoxhRAUsAoeAbNK4&$m]SfjBenvmUL9]!.MQBWzpFEbIcktLQSgasdlHc#&&Eg&!e
输出：!.MQBWzpFEbIcktLQSgasdlHc#&&E$]SfjBenvmUL9]!.MQBWzpFEbIc
ktLQSgasdlHc#&

输入：f&]LP!rSAlV&INLbdV!&.soP%D@YDxCzkUaC63IgUV7sLsgxzcgnFsbWCz7!D2H DqA&]
输出：!rSAlV&INLbdV!&.soP%D@YDxCzkUaC63IgUV7sLsgxzcgnFsbWCz7!D2
H Dq$]LP!rSAlV&INLbdV!

输入：Z](&SCh2!&(SWbZZjJOD1PC]FlgEAwCAXNZsLbdmJbzFLwd$V!&?4n&qVVQ
输出：!&(SWbZZjJOD1PC]FlgEAwCAXNZsLbdmJbzFLwd$V!&?4n&qVV$](&
SCh2!&(SWbZZjJOD1PC]FlgEAwCAXNZsLbdmJbzFLwd$V!

真实程序：

```
拼接 c( 子字符串 s( 正则表达式 r( "!" 0 起始位置 r) 正则表达式 r( 单词 -1 结束 r) s)
常量字符串 k( "$" k)
子字符串 s( 正则表达式 r( "]" 0 结束 r) 正则表达式 r( "&" 2 结束 r) s) c)
```

输入: ;(qx{Y„,FkAUtb}))D))DCAt1W,(2,S;BAspoCdBEEICIvPaNkuHajGdqFpi4;:F
输出: (qx{Y„))DCAFkAUtb}))D))DCAt1W,(2,S&

输入: VI,r#;zGfo6,q);/X(3„roqpvaDzj0,R;Z')(&yK;ZDEq3)OSus$xGc)AqskKS7BX{lh
输出: (3)OSus$xGc)AqskKS7BzGfo6,q)&

输入:)S),);NpQFmcNGTMMQB4bbXkWitgy0,;(S,o;,)LmjUGnwzcm7,(IPpqrv10
输出: (S);NpQFmcNGTMMQB4bbXkWitgy0,;(S,o;,)LmjUGnwzcQFmcNGTMMQB4bb XkWitgy0,&

输入: HW(fMhJyyV0)T)ex,),iLiRno'E;;,),)yZXAIbscuV6(vUHgjyB7,;o
输出: (fMhJyyV0)T)ex,),iLiRno'E;;),)yZXAIbscuMhJyyV0)T)ex,),iLiRno 'E;&

输入:)zRxForqNOFp7FL,4))2;ksgGpWkZRRdGMvjxaT2((„;1;4)jYeft9,FYrrzBY7,b
输出: ((,)2;ksgGpWkZRRdGMvjxaT2((„;1;4)jYefForqNOFp7FL,4))2;ksgGp WkZRRdGMvjxaT2((„&

真实程序: 拼接 c(子字符串 s(正则表达式 r("(" 0 起始位置 r) 正则表达式 r("," -3 起始位置 r) s) 子字符串 s(正则表达式 r(")" -2 起始位置 r) 正则表达式 r(单词 -2 结束 r) s) 子字符串 s(正则表达式 r(字符 3 结束 r) 正则表达式 r(";" 1 结束 r) s) 常量字符串 k("&" k) c)

输入: 2q@?QK6IAd}t2?))}!}!l?AV}UP,Hnriz1OoH?!CsyjyICX!!!Esi$80)DU5i}}
输出: @?QK6IAd}t2?))}!}!l?AV}UP,Hnriz1OoH?!CsyjyICX))}!}!l?A V}Ut2?))}!}!l?AV?!

输入: !WHj!?XUqQT }}yOI)x1[SWU'!KoVWd.RoutoK}!?}?Rocescz7!lWBxhk$OXu)}a))
输出: Hj!?XUqQT }}yOI)x1[SWU'!KoVWd.RoutoK})x1[SWU'!KoVyOI)x1[S WU'!KoVWd.RoutoK?Rocescz7!lW

输入: Xcxu6EQ1!Bx(}}1KMx?!}8U@}?!!?M)XpoeCWokeC!Lbiai.EQ7un}BH!})zy{QA(
输出: xu6EQ1!Bx(}}1KMx?!}8U@}?!)XpoeCWokeC!Lbiai.E1KMx?!}8U@? M)XpoeC

输入: EAb?u6!)DK]!Cp#?mFo$IR&cC/})Y!3Imegp3SW@CV(!}n}?MseO}Jbdt3W!Tr7}BKp
输出: b?u6!)DK]!Cp#?mFo$IR&cC/})Y!3Imegp3SW@CV()DK]!Cp#?mFo$ IR&cC/})Y!3Imegp3SDK]!Cp#?mFo$IR&cC/})Y!3Imegp3SW@CV(!}n?MseO}Jbdt3W!

输入: CR@Vo,!?!))!NL'PBoSZ1}!ToKOOoja$TphjC!}BjhfwkJJO(?q}?}DrpRtNN2}
输出: @Vo,!?!))!NL'PBoSZ1}))!NL'PBoSZ1}!ToKONL'PBoSZ1}!ToKOOoja $TphjC!}BjhfwkJJO(?q?}

真实程序: 拼接 c(子字符串 s(常量位置 p(2 p) 正则表达式 r("!" 3 起始位置 r) s) 子字符串 s(正则表达式 r(")" 0 起始位置 r) 正则表达式 r(全大写 -3 结束 r) s) 子字符串 s(正则表达式 r(字母数字 2 起始位置 r) 正则表达式 r("}" -3 结束 r) s) 子字符串 s(正则表达式 r("?" -1 起始位置 r) 正则表达式 r(首字母大写 -2 起始位置 r) s) c)

D 扩展结果

D.1 生成的程序

我们展示了由 HNPS 模型生成的程序示例。每个程序都是根据从 D_{composed} 中随机选取的任务规范生成的。我们将每个生成程序与数据集给出的真实程序进行比较。

真实值程序: 拼接 c(子字符串 s(正则表达式 r(字母数字 -3 起始位置 r) 正则表达式 r("(" -2 起始位置 r) s) 子字符串 s(正则表达式 r(字母数字 -1 结束 r) 正则表达式 r("." -1 起始位置 r) s) c)

HNPS 生成的程序: 拼接 c(子字符串 s(正则表达式 r(字母数字 -3 起始位置 r) 正则表达式 r("(" -2 结束 r) s) 子字符串 s(正则表达式 r(字母数字 -1 结束 r) 正则表达式 r("." -1 起始位置 r) s) c)

真实程序: 拼接 c(子字符串 s(正则表达式 r("!" 0 起始位置 r) 正则表达式 r("$" -1 结束 r) s) 子字符串 s(常量位置 p(12 p) 正则表达式 r("." -1 结束 r) s) c)

HNPS 生成的程序: 拼接 c(子字符串 s(正则表达式 r("!" 0 结束 r) 正则表达式 r("." -1 结束 r) s) 子字符串 s(常量位置 p(12 p) 正则表达式 r("$" -1 结束 r) c)

真实程序:	拼接 c(子字符串 s(正则表达式 r(首字母大写 -3 结束 r) 正则表达式 r("[" -2 起始位置 r) s) 子字符串 s(正则表达式 r("." 1 起始位置 r) 正则表达式 r(首字母大写 1 起始位置 r) s) 子字符串 s(正则表达式 r("@" -3 结束 r) 常量位置 p(-8 p)) c)
HNPS 生成的程序:	拼接 c(子字符串 s(正则表达式 r(首字母大写 -3 结束 r) 正则表达式 r("[" -2 结束 r) s) 子字符串 s(正则表达式 r("." 1 结束 r) 正则表达式 r(首字母大写 1 起始位置 r) s) 子字符串 s(正则表达式 r("@" -3 起始位置 r) 常量位置 p(-8 p) c)

真实值程序:	拼接 c(子字符串 s(正则表达式 r("SPACE" -1 结束 r) 正则表达式 r("{" -2 结束 r) s) 子字符串 s(正则表达式 r("?" 2 结束 r) 正则表达式 r(小写 2 结束 r) s) 子字符串 s(正则表达式 r(全大写 3 起始位置 r) 正则表达式 r("," -1 起始位置 r) s) c)
HNPS 生成的程序:	拼接 c(子字符串 s(正则表达式 r("空格" -1 起始位置 r) 正则表达式 r("{" -2 结束 r) s) 子字符串 s(正则表达式 r("?" -2 起始位置 r) 正则表达式 r(小写 -1 结束 r) s) 子字符串 s(正则表达式 r(全大写 3 起始位置 r) 正则表达式 r("," -1 起始位置 r) s) c)

真实值程序:	拼接 c(子字符串 s(常量位置 p(14 p) 正则表达式 r("$" 0 结束 r) s) 子字符串 s(正则表达式 r("" -2 起始位置 r) 正则表达式 r(数字 -2 起始位置 r) s) 子字符串 s(正则表达式 r(数字 0 结束 r) 正则表达式 r(单词 3 结束 r) s) 子字符串 s(正则表达式 r("!" 2 起始位置 r) 正则表达式 r(字符 -3 起始位置 r) s c)
HNPS 生成的程序:	拼接 c(子字符串 s(常量位置 p(14 p) 正则表达式 r(字符 -1 结束 r) s) 子字符串 s(正则表达式 r("" 2 结束 r) 正则表达式 r(数字 -2 起始位置 r) s) 子字符串 s(正则表达式 r(字符 3 结束 r) 正则表达式 r(单词 3 结束 r) s) 子字符串 s(正则表达式 r("!" 2 起始位置 r) 常量位置 p(-4 p) c)

拼接 c(常量字符串 k("空格" k)	子字符串 s(正则表达式 r("." 2 结束 r) 正则表达式 r("?" 2 结束 r) s) 子字符串 s(正则表达式 r("@" -2 结束 r) 正则表达式 r(字母数字 3 结束 r) s) 子字符串 s(正则表达式 r(")" 1 起始位置 r) 正则表达式 r(数字 -1 起始位置 r) s) s) c)
HNPS 生成的程序:	拼接 c(常量字符串 k("空格" k) 子字符串 s(正则表达式 r("." 2 结束 r) 正则表达式 r("?" 2 结束 r) s) 子字符串 s(正则表达式 r("@" -2 结束 r) 正则表达式 r(字母数字 3 结束 r) s) 子字符串 s(正则表达式 r(")" 1 结束 r) 正则表达式 r(数字 -1 起始位置 r) s) c)

D.2 在 D_{COMPOSED}上的评估

我们还在自己的 $D_{composed}$ 数据集上评估了每种方法的执行准确率。在表6中，我们根据程序的词元数量对程序进行分组。在表7中，我们根据用于组合程序的短程序数量对程序进行分类。从结果中可以看出，HNPS 和 HNPS-PT 的性能优于 H-朴素-PT。随着程序变长，它们与 H-朴素-PT 之间的性能差距也随之增大。对于由 2 个短程序组合而成的长程序，H-朴素-PT 的性能达到 HNPS 的 93%；但对于由 4 个短程序组合而成的长程序，其执行准确率仅为 HNPS 的 60%。这凸显了任务嵌入空间对于采用层次化架构进行程序合成的重要性。我们还观察到，在训练集和测试集程序上，非层次化基线方法（Naïve-*）的性能都显著低于层次化方法。对于由 4 个短程序组合而成的程序，即使是在训练集上，所有层次化变体的执行准确率都超过了 10%，而非层次化基线方法的执行准确率均未达到 2%。这支持了我们的论断：缺少层次化架构会从根本上限制模型拟合长程序分布的能力。

在表 8 中，我们还展示了 D_{composed}上训练集与测试集性能之间的差异，即“泛化差距”。HNPS 的泛化差距小于 HNPS-PT 和 H-朴素-PT。比较 HNPS 与 HNPS-PT 在长度超过 40 个词元的程序上的表现时，这一差异尤其明显：尽管 HNPS-PT 在训练集上取得了更高的执行准确率，但在测试集上的表现却更差。这表明，程序嵌入损失有助于防止层次化模型过拟合。

Table 6：各方法在 D_{composed}的训练集和测试集上的执行准确率。程序按 e sepa词元数量（长度）分组。

方法	训练				Test			
	10-25	25-40	40-55	55-70	10-25	25-40	40-55	55-70
Naïve	54.55%	21.73%	2.22%	0.05%	50.08%	17.8%	1.64%	0.01%
朴素短版	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Naïve-short-finetune	56.05%	28.37%	6.10%	0.34%	53.82%	24.21%	3.87%	0.18%
H-朴素-PT	77.82%	51.62%	26.62%	8.39%	69.54%	37.48%	11.35%	1.04%
HNPS-PT	**82.50%**	**59.84%**	**33.09%**	**11.72%**	**74.05**%	**45.04%**	15.59%	1.80%
HNPS（我们的完整版本）	80.74%	56.75%	30.00%	9.66%	73.36%	43.88%	**15.70%**	**2.30%**

表7：各方法在 D_{composed}的训练集和测试集上的执行准确率。程序按照组成它们所用的子程序数量分组。

方法	训练			Test		
	2	3	4	2	3	4
Naïve	27.30%	5.83%	0.60%	22.99%	4.27%	0.51%
朴素短版	0.02%	0.00%	0.00%	0.00%	0.00%	0.00%
Naïve-short-finetune	34.66%	10.06%	1.08%	30.10%	7.16%	0.53%
H-朴素-PT	56.08%	32.12%	13.19%	43.15%	16.81%	3.02%
HNPS-PT	**64.10%**	**39.95%**	**16.25%**	**50.80%**	21.85%	4.26%
HNPS（我们的完整版本）	61.09%	36.27%	14.55%	48.96%	**22.02%**	**5.06%**

表 8：我们通过计算训练集与测试集性能之间的差值，展示各方法的“泛化差距”。

方法	按词元数量				按短程序数量		
	10-25	25-40	40-55	55-70	2	3	4
Naïve	-4.48%	-3.94%	-0.58%	-0.04%	-4.32%	-1.56%	-0.09%
朴素短版	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Naïve-short-finetune	-2.23%	-4.16%	-2.22%	-0.16%	-4.57%	-2.9%	-0.55%
H-朴素-PT	-8.28%	-14.14%	-15.26 %	**-7.35%**	-12.93%	-15.31%	10.17%
HNPS-PT	-8.46%	-14.8%	-17.5%	-9.92%	-13.3%	-18.1%	-11.99
HNPS（我们的完整版本）	**-7.38%**	**-12.87%**	**-14.3%**	-7.36%	**-12.13%**	**-14.25%**	**-9.49%**