GRADIENT-BASED PROGRAM SYNTHESIS WITH NEURALLY INTERPRETED LANGUAGES

Anonymous authorsPaper under double-blind review

000

001

002 003 004

010 011

012

013

014

016

017

018

019

021

025

026

027

028

029

031

033

037

040

041

042

043

044

046

047

048

051

052

ABSTRACT

A central challenge in program induction has long been the trade-off between symbolic and neural approaches. Symbolic methods offer compositional generalisation and data efficiency, yet their scalability is constrained by formalisms such as domain-specific languages (DSLs), which are labor-intensive to create and may not transfer to new domains. In contrast, neural networks flexibly learn from data but fail to generalise systematically. We bridge this divide with the Neural Language Interpreter (NLI), an architecture that learns its own discrete, symbolic-like programming language end-to-end. NLI autonomously discovers a vocabulary of subsymbolic primitive operations and uses a novel differentiable neural executor to interpret variable-length sequences of these primitives. This allows NLI to represent programs that are not bound to a constant number of computation steps, enabling it to solve more complex problems than those seen during training. To make these discrete, compositional program structures amenable to gradient-based optimisation, we employ the Gumbel-Softmax relaxation, enabling the entire model to be trained end-to-end. Crucially, this same differentiability enables powerful test-time adaptation. At inference, NLI's program inductor provides an initial program guess. This guess is then refined via gradient descent through the neural executor, enabling efficient search for the neural program that best explains the given data. We demonstrate that NLI outperforms in-context learning, test-time training, and continuous latent program networks (LPNs) on tasks that require combinatorial generalisation and rapid adaptation to unseen tasks. Our results establish a new path toward models that combine the compositionality of discrete languages with the gradient-based search and end-to-end learning of neural networks.

1 Introduction

A central challenge in machine learning is the trade-off between symbolic and neural representations. Symbolic approaches rely on explicit rules, which enable strong compositional generalisation (Lake & Baroni, 2018), often from only a few examples (Solar-Lezama et al., 2006; Gulwani, 2011). Yet their scalability is constrained by formalisms such as domain-specific languages, which require human effort to generate, may not transfer to other domains, and are combinatorially expensive to search. Neural approaches, by contrast, scale effectively but behave as monolithic models. The knowledge they acquire is entangled within their weights, making it difficult to reuse beyond the training distribution, even when generalisation only requires recombining concepts already learned (Baroni, 2020).

In the context of program synthesis, we make progress toward bridging this divide with a model that learns its own symbolic representation, end-to-end, directly from data. Specifically, it simultaneously learns a domain-specific neural language and a neural interpreter for such a language. Similar to traditional handcrafted symbolic representations, the learned language enables compositional generalisation. Similar to neural representations, the neural interpreter's differentiability allows us to use gradient descent to search the language-induced space for solution programs. Recent work, such as Latent Program Networks (LPNs) (Macfarlane & Bonnet, 2024), has explored learning program representations with continuous latent spaces. However, this approach is limited in its ability to generalise by composing learned concepts, which is a key strength of symbolic representations.

Our architecture, the Neural Language Interpreter (NLI), uses an encoder-decoder model to discover discrete representations (Jang et al., 2017; Maddison et al., 2016) of programs. To learn a discrete vocabulary that can represent programs in the target domain, we train on programming by example (PBE) tasks. During inference, conditioned on a specification of examples, NLI's encoder produces a sequence of discrete tokens, as its internal inferred program representation. NLI's encoder acts as a program inductor; the token sequence forms a neural program. The decoder serves as a neural executor, interpreting the program one token at a time, mapping the test input to an output. Both the encoder and decoder are designed to be fully differentiable, and so NLI can be trained end to end.

Since the neural executor consumes one token at a time, NLI is not bound to a constant number of computation steps, as in previous approaches such as LPN. The number of steps in NLI's programs can grow with the number of tokens of the neural program. This is important because it enables NLI to solve problems more challenging than those with constant-time requirements, seen during training. Moreover, since NLI's programs can recombine learned tokens in different ways and at different lengths, we hypothesize that its language supports the combinatorial generalisation lacking in previous approaches.

In addition to the engineering hurdle of designing domain-specific languages, our work is motivated by the need to bypass the difficult combinatorial search problem inherent to program synthesis. Rather than learning external guiding functions for search (Barke et al., 2020; Odena et al., 2021; Ameen & Lelis, 2023), guidance is embedded in the language NLI learns. Because the neural executor is differentiable, we can search in the space of neural programs with gradient descent. Synthesizing a neural program with NLI is thus analogous to local search in symbolic spaces (Husien & Schewe, 2016), but with the advantage of having gradient signals. Another benefit of a learned language is how the search is initialized, which can dramatically affect efficiency (Hoos & Stützle, 2004; Sadmine et al., 2024). NLI's inductor provides an initial guess for a neural program solution at test time, and the gradient search then refines this guess to find the combination of learned primitives that solves the problem.

In this paper, we introduce the Neural Language Interpreter (NLI), a model that learns its own discrete programming language and a differentiable interpreter for executing it. By combining symbolic compositionality with neural end-to-end training and gradient-based program search, NLI addresses the limitations of both paradigms. Across sequence based compositional benchmarks, NLI achieves strong out-of-distribution accuracy on length extrapolation, primitive extraction, and novel composition tasks, where in-context learning, test-time training, and latent program networks fail. NLI matches or exceeds the performance of neuro-symbolic baselines on DeepCoder, despite training only from input—output examples without ground truth program representations.

2 Problem Statement

We formalize our task as **program induction**, where the goal is to infer the underlying behaviour of an unknown program p from input-output examples using a model M. Given a set $S = \{(x_i, y_i)\}_{i=1}^n$ of n input-output pairs generated by p and a new query input x_{n+1} , $M(S, x_{n+1})$ predicts the corresponding output $p(x_{n+1})$. This aligns with the programming by example (PBE) formalization, where information about program p is available only via its outputs. Training tasks are formed by sampling a latent program p from a distribution P_{train} over the space of possible programs \mathcal{P} . Program specifications are formed from n+1 inputs sampled from the program-dependent conditional distribution $\{x_i\}_{i=1}^{n+1} \sim P(X|p)$. This distribution generates inputs relevant to the logic of program p. The first n input-output pairs form the specification $S = \{(x_i, p(x_i))\}_{i=1}^n$, from which the model must induce the program's logic. The model's objective is to minimise the prediction error between its prediction, $\hat{y}_{n+1} = M(S, x_{n+1})$, and the true output $p(x_{n+1})$, to train the model to generalise program execution to a new input, not merely fit the given pairs. The model has no access to the program's fully observable representation p during training or test time. This is vital, as real-world tasks often involve latent functions without an observable specification. At test time, the model is evaluated on programs drawn from P_{test} , which can differ from P_{train} in order to test for compositional generalisation.

3 DISCRETE SEQUENTIAL INFERENCE

Existing neural program synthesis methods fail at compositional generalisation, struggling to recombine learned concepts for novel tasks. The Neural Language Interpreter (NLI) overcomes this by learning a discrete, symbolic-like programming language end-to-end. Programs are variable-length token sequences processed by a differentiable neural executor, enabling training on input-output examples via Gumbel-Softmax. This facilitates efficient, gradient-based search at test time to refine initial programs. NLI outperforms baselines on tasks requiring combinatorial generalisation, successfully extrapolating program lengths and synthesizing novel compositions of learned skills.

4 TRAINING OBJECTIVE

We train the encoder-decoder with a variational objective inspired by the ELBO. The goal is to reconstruct a program's output for a query input, conditioned on a specification of other input-output pairs from the same program.

Formally, the program inductor q_{ϕ} infers a latent program representation from the specification, which the neural interpreter p_{θ} then executes to predict the output for a new query. The model is trained end-to-end on specifications of size n using a leave-one-out loss: for each pair, NLI induces a program from the remaining n-1 pairs $(S_i=S\setminus\{(x_i,y_i)\})$ and maximizes the likelihood of predicting the held-out pair. The objective is:

$$\mathcal{L}(\phi, \theta, \mathcal{D}) = \underset{S_i \sim \mathcal{D}}{\mathbb{E}} \left[\frac{1}{n} \sum_{i=1}^{n} \left(\mathcal{L}_{\text{recon}}(\phi, \theta; x_i, y_i, S_i) + \lambda_{\text{enc}} \cdot \mathcal{L}_{\text{enc}}(\phi; S_i) \right) \right]$$
(1)

where \mathcal{D} is the distribution of specifications. This objective has two components.

Reconstruction Loss ($\mathcal{L}_{\text{recon}}$) This term ensures that the latent program is expressive enough to predict the output of the program on a held-out input. It is defined as the negative log-likelihood of the target output y_i given the input x_i and the latent program $\tilde{\mathbf{z}}_i$ inferred from the sub-specification S_i :

$$\mathcal{L}_{\text{recon}}(\phi, \theta; x_i, y_i, S_i) = -\log p_{\theta}(y_i \mid x_i, \tilde{\mathbf{z}}_i), \quad \tilde{\mathbf{z}}_i \sim q_{\phi}(\cdot \mid S_i). \tag{2}$$

Encoder Loss (\mathcal{L}_{enc}) This regularizer shapes the latent program space. If \mathcal{L}_{enc} is chosen as a KL divergence between the encoder distribution $q_{\phi}(z \mid S_i)$ and a prior p(z), the objective reduces to the standard ELBO used in variational autoencoders (Kingma, 2013). In this work, we design \mathcal{L}_{enc} to encourage reuse of tokens in the neural vocabulary, biasing the encoder toward discovering a compositional latent program space. This choice promotes generalisation by encouraging primitives to be reused, rather than relying on arbitrary tokens to represent each new program.

4.1 DISCRETE PROGRAM REPRESENTATION LEARNING

The encoder of NLI functions as a program inductor, denoted as q_{ϕ} , from which a latent program representation $\mathbf{z}=(z_1,\ldots,z_T)$ is sampled, given a specification S_i containing input-output examples. This representation, \mathbf{z} , is a sequence of continuous vectors that serves as a differentiable proxy for a sequence of discrete tokens drawn from a learned codebook of size K. This codebook includes a dedicated skip token, which functions as a no-op, allowing the model to effectively learn shorter programs by ignoring certain computational steps within the fixed-length sequence T.

For program induction firstly a transformer, which we denote by the function h_{ϕ} , maps each pair (x_j,y_j) in a single specification to a sequence of contextual embeddings, $e_j=h(x_j,y_j)=(e_{j,1},e_{j,2},\ldots,e_{j,T})$. These sequences are then aggregated, across the specification, by computing the element-wise arithmetic mean across all n-1 pairs, $\bar{e}_t=\frac{1}{n-1}\sum_{j=1}^{n-1}e_{j,t}$, to produce a single, permutation-invariant sequence of specification embeddings, $\bar{e}=(\bar{e}_1,\bar{e}_2,\ldots,\bar{e}_T)$.

To obtain a differentiable proxy for a discrete program, this sequence of continuous embeddings is projected to the codebook space. A shared multi-layer neural network f maps each embedding e_t to a vector of logits, parameterising a categorical distribution over the K codebook entries. We then

163

164 165 166

167

168

169 170

171

172

173 174

175176

177

178

179

180

181

182

183

185

187

188

189

190 191

192

193

194

195

196

197

199

200

201

202

203

204205206207

208

209

210

211

212

213

214

215

apply the Gumbel-Softmax relaxation to sample a "soft" one-hot vector at each position $\tilde{\pi}_t$:

$$l_t = f(e_t) \in \mathbb{R}^K \qquad \qquad \pi(e_t, \tau_p, g_t) = \operatorname{softmax}\left(\frac{l_t + g_t}{\tau_p}\right) \in \Delta^K$$

where g_t is a sample from a Gumbel distribution and τ is the temperature. The final program representation $z=(z_1,\ldots,z_T)$ is defined as the continuous approximation of a discrete program, which is passed to the decoder, and is constructed by taking a weighted combination of the codebook embeddings V using these soft vectors:

$$z_t = V^{\top} \pi(\bar{e}_t, \tau_p, g_t), \quad z = (z_1, \dots, z_T)$$

During training, the temperature τ is steadily annealed, progressively improving the approximation of a discrete sample from the un-normalised distribution l_t .

4.2 RECURRENT NEURAL PROGRAM EXECUTION

A common failure point for standard decoders is that they overfit to program lengths and structures seen during training. We implement the neural interpreter as a recurrent application of an executor network to achieve compositional generalisation. This executor network conditions on the program representation z one token at a time, using a shared neural executor d_{θ} to iteratively update an intermediate program state s_t . This sequential execution naturally handles novel combinations of primitives and variable program lengths, forcing the model to learn reusable, abstract building blocks. This approach stands in contrast to methods like LPNs, which are limited to representing entire programs in a single monolithic embedding.

The execution process is detailed in Algorithm 1. An initial state is created by embedding the input query x_q . This state is then refined over T steps in a loop, where at each step t, the executor d_θ uses the current program token z_t to compute an updated state. A crucial feature is the skip-token gating mechanism: the probability of the skip token, $\pi_t[\text{skip_idx}]$, is taken from the encoder's output and used to linearly interpolate between the previous state s_{t-1} and the newly computed state. This allows the model to effectively ignore an instruction z_t . After the final token is processed, the last state is used to generate the output.

Algorithm 1 Neural Language Interpreter (Decoder p_{θ})

```
1: function p_{\theta}(y_q|x_q, z, \tau_d, h)
         (E, d, \text{MLP}) \leftarrow \theta
                                                                            \triangleright Unpack implicit parameters from \theta
 2:
         s_0 \leftarrow \operatorname{Embed}(x_q, E)
 3:
                                                                                                        for t=1 \rightarrow T do
 4:
 5:
              k_t \leftarrow d(s_{t-1}, z_{t-1})
                                                                                                    ▶ Transform state
              l_t \leftarrow \text{MLP}(k_t)
 6:
                                                                                     ▶ Project hidden state to logits
              \pi_t \leftarrow \operatorname{softmax} ((l_t + h_t)/\tau_d) > Apply Gumbel-Softmax (h_t is Gumbel noise)
 7:
 8:
              o_t \leftarrow E^{\top} \pi_t
                                                                                     s_t \leftarrow \pi_t[\text{skip\_idx}] \cdot s_{t-1} + (1 - \pi_t[\text{skip\_idx}]) \cdot o_t
 9:

    □ Update state with skip-gating

         end for
10:
11:
         l_u \leftarrow \text{MLP}(k_T)
                                                                                      return softmax(l_u)
13: end function
```

4.3 SEARCHING NEURAL PROGRAMS

A key benefit of our model is the ability to refine an initial program prediction at test time using gradient-based search. While the encoder provides a fast first guess, it may not be optimal, especially for out-of-distribution programs. Our search operates not in the discrete space of programs, but in a continuous relaxation of the discrete tokens. Because it represents programs as sequences of learned primitives, this space can construct entirely new programs of arbitrary length, even those never seen during training. The search, therefore, becomes a process of discovering these novel compositions.

The key benefit of searching in the relaxed representation of discrete tokens is that, like during training, execution remains fully end-to-end differentiable. This allows us to use gradient ascent, an

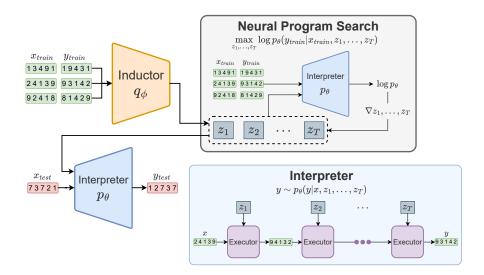


Figure 1: Overview of the inference process. The program inductor generates a sequence of latent program tokens conditioned on the specification. Neural Program Search refines this program to better explain the specification, and the neural interpreter then executes the improved program token by token.

efficient optimisation method crucial for navigating the vast combinatorial space of possible programs. The search objective is to find the latent embeddings e^* that maximise the expected likelihood of the specification data $S = \{(x_j, y_j)\}$, with the expectation taken over the program and layer Gumbel samples:

$$e^* = \arg\max_{\substack{e \\ h \sim \text{Gumbel}(0,1)^T \\ h \sim \text{Gumbel}(0,1)^T}} \mathbb{E} \left[\sum_{(x_j,y_j) \in S} \log p_{\theta}(y_j \,|\, x_j, z(e,\tau_p,g), \tau_d, h) \right]$$

In order to encourage finding a strong approximation to a discrete solution, during the optimisation of this objective, we perform temperature annealing of both τ_p and τ_d . The optimisation begins with a high temperature to smooth the landscape and encourage broad exploration. As the search progresses, we gradually lower the temperature, encouraging the discovery of a discrete program along with discrete intermediate computation between tokens.

Instead of using a single starting point, we initialize the search from multiple locations to better explore the vast program space and avoid getting stuck in local minima. Specifically, we sample m initial latent embeddings from a Gaussian distribution, with parameters $\mu=e_t,\sigma$. We then perform L steps of gradient ascent in parallel from each of the starting points. This parallel search strategy is not only effective for exploration but is also highly scalable, allowing us to leverage multiple hardware devices efficiently.

Most program induction methods have to contend with the problem of finding many programs that explain a given specification. This is a considerable concern in both pure parameter-based modeling and also latent space optimisation (Macfarlane & Bonnet, 2024), where early stopping is often used to limit the chance of converging to programs that do not generalise. In NLI, this is less of a concern as the search space is less flexible than such continuous search spaces, limited to recomposing a set of discrete embeddings. Therefore, the risk of overfitting is relatively limited.

5 EXPERIMENTS

We evaluate NLI's compositional generalisation capabilities across a custom diagnostic benchmark for compositional generalisation and the compositionality version of the DeepCoder benchmark (Balog et al., 2016), introduced in (Shi et al., 2023), comparing to a range of neural and neuro-symbolic baselines.

Benchmarks The custom suite uses fixed-length sequences (20) and is designed to reveal failure modes in PBE, where models see only input-output pairs. It comprises three splits containing different tasks: Shift-L, training on small sequence shifts $k \in \{1, \ldots, 5\}$ and testing on larger unseen shifts $k \in \{6, \ldots, 10\}$; Shift-P, the inverse, training on large shifts $k \in \{7, 8, 9\}$ and testing on smaller ones $k \in \{1, 2, 3\}$; and Comp-I, where models trained on single primitives (e.g., f(x)) must compose them at test time (e.g., f(g(x))). We also explore the compositionality deepcoder dataset that scales the number of primitives and program complexity, see appendix A.

Baselines and Models We compare NLI against several strong baselines: In-Context Learning (ICL), Test-Time Training (TTT), Latent Program Networks (LPN), and a discrete variant (D-LPN). We evaluate three inference strategies for our model: Base Inference (direct encoder output), Prior Search (sampling from the encoder), and our primary method, Gradient Search, which optimises the program in the latent space.

5.1 COMPOSITIONAL GENERALISATION IN NEURAL MODELS

Table 1: Performance for different methods and datasets, in the custom suite. We report final accuracy for both in-distribution and out-of-distribution test splits (ID and OOD).

	Shift-L		Shift-P		Comp-I	
Method	ID	OOD	ID	OOD	ID	OOD
In-Context	1.00	0.00	1.00	0.00	1.00	0.13
TTT	1.00		1.00	0.00	0.95	0.14
LPN	1.00	0.00	1.00	0.00	1.00	0.18
LPN Gradient Search	1.00	0.03	1.00	0.00	1.00	0.29
D-LPN	1.00	0.02	1.00	0.00	0.99	0.15
D-LPN Gradient Search	1.00	0.01	1.00	0.00	0.99	0.20
NLI	1.00	0.00	1.00	0.00	1.00	0.17
NLI Prior Search	1.00	0.10	1.00	0.00	1.00	0.23
NLI Gradient Search	1.00	0.99	1.00	1.00	1.00	0.91

We train all models for 100k batches, of size 512 and evaluate on held out test splits in-distribution and out of distribution. Due to the inference cost differences between NLI and baselines, for completeness for all baselines we also performed training runs with matched compute by increasing decoder layers, for all baselines this led to a degradation of in-distribution performance and no generalisation. We report the higher, low inference 2-layer decoder results in table 1.

5.1.1 LEARNED PROGRAM REPRESENTATIONS

Learned Program Representations for Shift-L					
Ground Truth Program	NLI Program Representation				
shift_left(1)	231				
shift_left(2)	231 231				
shift_left(3)	231 231 231				
shift_left(4)	231 476 231				
shift_left(5)	231 476 476				
shift_left(8) (OOD)	231 231 231 231 476 476				

325

326

327

328

329

330

331

332

333

334

335

336

337

338

339

340 341

342 343

344

345

346

347

348

349

350

351

352

353

354

355

356

357

358

359

360 361

362

364

366

367

368

369

370

371

372373

374

375

376

377

All models achieve near-perfect accuracy on the in-distribution (ID) test sets, demonstrating their ability to solve tasks similar to those seen during training with neural induction. On the more challenging out-of-distribution (OOD) splits, however, all baselines and the non-search variants of our model fail to generalise. In-Context Learning (ICL) and the Latent Program Network (LPN and D-LPN) show near 0% OOD accuracy on the shift tasks (Shift-L and Shift-P). Search-based LPNs achieve only minor gains on Compose Isolation (Comp-I), but still fail to solve the task. In contrast, NLI with Gradient Search exhibits strong compositional generalisation across all three benchmarks: on Shift-L (length generalisation) it reaches 99% by extrapolating from small to larger unseen shifts; on Comp-I (composing concepts) it achieves 91% by synthesising programs such as f(q(x)); and on Shift-P (primitive extraction) it attains a perfect 100% by "decompiling" primitives after training only on complex ones. These results confirm that NLI achieves systematic generalisation, enabled by gradient-based search, whereas the base encoder and prior search variants collapse to baseline performance. The learned codes further reveal systematic reuse of primitives. The model consistently represents a single left shift with token 231 and a two-step shift with token 476, constructing larger programs by combining these two building blocks. The OOD case of eight shifts is also expressed as a mixture of these primitives (found via gradient search), highlighting how generalisation arises from recombination rather than memorisation.

5.2 UNDERSTANDING THE ORIGINS OF NLI'S GENERALISATION CAPABILITIES

To investigate the origins of NLI's generalisation, we conduct an ablation study across the datasets in table 1, with results shown in fig. 2. The base model achieves nearly perfect OOD accuracy (97%), and we remove components individually to assess their importance. Most prove indispensable: dropping recurrent execution or the discreteness of either program or layer representations collapses OOD accuracy to near zero (1-5%). This shows that discrete programs, discrete layer traces, and recurrent dynamics are all essential for generalisation. Gumbel-Softmax has a subtler role. Removing layer-level Gumbel (no layer gumbel) causes failure (5%), but removing program-level Gumbel (no program gumbel) still yields strong accuracy (76%). Finally, dropping the skip token reduces perfor-

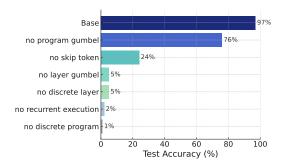


Figure 2: Ablations of the NLI base model to identify components critical for OOD generalisation.

mance to 24%, consistent with the model's ability to learn its own skip but benefiting from a dedicated token for faster and more stable training.

5.3 SCALING TEST-TIME PROGRAM SEARCH

To evaluate the effectiveness of our gradient-based search, we analyse how performance scales with the available computational budget at test time. We benchmark on the Comp-I dataset, varying two key hyperparameters: the number of parallel initialisations, Num starts, and the number of optimisation iterations, Gradient steps. The results, presented in fig. 3, demonstrate a strong and consistent positive correlation between test-time compute and final accuracy.

We observe clear, synergistic benefits from scaling both dimensions of the search. For any fixed number of starts, increasing the gradient steps consistently improves performance. For instance, with 2048 starts, accuracy grows from just under 50% with a single gradient step to nearly 80% with 400 steps. This

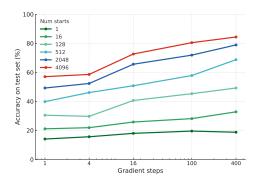


Figure 3: Performance on Compose-I, scaling two axes of test-time compute: gradient steps and number of starts.

shows that longer optimisation allows the model to better refine initial program specifications, effectively navigating the continuous embedding space to find more correct solutions. Simultaneously, increasing the number of starting points provides a substantial performance boost. At 16 gradient steps, moving from 1 start to 4096 starts elevates accuracy from approximately 18% to over 70%. This highlights the utility of a broader search in discovering a promising initialisation that can converge to a correct program.

5.4 DeepCoder

To assess the scalability of our approach, we evaluate NLI on the DeepCoder benchmark (Shi et al., 2023), a standard testbed for compositional generalisation in program synthesis. We note that the original training datasets for DeepCoder composition have not been made public; therefore, data generation was run from scratch to generate datasets of size 11.6 million induction tasks, to train the neural baselines (NLI, LPN and In-context). Due to the prohibitive costs of the data sampling function, this is less than the 60 million used in the original work, however baselines all achieve competitive performance. This benchmark contrasts fully neural programming-by-example (PBE) methods with neuro-symbolic approaches that leverage access to ground-truth programs during training. In contrast, NLI, along with neural baselines such as LPN, and an In-context baseline, must induce program behavior solely from input-output pairs. All neural benchmarks were trained for 200k batches of size 512, see appendix B for more details. We find that end-to-end neural methods such as

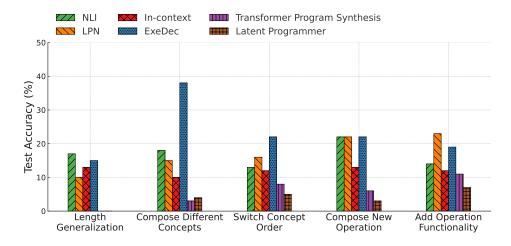


Figure 4: Comparison of fully neural baselines and NLI against neuro-symbolic methods. Neuro-symbolic models (ExeDec, Transformer, Latent Programmer) use ground-truth program annotations, while neural models (In-context, LPN, NLI) rely only on input—output pairs.

NLI and LPN substantially outperform earlier Latent Programmer approaches and Transformer-based program synthesis. Secondly, despite the absence of program supervision, they achieve performance competitive with ExeDec (Shi et al., 2023), highlighting the capacity of neural PBE approaches to autonomously discover structured program representations.

A direct comparison between NLI and LPN further reveals complementary strengths. NLI generalises more effectively to longer programs and novel concept compositions, which is a particular strength of NLI due to its ability to compose programs of arbitrary length. Whereas LPN excels at switching concept order and extending functionality with new operations. These differences suggest that their learned latent structures capture distinct inductive biases, leading to different generalisation behaviours out of distribution.

6 RELATED WORK

Symbolic Program Synthesis. Early work in program synthesis largely relied on symbolic techniques and DSLs. Classical systems, such as those by Summers (1977) and Gulwani (2011), used predefined DSLs with explicit search over symbolic programs. These methods provide interpretability and

exactness but suffer from scalability issues, as every new domain requires manual DSL design. Recent neuro-symbolic hybrids, such as DeepCoder (Balog et al., 2016), combine a neural predictor with symbolic search, predicting program components to accelerate search. However, their reliance on restricted DSLs limits generalisation beyond the designed primitives.

Neural Program Induction and Meta Learning Neural approaches aim to overcome the brittleness of symbolic methods by learning programs directly from examples. Neural Programmer-Interpreters (Reed & De Freitas, 2016) execute programs implicitly with recurrent models, while Devlin et al. (2017) introduced meta-induction for few-shot learning. These models improve adaptability but often fail to generalise compositionally and demand large supervision. ExeDec (Shi et al., 2023) added execution decomposition as an inductive bias, yet still relies on ground-truth decompositions and remains costly. Meta learning advances this by training networks to adapt across task distributions (Finn et al., 2017), a setup closely related to the optimisation considered here.

Latent Representations of Programs. Another line of work introduces latent spaces to represent programs more flexibly. CompILE (Kipf et al., 2018) segments demonstrations into reusable latent codes with Gumbel-Softmax relaxation, showing benefits for imitation learning. The Latent Programmer (Hong et al., 2020) extends this idea to discrete latent codes that plan over input—output examples, using a VQ-VAE style autoencoder with beam search in latent space, see appendix C for a discussion on its relation to NLI. Most recently, Latent Program Networks (LPNs) (Macfarlane & Bonnet, 2024) proposed continuous latent program representations to facilitate test-time search, but the lack of discrete compositional structure hinders combinatorial generalisation.

Compositionality and Generalisation. Compositional generalisation remains a central challenge in neural program synthesis. Lake & Baroni (2018) demonstrated that standard seq2seq models fail to generalise systematically to novel compositions. Approaches such as the Compositional Recursive Learner (CRL) (Chang et al., 2019) attempt to address this by learning to compose reusable transformations. Similarly, recursion-based methods (Cai et al., 2017) leverage inductive biases from programming languages to handle inputs of greater complexity than those seen during training. While these directions highlight the importance of compositional structure, they either rely on strong supervision or achieve only limited scalability.

Discrete Representation Learning. Discrete latent representations provide a natural way to capture compositional structure and improve interpretability. The Vector-Quantized Variational Autoencoder (VQ-VAE) (van den Oord et al., 2017) exemplifies this approach by learning a finite codebook of tokens, with gradients passed via a straight-through estimator. A complementary method is the Gumbel-Softmax relaxation (Jang et al., 2017), which reparameterizes categorical sampling with a differentiable approximation. Together, these techniques enable end-to-end training with discrete variables while retaining symbolic structure. A practical example arises in hierarchical reinforcement learning, where the options framework (Sutton et al., 1999) defines a set of reusable, temporally extended actions that compose into complex behaviors. Such discrete units, whether tokens in generative models or skills in RL, form compact and interpretable building blocks that support compositional generalization and long-horizon reasoning.

7 Conclusion

In this work, we introduced the Neural Language Interpreter (NLI), a novel architecture that bridges the divide between symbolic and neural approaches in program synthesis. By learning a discrete, symbolic-like language and a differentiable interpreter, NLI combines the compositional strengths of symbolic systems with the flexibility of neural networks. The model discovers a vocabulary of primitive operations and composes them into variable-length programs, refined at test time through efficient gradient-based search. Our evaluations show that NLI outperforms existing methods on challenging compositional generalisation tasks, with ablations confirming that the discrete, sequential program representation is key to this success. **Limitations and Future Work:** A limitation of this work is the relatively high search cost in the representation space, though we find NLI is less susceptible to overfitting despite this budget. Another limitation is that the interpreter currently conditions each layer on a single token, whereas conditioning on multiple tokens could enable the model to learn parameterised functions (e.g., add(1), add(2)). Future work could consider more budget-efficient search strategies and alternative training methods to make search more efficient, as well as extending the token conditioning to support richer parameterised primitives.

REFERENCES

- Saqib Ameen and Levi H. S. Lelis. Program synthesis with best-first bottom-up search. *Journal of Artificial Intelligence Research*, 77:1271–1310, 2023. doi: 10.1613/jair.1.14394.
- Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.
 - Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. In *Proceedings of the ACM on Programming Languages*, volume 4, pp. 1–29, 2020.
 - Marco Baroni. Linguistic generalization and compositionality in modern artificial neural networks. *Philosophical Transactions of the Royal Society B*, 375(1791):20190307, 2020.
 - Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. *arXiv preprint arXiv:1704.06611*, 2017.
 - Michael Chang, Abhishek Gupta, Sergey Levine, and Thomas L Griffiths. Automatically composing representation transformations as a means for generalization. In *International Conference on Learning Representations (ICLR)*, 2019.
 - Jacob Devlin, Jonathan Uesato, Rishabh Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Neural program meta-induction. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.
 - Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pp. 1126–1135. PMLR, 2017.
 - Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.
 - Jason Hong, Maxwell Nye, Joshua B Tenenbaum, and Charles Sutton. Latent programmer: Discrete latent codes for program synthesis. In *Advances in Neural Information Processing Systems* (NeurIPS), 2020.
 - Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Elsevier/Morgan Kaufmann, 2004.
 - Idress Husien and Sven Schewe. Program generation using simulated annealing and model checking. In Rocco De Nicola and Eva Kühn (eds.), *Software Engineering and Formal Methods*, pp. 155–171, 2016. ISBN 978-3-319-41591-8.
 - Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In *International Conference on Learning Representations (ICLR)*, 2017.
 - Diederik P Kingma. Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114, 2013.
 - Thomas Kipf, Ethan Li, Hanjun Dai, Zornitsa Kozareva, Jiaming Song, Arvind Neelakantan, and Max Welling. Compile: Compositional imitation learning and execution. In *International Conference on Machine Learning (ICML)*, 2018.
 - Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International conference on machine learning*, pp. 2873–2882. PMLR, 2018.
 - Matthew V Macfarlane and Clément Bonnet. Searching latent program spaces. *arXiv preprint arXiv:2411.08706*, 2024.
 - Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.
 - Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. BUSTLE: Bottom-up program synthesis through learning-guided exploration. In *International Conference on Learning Representations*, 2021.

Scott Reed and Nando De Freitas. Neural programmer-interpreters. In *International Conference on Learning Representations (ICLR)*, 2016. Quazi Asif Sadmine, Hendrik Baier, and Levi Lelis. Language models speed up local search for finding programmatic policies. Transactions on Machine Learning Research, 2024. ISSN 2835-8856. Zhi Shi, Maxwell Nye, Rudy Bunel, Rishabh Singh, Pushmeet Kohli, and Alexander L Gaunt. Exedec: Execution decomposition for compositional generalization in neural program synthesis. In Advances in Neural Information Processing Systems (NeurIPS), 2023. Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Martin Rinard. Combinatorial sketching for finite programs. In Computer Aided Verification, volume 4144 of Lecture Notes in Computer Science, pp. 404–420. Springer, 2006. Phillip D Summers. A methodology for lisp program construction from examples. Journal of the *ACM (JACM)*, 24(1):161–175, 1977. Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. Artificial intelligence, 112(1-2):181–211, 1999. Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. In Advances in Neural Information Processing Systems, 2017.

A DATASETS

A.1 COMPOSITIONALITY BENCHMARK

We constructed the Compositionality Benchmark using our own sampling and problem synthesis procedures to evaluate distinct facets of compositional reasoning. The benchmark comprises three main tasks designed to probe different dimensions of generalisation. For each task, dataset sizes were chosen to provide a robust training scale and sufficient evaluation coverage. The three splits are:

- Permutation Length Generalisation (Shift-L). This task measures extrapolation on a parameterized function. The model learns a left_shift (n) operation on a sequence. Training is restricted to a small, contiguous range of integer shifts, specifically for n ∈ {1, 2, 3, 4, 5}. Evaluation is performed on larger, unseen shift values, n ∈ {6, 7, 8, 9, 10}. This measures the model's ability to generalise beyond the magnitude of parameters observed during training.
- 2. Sub-Function Extraction (Shift-P). This task tests whether a model can infer a general, parameterized function from sparse and non-contiguous examples. The underlying operation is again left_shift (n). Training is performed on a sparse set of non-adjacent shift values (e.g., $n \in \{5,7,9\}$). Evaluation then probes generalisation to a different, unseen range of values (e.g., $n \in \{1,2,3\}$), testing whether the model has learned the abstract concept of "shifting by n" rather than memorizing separate programs for each training example.
- 3. Composition of Primitives (Comp-I). This task evaluates whether a model can compose primitive functions it has only seen in isolation. The model is provided with a library of over 20 primitive sequence-to-sequence operations (e.g., reverse, shift_left_3, increment_2). During training, the model only sees programs consisting of a single primitive operation. For evaluation, it must execute programs that are compositions of two or more primitives, testing for generalisation from individual operations to novel compositions.

Table 2: Dataset sizes for the Compositionality Benchmark.

Split	Size
Train	2,000,000
Test	10,000

A.2 DEEPCODER

We use the DeepCoder domain and adopt the compositional generalisation splits from the ExeDec codebase Shi et al. (2023). Following their Domain-Specific Language (DSL) and splitting procedures, we sampled 2,000,000 training tasks and 10,000 test tasks. The five splits are designed to probe different dimensions of compositional generalisation:

- 1. **Length-Generalisation.** Training programs contain 1–4 lines, while test programs have length 5. This evaluates whether models can extrapolate to deeper compositions than observed during training Balog et al. (2016).
- Compose-Different-Concepts. Operations are partitioned into two groups: (i) all first-order operations plus Map, and (ii) all remaining higher-order operations. Training only composes within a single group, while test programs require mixing across groups. This measures cross-concept compositionality.
- 3. **Switch-Concept-Order.** Training tasks always compose operations in a fixed group ordering (e.g., first-order → higher-order), while test tasks reverse the ordering. This evaluates whether models can generalise to new sequential structures of concepts.
- 4. Compose-New-Operation. The held-out operation is Scanl1. Training tasks either use Scanl1 in isolation (25% of tasks) or exclude it entirely, while test tasks require Scanl1 to be composed with other operations. This probes whether the model can generalise an operator from isolated usage to composed contexts.

5. **Add-Operation-Functionality.** Training only uses Scanl1 with lambdas (-) and min. Test tasks require Scanl1 with new lambdas (+), (\times) , and max. This tests whether models can extend their understanding of a known operator by analogy to other operations.

Table 3: Dataset sizes for DeepCoder, generated using the ExeDec repository.

Split	Size
Train	11,600,000
Test	10,000

B HYPERPARAMETERS

Table 4: Model Hyperparameters for NLI. The same default configuration was used across all datasets.

Hyperparameter	Shift-L	Shift-P	Compose I	DeepCoder
	Model Arcl	nitecture		
Model Dimension (d_{model})	128	128	128	128
Number of Heads (n_{head})	8	8	8	8
Feed-Forward Dimension ($d_{\rm ff}$)	512	512	512	512
Encoder Layers	2	2	2	4
Decoder Layers	2	2	2	2
Positional Embedding	Sinusoidal	Sinusoidal	Sinusoidal	Sinusoidal
Gradient Clip Norm	2.0	2.0	2.0	2.0
	Program Ge	eneration		
Program Vocabulary Size	512	512	512	512
Program Length (Training)	10	10	4	4
	Train	ing		
Learning Rate	2e-4	2e-4	2e-4	2e-4
Num Batches	100k	100k	100k	200k
Gumb	el-Softmax Sa	mpling (Progra	am)	
Use Program Gumbel	True	True	True	True
Start Temperature	8.0	8.0	8.0	8.0
End Temperature	0.5	0.5	0.5	0.5
Annealing Batches	20,000	20,000	100,000	200,000
Decay Strategy	Exponential	Exponential	Exponential	Exponential
Straight-Through	False	False	False	False
Gumbel-S	Softmax Samp	ling (Decoder	Layer)	
Use Layer Gumbel	True	True	True	True
Start Temperature	2.0	2.0	2.0	2.0
End Temperature	0.5	0.5	0.5	0.5
Annealing Batches	20,000	20,000	100,000	200,000
Decay Strategy	Exponential	Exponential	Exponential	Exponential
Straight-Through	False	False	False	False
	Regularizatio	n & Losses		
Encoder Loss Coefficient	0.00001	0.00001	0.00001	0.00001
	Sear	ch		
Gradient Steps	100	100	100	100
Number of Initializations	1024	1024	8192	1024
Std for initialisation	7.5	7.5	7.5	7.5

Table 5: Model Hyperparameters for LPN. The same default configuration was used across all datasets.

Hyperparameter	Shift-L	Shift-P	Compose I	DeepCoder	
Model Architecture					
Model Dimension (d_{model})	512	512	512	512	
Number of Heads (n_{head})	8	8	8	8	
Feed-Forward Dimension ($d_{\rm ff}$)	512	512	512	512	
Encoder Layers	2	2	2	4	
Decoder Layers	2	2	2	2	
Use Layer Normalization	True	True	True	True	
Positional Embedding	Sinusoidal	Sinusoidal	Sinusoidal	Sinusoidal	
Dropout Rate	0.0	0.0	0.0	0.0	
$VA\hat{E}$ Beta (β)	0.001	0.001	0.001	0.001	
Gradient Clip Norm	2.0	2.0	2.0	2.0	
Training					
Learning Rate	2e-4	2e-4	2e-4	2e-4	
Num Batches	100k	100k	100k	200k	

Table 6: Model Hyperparameters for D-LPN. The same default configuration was used across all datasets.

datasets. Hyperparameter	Shift-L	Shift-P	Compose I			
Model Architecture						
Model Dimension (d_{model})	512	512	512			
Number of Heads (n_{head})	8	8	8			
Feed-Forward Dimension ($d_{\rm ff}$)	512	512	512			
Encoder Layers	2	2	2			
Decoder Layers	2	2	2			
Use Layer Normalization	True	True	True			
Positional Embedding	Sinusoidal	Sinusoidal	Sinusoidal			
Dropout Rate	0.0	0.0	0.0			
Gradient Clip Norm	2.0	2.0	2.0			
	Training					
Learning Rate	2e-4	2e-4	2e-4			
Num Batches	100k	100k	100k			
Gumbel-Softr	nax Sampling	(Program)				
Use Program Gumbel	True	True	True			
Start Temperature	8.0	8.0	8.0			
End Temperature	0.5	0.5	0.5			
Annealing Batches	20,000	20,000	100,000			
Decay Strategy	Exponential	Exponential	Exponential			
Straight-Through	False	False	False			
Gumbel-Softmax Sampling (Decoder Layer)						
Use Layer Gumbel	True	True	True			
Start Temperature	2.0	2.0	2.0			
End Temperature	0.5	0.5	0.5			
Annealing Batches	20,000	20,000	100,000			
Decay Strategy	Exponential	Exponential	Exponential			
Straight-Through	False	False	False			

Table 7: Model Hyperparameters for In-context. The same default configuration was used across all datasets.

Hyperparameter	Shift-L	Shift-P	Compose I	DeepCoder	
Model Architecture					
Model Dimension (d_{model})	512	512	512	512	
Number of Heads (n_{head})	8	8	8	8	
Feed-Forward Dimension ($d_{\rm ff}$)	512	512	512	512	
Encoder Layers	2	2	2	4	
Decoder Layers	2	2	2	2	
Use Layer Normalization	True	True	True	True	
Positional Embedding	Sinusoidal	Sinusoidal	Sinusoidal	Sinusoidal	
Dropout Rate	0.0	0.0	0.0	0.0	
Gradient Clip Norm	2.0	2.0	2.0	2.0	
Training					
Learning Rate	2e-4	2e-4	2e-4	2e-4	
Num Batches	100k	100k	100k	200k	

C COMPARISON TO DISCRETE LATENT PROGRAMMER

We compare our model with the Discrete Latent Programmer (DLP) (Hong et al., 2020), which also employs discrete latent codes for program induction. While both approaches share this high-level similarity, they diverge substantially in their architectures, training assumptions, and mechanisms for test-time adaptation. The key differences lie in how programs are executed and how search is performed at inference.

C.1 PROGRAM EXECUTION AND SUPERVISION

In our model, program tokens are interpreted by a recurrent neural interpreter that applies each token as an operation to an intermediate state. This sequential execution enables variable-length programs, promotes compositional reuse of learned primitives, and allows the model to be trained end-to-end on raw input—output examples alone. Since outputs can be directly compared to targets, no ground-truth program annotations are required.

DLP, by contrast, does not include a neural interpreter. Its decoder predicts full program sequences from latent codes, and training requires access to the underlying program representations. This reliance on program supervision restricts DLP to domains where the generating programs are known and a domain-specific language is available, limiting its applicability beyond synthetic benchmarks.

C.2 TEST-TIME PROGRAM SEARCH

A further distinction arises in test-time adaptation. Our model exploits the differentiability of the neural interpreter to refine latent program embeddings via gradient-based search. This procedure enables efficient adaptation: initial program guesses from the encoder can be continuously optimized to better fit new examples, even when they require novel compositions not seen during training.

In contrast, DLP performs beam search in the discrete program space. This search is combinatorial, lacks gradient guidance, and cannot refine programs based on execution error. As a result, DLP's generalisation is hindered, particularly in out-of-distribution settings where small corrections to a predicted program are necessary. By enabling gradient-based refinement in a relaxed latent space, our model provides a more powerful and adaptive mechanism for program synthesis.

D NLI PROGRAM REPRESENTATIONS

In this section, we provide examples of the discrete latent codes discovered by NLI, both for in-distribution programs and for how these primitives are composed by search to generalise out-of-distribution (OOD).

D.1 SHIFT-L

We study the task of shifting sequences to the left. During training, the model observes shifts of length 1 to 5 (inclusive). In principle, the network could learn a separate token for each shift. Instead, it discovers a more efficient representation by reusing tokens. Specifically, it learns a token (231) that corresponds to a single left shift. By repeating this token, the network composes shifts of lengths 2 and 3. For larger shifts, it introduces a second token (476), which corresponds to a two-step shift. This enables the model to combine primitives to generate more complex shifts.

For example, a shift of 4 is represented as one two-step shift plus two one-step shifts. At test time, when generalising OOD to larger shifts, the model composes primitives in the same manner. For instance, to represent an 8-step shift, it uses four single-shift tokens and two two-shift tokens. This demonstrates both compression (a small set of primitives) and compositionality (systematic reuse of primitives).

```
991
992
993
       Example 1: Shift Left by 1
994
995
       Task Specification:
       Input: [8, 2, 5, 9, 1, 6, 3, 4, 7, 0]
996
       Output: [2, 5, 9, 1, 6, 3, 4, 7, 0, 8]
997
998
       Ground Truth Program: y = left_shift(x, 1)
999
      NLI Program Representation: 231
1000 11
1001
1002
       Example 2: Shift Left by 2
1003
1004
      Task Specification:
    16
      Input: [4, 6, 7, 1, 9, 0, 3, 8, 5, 2]
1005
    17
      Output: [7, 1, 9, 0, 3, 8, 5, 2, 4, 6]
1006
1007
      Ground Truth Program: y = left_shift(x, 2)
1008
      NLI Program Representation: 231 231
1009
1010
1011 24
       Example 3: Shift Left by 3
1012 25
1013
      Task Specification:
1014 28
      Input: [3, 7, 4, 0, 6, 2, 9, 5, 8, 1]
1015 29
      Output: [0, 6, 2, 9, 5, 8, 1, 3, 7, 4]
1016 30
       Ground Truth Program: y = left_shift(x, 3)
    31
1017
      NLI Program Representation: 231 231 231
    32
1018
1019
1020 35
1021 36 Example 4: Shift Left by 4
1022
      Task Specification:
    38
1023
      Input: [5, 1, 9, 2, 8, 6, 0, 7, 3, 4]
1024
      Output: [8, 6, 0, 7, 3, 4, 5, 1, 9, 2]
1025
    41
      Ground Truth Program: y = left_shift(x, 4)
```

```
1026
     43 NLI Program Representation: 231 476 231
1027
1028 45
1029 46
1030 <sup>47</sup>
       Example 5: Shift Left by 5
1031
     49 Task Specification:
1032 50 Input: [9, 4, 1, 5, 2, 7, 6, 0, 3, 8]
1033 51 Output: [7, 6, 0, 3, 8, 9, 4, 1, 5, 2]
1034 52
       Ground Truth Program: y = left_shift(x, 5)
1035 53
1036 54 NLI Program Representation: 231 476 476
1037
1038 57
1039 58 Example 6 (OOD): Shift Left by 8
1040 59
1041 60 Task Specification:
       Input: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Output: [8, 9, 0, 1, 2, 3, 4, 5, 6, 7]
    61
1042 62
1044 64 Ground Truth Program: y = left_shift(x, 8)
1045 65 NLI Program Representation: 231 231 231 476 476
1046
1047
```

Listing 1: Learned NLI Program Representations for List Shift Tasks.

E APPENDIX: TOKEN REUSE REGULARISATION LOSS

To bias the model toward learning a compact and reusable set of primitives, we introduce a regularisation term applied to the encoder's output distribution. Since the encoder generates the latent program representation, this regulariser directly shapes the structure of the learned latent space. We refer to this term as the *token reuse loss*. Its role is to discourage the encoder from spreading probability mass across too many distinct program tokens within a batch. By promoting reuse, the model is incentivised to discover a small set of fundamental operations that can be recombined to solve a broad range of tasks, thereby fostering compositional generalisation.

Formally, the loss is defined as the expected number of unique program tokens used across a training batch. Crucially, this expectation can be computed in a differentiable form, enabling direct optimisation via gradient descent.

Let P denote the tensor of token probabilities output by the encoder, with dimensions (B, N, V), where B is the batch size, N is the program length, and V is the vocabulary size. The probability of token k being chosen at position i in sequence b is denoted $p_{b,i,k}$.

To approximate the probability that token k never appears in the batch, we make the simplifying assumption that token draws are independent across positions and across examples. Under this assumption, the probability of never selecting token k is:

$$\mathbb{P}(\text{token } k \text{ never appears}) = \prod_{b=1}^{B} \prod_{i=1}^{N} (1 - p_{b,i,k}). \tag{3}$$

For numerical stability, we compute this in log-space:

$$\log \mathbb{P}(\text{token } k \text{ never appears}) = \sum_{b=1}^{B} \sum_{i=1}^{N} \log(1 - p_{b,i,k}). \tag{4}$$

The probability that token k appears at least once in the batch is therefore:

$$\mathbb{P}(\text{token } k \text{ appears}) = 1 - \exp\left(\sum_{b=1}^{B} \sum_{i=1}^{N} \log(1 - p_{b,i,k})\right). \tag{5}$$

By linearity of expectation, the expected number of unique tokens used in the batch is:

$$\mathcal{L}_{\text{reuse}} = \sum_{k=1}^{V} \mathbb{P}(\text{token } k \text{ appears}). \tag{6}$$

PRACTICAL CONSIDERATIONS

The token reuse loss is added to the encoder loss with weight λ_{reuse} . As a batch-level statistic, it can be sensitive to batch size and may be unstable for small batches. In practice, we used large batches and did not observe instabilities. Importantly, the loss must not dominate training. We therefore apply a very small weight, ensuring that it provides only a gentle inductive bias toward compact vocabularies. This prevents collapse into degenerate solutions such as a single-token language, which would be too limited to represent complex tasks.

F THE USE OF LARGE LANGUAGE MODELS (LLMS)

In this work, large language models (LLMs) were used solely as a tool for polishing the writing, specifically to remove grammatical and spelling errors. They did not contribute to research ideation or any other significant aspects of the paper.