TOWARDS SELF-IMPROVING LANGUAGE MODELS FOR CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Language models for code generation improve predictably with the size of their training dataset, but corpora of high-quality human-written code are inevitably a finite, or at least slow-growing, resource. It is therefore desirable to find ways to train language models to improve autonomously, ideally starting from scratch. In this paper, we present a method that combines search and learning in an expert iteration scheme to improve code generation models without requiring any human-written code. Solutions to programming problems found by neurally-guided search provide training data for the language model. As training progresses, the language model increasingly internalizes knowledge about programming, enabling more efficient search, thereby solving harder problems. Using small, randomly initialized language models, we study how different design choices, such as the nature of the search procedure, the difficulty of the programming problems, and the tradeoff between training and search compute, influence the rate of learning progress.

1 INTRODUCTION

The success of language models is largely due to their ability to learn in a supervised manner on huge corpora of data produced by humans with widely differing goals and skillsets, and to subsequently be fine-tuned and aligned on a desired task or sets of tasks (Radford et al., 2018). There is little evidence that this two-stage training scheme is capable of allowing language models to fulfill the promise of continuous self-improvement: while human-generated data is plentiful, only a small fraction of it is of high quality, and even then, there is no explicit mechanism for these models to *improve* over the data they were exposed to.

Kahneman (2017) propose to divide intelligence into two processes: *System 1* and *System 2*. System 1 embodies the capability to intuitively recognize patterns and values, such as the "goodness" of a certain board state during a game of chess; while System 2 embodies the capability to look ahead and reason about one's actions. Both are necessary for success, since looking ahead requires intuition to assess the possible value of actions, and this intuition serves in turn to suggest which actions might be advantageous to consider during planning. On that premise, Anthony et al. (2017a) devise an algorithm for the training of intelligent agents, referred to as Expert Iteration (ExIt), in which the intuitive component is embodied by a Neural Network (NN), and the reasoning component by a powerful search algorithm based on Monte Carlo Tree Search (MCTS) (Kocsis & Szepesvári, 2006). The search algorithm leverages the intuition provided by the NN to guide its search, and the results of this planning process are then distilled back into the NN, inducing a virtuous cycle in which the search algorithm *improves* over the policy embodied by the NN, which takes over the task of *generalising* these improvements to unseen situations.

ExIt-like approaches, in particular (Silver et al., 2018; Schrittwieser et al., 2020) for board games and (Mankowitz et al., 2023; Roy et al., 2021) for software and hardware design, have been spectacularly successful. These methods for self-improvement work around the need for high-quality training data by generating their own, up to the point that they have been successfully used to train agents *from scratch*, without human-generated experience. In this paper, we explore how ExIt could be used to lift the capabilities of LMs to the level of System 2-type planning and reasoning.

We consider the task of *Programming by Examples (PBE)* as a benchmark (Alet et al., 2021; Vaduguru et al., 2023; Gulwani, 2011; Chen et al., 2021). PBE consists in writing a program which satisfies user-defined input-output specifications, such as writing a program that outputs the addition of its inputs.

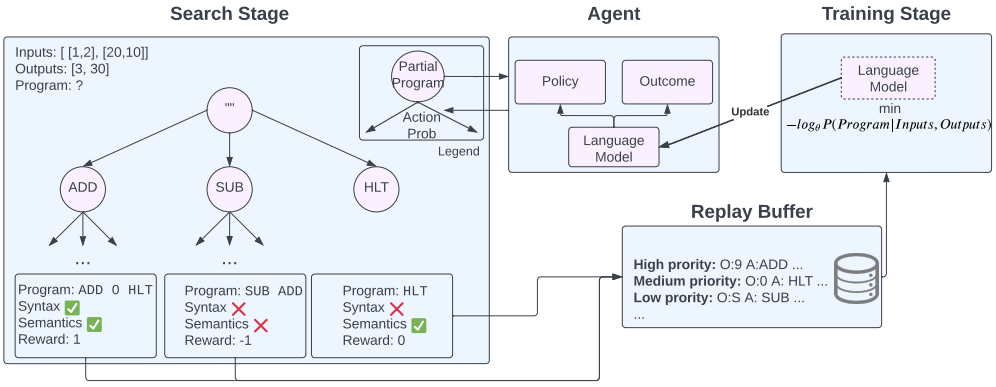


Figure 1: Overview of our expert iteration pipeline. In the search stage we generate programs guided by the language model, which are then stored in the replay buffer. During the training stage we fine-tune the language model on these, with a priority for high-reward programs.

Only the input-output specifications are available to the agent, which must, based on these alone, infer the meaning of the task and correctly write a program to solve it. One advantage of PBE lies in the fact that programs can be quickly and cheaply evaluated for *semantic* correctness, removing the need for complex reward engineering. In the machine learning (ML) literature, PBE has been mostly considered in the setting of inferring formulas (Chen et al., 2021) or regular expressions (Gulwani, 2011; Vaduguru et al., 2023) from examples. Here we shall instead focus on inferring programs written in an *assembly* language from numerical integer examples.

This paper is structured as follows: in section 2 we detail our ExIt method, with details on both its search and learning components. In section 3 we evaluate it on two families of tasks, each meant to benchmark the performance of ExIt when faced with a qualitatively different distribution of problems. Sections 4 and 5 situate our work into the broader ML literature and report the conclusion which can be drawn from our study, respectively.

2 METHOD

Our goal is to teach an agent to write programs that solve tasks specified by input-output examples. In particular, we consider the reinforcement learning setting where we have access to a reward function which assesses the correctness of complete programs. Based on this reward, the agent learns to write programs one instruction (action) at a time.

On a high level, expert iteration (Silver et al., 2018; Anthony et al., 2017b) repeats two phases: *search*, in which it discovers better programs by guiding a search algorithm with learned priors, and *train*, in which we train a neural network on those programs to improve the priors for the next search phase. Under the condition that the search step improve (in expectation) over the prior policy given by the neural network, this procedure leads to a virtuous cycle of self-improvement. The search step effectively produces synthetic data—this is akin to discovering new knowledge—which is then distilled during the train step to acquire a stronger prior.

2.1 OUTCOME AND REWARD

Upon completing a program P the agent gets a reward $r(P)$ that depends on the outcome of trying to run it. A program is completed either by the "HLT" instruction, or by reaching a set maximal length. Any action that does not complete the program gets zero reward, which means that the agent only gets a reward signal at the end of an episode. It can occur that properly terminated programs do not run, or that the agent does not manage to terminate the program before the maximal length expires. We aim to discourage those outcomes with negative rewards and distinguish between three cases: $r(P) = -2$ for programs that exceed a specified length (a truncation error), $r(P) = -1$ for

programs that contain syntax errors, and $r(P) = -1$ for programs that cannot be executed despite being syntactically correct (a runtime error). If the program is error-free and produces an output when executed, its reward will correspond to the fraction of program-produced outputs that match the ones specified in the input-output examples; hence $r(P) \in [0, 1]$ in this case.

2.2 PROGRAMMING WITH A LANGUAGE MODEL

We use a Transformer (Vaswani et al., 2017) to model the policy and value predictors. The agent's actions append a code unit (e.g., `ADD` or `2`) to the current program. The language model predicts the probability of the resulting program, which is then used to predict a distribution/policy over the next code unit.

As for the value, instead of directly predicting it, we predict a distribution over the different execution outcomes. We then compute the value as the expectation of this distribution, each outcome being associated with a numerical reward as detailed above. This allows the model to learn about semantically distinct outcomes, like syntax and runtime errors, that result in the same reward.

Since both the policy and value predictor greatly overlap in their purpose to identify high value continuations to the program, we learn both with a single model. We enable this by alternating between outcome predictions marked with `O:` and actions marked with `A:`:

```
inputs=[[0,1],[4,6]]outputs=[1,10]
   O:9 A:ADD O:9 A:0 O:9 A:HLT
```

When the program is error-free the outcome is represented by a character in `0,...,9` with `0` signifying no correct outputs and `9` all outputs being correct. When the outcome is an error we denote them with one of `T,S,R`. For the actions we refer the reader to section 3.1 for an outline of our custom assembly language. In the above example, the input-output examples $I_1 = (0, 1), O_1 = 1, I_2 = (4, 6), O_2 = 10$ are flattened to a string and used as part of the prompt for the subsequent outcomes and actions.

We train the transformer as a language model on strings, with a next token prediction objective. This has the advantage that we can train in parallel over a whole "trajectory" or episode rather than separate state-action pairs, which greatly enhances efficiency. In order to avoid wasting model capacity on learning the distribution of the input-output examples, we mask them during training. To accommodate code units that consist of multiple tokens, we compute the policy, conditioned on the input-output examples I, O and a partial program P as:

$$u(I, O, P, a) = \sum_{t_k \in a} \mathrm{LM}(t_k | [I, O, e(P), \texttt{"A:"} t_{0 \ldots k-1}]), \tag{1}$$

$$p(a_i | I, O, P) = \mathrm{softmax}(u(I, O, P, a_i)), \tag{2}$$

where the softmax is taken over the set of allowed code units $\mathcal{A}$, $e(\cdot)$ is a function which, given a partial program P, outputs the corresponding string representation described above, $[\ldots]$ denotes string concatenation, and $\mathrm{LM}(t|s)$ denotes the logits assigned by the language model for a token t given some prompt s.

2.3 TRAINING DISTRIBUTION

Our training approach for the language model relies on sequences corresponding to full state-action trajectories that start from an initially empty program and end at a complete or truncated program. The optimal target policy has zero probability for all actions that lead to incorrect or invalid programs. This means that the optimal training sequences are those that consist of programs that correctly solve the task of interest. Since our data is not labeled we do not have access to samples of this distribution. Instead, we aim to close in on this distribution through expert iteration, by improving the success rate of the generated samples with a search algorithm on top of the learned policy. We additionally speed up this process by storing all sampled trajectories in a replay buffer from which we then sample training sequences with reward-based prioritized sampling (Schaul et al., 2015). This has the effect that we primarily train the model on tasks for which the expert can find the correct program. If we assume that the search procedure is effective this set of solvable tasks gradually expands during expert iteration, including more programs that were previously not solvable. This effect is akin to

an implicit curriculum where the model first trains on simpler tasks, then gradually trains on more difficult tasks.

2.4 SEARCH METHODS

The purpose of search in expert iteration is to improve over the learned policy, and is crucial for finding high-quality training sequences for previously unsolved tasks.

2.4.1 SAMPLING & FILTERING (S&F)

Sampling & Filtering is a simple yet effective search method that only requires a prior policy. During the sampling step, actions are sampled using temperature sampling from the probability distribution over next actions conditioned on the program so far, defined in Equation 2. We can increase the amount of compute expended on search by increasing the number of completions for each task. We control this with the `num_rollouts` parameter. In the filtering step we only keep the program with the highest reward per task.

2.4.2 MONTE CARLO TREE SEARCH (MCTS)

In contrast to sampling & filtering, the Monte Carlo Tree Search (MCTS) algorithm we present here can exploit both a prior policy and a value function prediction. We base our MCTS method on the Gumbel MuZero (Danihelka et al., 2021) version for its good performance on low simulation budgets.

MCTS improves its search for high reward trajectories by aggregating value estimates from multiple Monte Carlo simulations to progressively guide future simulations towards more rewarding paths. Note that we execute MCTS anew at every state to select the agent's next action.

Based on the current state (partial program) we initialize the root node in the search tree. Then we grow the search tree one node at a time by repeating four steps: 1. starting from the root node successively select a child until reaching a leaf node in the search tree, 2. from this node select an action and add the new leaf to the tree (if admissible by the environment), 3. estimate the value or reward at the new leaf node, and 4. backpropagate the new information to all nodes in the path from the new leaf node to the root.

In the first step we select child nodes at the *root* node using the Sequential Halving algorithm (Karnin et al., 2013; Danihelka et al., 2021). It divides the total simulation budget into equal phases. In each phase all the considered actions are visited equally often until the simulation budget for that phase is used up. Then, half of the actions with the lowest score are dropped, and this is repeated in the next phase. We compute the score for an action a, the current action value estimate $q(a)$, and the logits $u(I, O, P, a)$ (Equation equation 2) as $g(a) + u(I, O, P, a) + \sigma(q(a))$ (Danihelka et al., 2021). To understand this sum, note that $\arg\max_a g(a) + u(I, O, P, a)$ with Gumbel noise $g(a) \sim \text{Gumbel}(0, T)$ effectively samples from the policy $p(a|I, O, P)$ (Kool et al., 2019). The Gumbel scale T controls the temperature with which the samples are taken. Following Danihelka et al. (2021), we select child nodes for the *interior* nodes by choosing the action that minimizes the difference between the improved policy $p'(a|I, O, P) = \text{softmax}(u(I, O, P, a) + \sigma(q(a)))$ and the normalized visit counts, resulting in the deterministic action selection rule

$$\arg\max_{a \in \mathcal{A}} \left(p'(a|I, O, P) - \frac{N(a)}{\sum_b 1 + N(b)} \right). \tag{3}$$

In the third step we assign a predicted value to leaf nodes that are not terminal based on the language model presented in Section 2.2. The prioritized sampling improves the training distribution for the policy, but comes at the cost of heavily biasing the outcome predictor's training targets to perfect outcomes. During the search process we require the language model to accurately evaluate outcomes even for incorrect or syntactically invalid programs. To counteract the bias introduced by the prioritized sampling, we populate a separate replay buffer with trajectories that are solely used to train the outcome predictor to avoid slowing down the policy training with noisy targets. We acquire these trajectories by sampling from the policy $p(a_i|I, O, P)$ with an ϵ-greedy sampler. This guarantees a wide coverage of states (partial programs) while still favoring those that are likely to be encountered during the MCTS search.

Note that the dichotomy between the optimal training distribution for the policy and the outcome predictor exist even without the prioritized replay buffer. When the MCTS policy is optimal, then the training distribution would primarily consist of trajectories with perfect outcomes. This can deteriorate the accuracy of the outcome predictor, which in turn deteriorates the performance of the MCTS policy. By employing different training distributions for the policy and outcome, we avoid instabilities at and near the optimum.

3 EXPERIMENTS

We carry out an extensive set of experiments to investigate the feasibility of a self-improving programming agent combining language models and powerful search. Following (Jones, 2021), we plot all of our results in terms of performance, measured as the percentage of solved tasks, against expended compute resources, measured in FLOPs.

We seek to answer the following questions:

Search methods How do the search methods, MCTS and Sampling and Filtering, differ? Should one be preferred?

Compute management How should the compute be divided between search and training? Is it e.g. possible to compensate for a reduction in search budget via an increase in training time?

Curriculum Is a *curriculum* necessary to make ExIt-type methods work?

Learning The LLM can learn the policy, the value function, or both together. Do these choices differ in how well they can be learned and how helpful they are to guide MCTS?

3.1 SETUP

Assembly language The "machine" we choose to program is a modified version of the Little Man Computer, [1] a simple Instruction Set Architecture (ISA) meant for educational purposes. It is an example of an *accumulator machine*, where one operand of each operation (e.g., addition) is always understood to be a single "accumulator" register; as a result of this, all operations are unary. We shall see that despite its simplicity, programming-by-examples on this ISA is difficult. We describe the ISA in detail in appendix A.

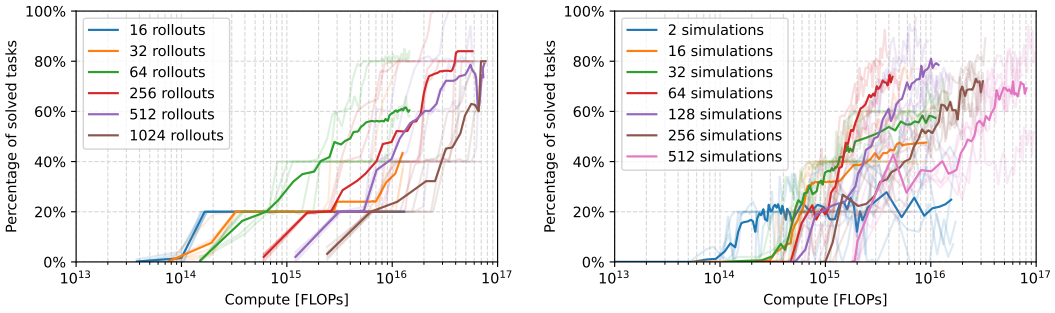
Tasks Each "task" we consider is parameterized by a list of input-output pairs. The inputs can be integers or lists of integers, whilst outputs are single integers. For example, an input for the task of adding would be a list containing the two addenda, e.g., `[5,4]`, and the corresponding output would be the single integer `9`. We provide multiple input-output pairs for each task, and we make a distinction between *training* pairs, which are part of the task description used to prompt the LLM, and *test* pairs, which are used to compute rewards and evaluate performance. By using different train/test pairs, we can check whether the agent "cheats" by copying the output values given in the prompt into the generated code.

We consider the following two datasets, each capturing a different distribution of tasks:

Add-N. Given a list of inputs, the corresponding output is their sum. The length of the list can vary from 2 to N, and the longer the list, the more difficult the task. We provide 3 training and 3 test I/O pairs per task.

Random Programs. We generate programs, and their associated inputs, at random. Because our assembly language is regular, we can generate one operation at a time while guaranteeing syntactic correctness. We set the program length according to the number of inputs, so that the space over which we sample programs includes those that intervene on all input elements. We provide 4 I/O pairs for training and as many for testing. More details on the generation algorithm can be found in appendix B.

[1] See e.g. `https://elearning.algonquincollege.com/coursemat/dat2343/lectures.f03/12-LMC.htm`



(a) Sampling and Filtering

(b) MCTS

Figure 3: Percentage of add-N tasks solved against total compute. For different balances of search budget. Searching more yields higher performance in the long run.

The Add-N dataset assesses the ability to solve progressively more difficult, but related tasks. Given an N, the dataset effectively contains only $N-1$ different tasks, independently of how many input-output pairs are contained in it. And a correct program for adding N inputs contains as a sub-program one to add $N-1$ inputs, for any $N > 2$. In contrast, the random programs dataset benchmarks the ability to generalize between qualitatively distinct tasks: each input-output pair parameterizes, in principle, a distinct task.

Tiny language model The backbone sequence model used in this work is based on the architecture of Llama 2 (Touvron et al., 2023). We explore self-improvement dynamics, initializing the model with randomly initialized weights. To make experimentation feasible on the hardware available to us, we scale down the model to a modest size of 3.6 million parameters. This reduction is achieved by limiting the hidden dimension to 256 and using only 4 heads and 4 transformer layers. Emphasizing simplicity, we use an UTF-8 tokenizer as proposed by Xue et al. (2022).

3.2 ExIt climbs the programming difficulty ladder

Figure 2 shows evidence, in our setting, of the complementarity between System 1 and 2 that we argue for in Section 1, and serves as a justification to use ExIt for code generation.

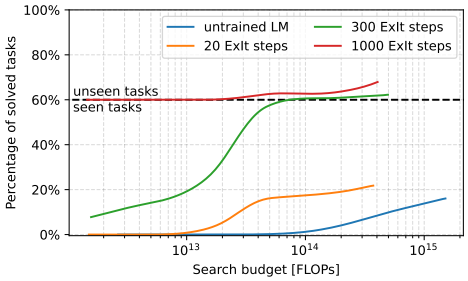


Figure 2: Performance on Add-6 against search budget, at different stages of ExIt training on Add-4. It shows we need both System-1 intuition via knowledge distilled in our LM, and System-2 thinking via search to start solving unseen and difficult tasks, enabling ExIt to climb the programming difficulty ladder.

3.3 Comparing search methods

Figures 4 and 3 show our main result: the performance of our two variants of ExIt (MCTS and S&F) on the Random Program and Add-6 families of tasks, respectively.

For the random programs synthesis task, S&F shows a consistent increase in task completion percentage at bigger compute budget with increasing rollouts, indicating a positive correlation between the number of rollouts and the method's effectiveness. The rate of improvement is steeper for smaller number of rollouts at smaller compute budgets. However, the performance at higher compute budget is positively impacted when one increases the number of rollouts. A notable improvement plateau starts at around 2×10^{16} FLOPs. Conversely, MCTS demonstrates more variability in task performance, especially when using fewer simulations. However, with a sufficiently large number of simulations, MCTS reaches a performance comparable to S&F, albeit with greater variance across the compute spectrum.

In the add6 task, both S&F and MCTS exhibit steep performance improvements with increased computational budget. For S&F, the performance jumps significantly between 16 to 256 rollouts but then levels off between 512 to 1024 rollouts, suggesting a value of 256 rollouts to be optimal. That is in contrast to the results on random program synthesis, where the number of rollouts always correlated positively with performance. A similar trend is present in the MCTS case, where again intermediate values of the number of rollouts seem optimal in achieving highest performance. We remark that the convergence behaviour is erratic, with notable disparities in effectiveness between different search budgets.

Given these observations, S&F could be preferred when a stable and predictable performance is paramount, particularly at higher computational budgets. MCTS might be favored when the computational budget is restricted since it can occasionally match or outperform S&F with fewer simulations and lower overall compute. The choice of method also depends on the specific task; for the add6 task, both methods perform similarly at higher compute levels, with MCTS showing a steeper performance increase. For random programs, S&F shows a slight edge.

The presented data suggest that the optimal search method and configuration (in terms of rollouts or simulations) should be chosen based on the particular requirements of the task and the available compute budget. While S&F tends towards stability, MCTS has the potential for high performance but with greater uncertainty.

3.4 SEARCHING–LEARNING TRADEOFF

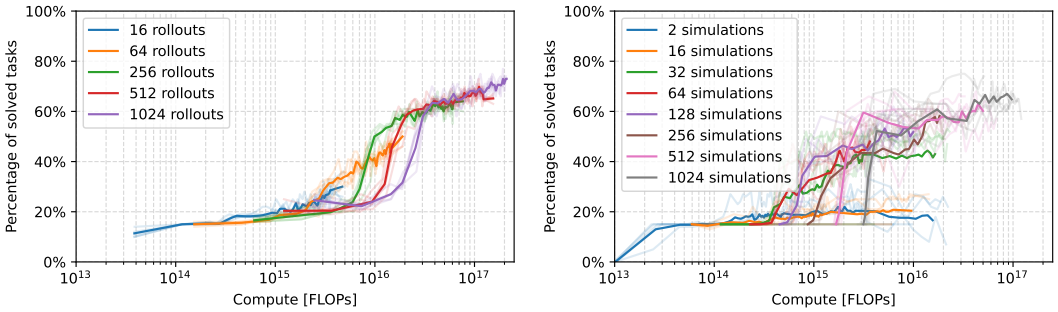
Here we address the question of how compute should be divided between the search and training components of Expert Iteration; this question is relevant if one has a pre-determined compute budget to expend, e.g. a certain amount of paid-for compute hours on a cloud platform. Carrying out a smaller amount of search per meta-iteration can allow one to pack more gradient updates into this budget, and vice-versa. For both families of tasks, our results show that a minimal number of simulations (for MCTS) and a minimal number of rollouts (for S&F) are needed for performance to increase reliably as training progresses; below that threshold, performance quickly plateaus and expending more compute does no longer lead to improvement. Above the threshold, one can observe a difference in the behavior of ExpIt between the Add-N family and the Random Programs one. In the first case (Figure 3), there seems to be an optimal amount of search whereupon the highest speed of learning (as a function of FLOPs) is attained; at this optimum, one is making the best use of the available compute budget, by maximizing the amount of improvement per expended FLOP. In the case of the Random Programs task family we consider in figure 4, this effect is not visible, and no clear optimum exists.

In both cases, we can observe that the amount of performance obtained once a certain number of FLOPs is attained can depend strongly on the search budget, and sometimes in a counter intuitive way: for example, in Figure 3b, one can observe that a practitioner with 4×10^{16} FLOPs to expend is better off setting the number of simulations to 64 rather than 512, despite the intuition that more simulation should lead to more improvement and higher speed of learning. A complementary question concerns *asymptotic* performance, i.e. what is the best budgeting of train/search compute when the number of available FLOPs is unlimited, or anyway very large. From our results, it does not appear that the level of asymptotically attainable performance be strongly impacted by the search budget, once one sets it above the minimal number discussed above. Providing a conclusive empirical answer to this question is however beyond the scope of this work.

3.5 COMPARING TASK DISTRIBUTIONS

In the previous sections, we observed a qualitative difference in the behavior of performance as a function of search budget, for both methods, between the two families of tasks: in the first case, an optimal number of rollouts and simulations exists, beyond which performance stops increasing with training compute, or even starts to deteriorate; in the second case, increasing the number of simulations invariably benefits final performance (though the improvement appears to eventually saturate), and no clear optimal value can be established.

The behavior observed in the Add-N case is the one commonly observed in successful applications of ExIt, see e.g. Jones (2021), and is a sign that the priors provided by the LM are successful in guiding



(a) Sampling and Filtering

(b) MCTS

Figure 4: Percentage of random programs solved against total compute. For different balances of search budget. Searching more yields higher performance in the long run.

the search: while the search tree grows exponentially with the length of the program, only a finite number of well-focused simulations are necessary to improve on the NN policy during the search stage. We speculate this to be due to the fact that the Add-N family naturally defines a *curriculum*: the correct program to add two inputs, `ADD; 0; HLT`, provides a good prior for the solution of the addition of three inputs, `ADD; 0; ADD; 1; HLT`; the one for three inputs provides a prior for the addition of four of them, and so on. The agent can effectively and always reuse the knowledge distilled in the LM to find the solution to the subsequent task. Is had been speculated, e.g. in (Polu et al., 2022), that the spectacular success of ExIt in the context of two-player games (such as chess and go) be indeed due to the self-play setup effectively establishing a curriculum, in which the agent only has to beat an opponent with a similar level of proficiency. While this has not been rigorously proven, our results appear to corroborate this point of view.

In the case of the random programs family, no obvious curriculum exists: while not all tasks have the same difficulty, the easier ones do not necessarily provide a useful prior for the more challenging ones. An important question is whether a finite simulation budget is sufficient to eventually achieve perfect performance and solve all tasks, or whether final performance will tend asymptotically to 100% as the number of simulations grows. Our runs are not long enough to provide a solid answer to this question, which we leave for future work.

3.6 Learning actions versus values

During search we can use two different approaches to judge how to start or continue a (partial) program: a policy that specifies the distribution over the next operation, or a value function that assess the expected reward when continuing with a certain operation. An optimal value function allows us to recover the optimal policy by assigning a probability of one to the action with the maximum value. In practice learning the optimal value function is often infeasible. Furthermore, the policy function and the value function can differ in how they generalize to new states. We assess the effectiveness of each individually and the combination of both. In Figure 5, we observe that using both in combination works best.

4 Related work

Program synthesis with language models Haluptzok et al. (2022) showed that LMs can learn to improve their performance using self-play on programming puzzles. They leverage problems that are specified formally as programming puzzles Schuster et al. (2021), a code-based problem format where solutions can easily be verified for correctness by execution, similar to our setting. Similarly to Haluptzok et al. (2022) we use the language policy to generate the solution and then filter for correctness using an interpreter. However, in that work inputs are proposed by the language model, whereas we sample the inputs from a pre-defined input distribution. They also only investigated temperature sampling and did not consider more advanced search techniques.

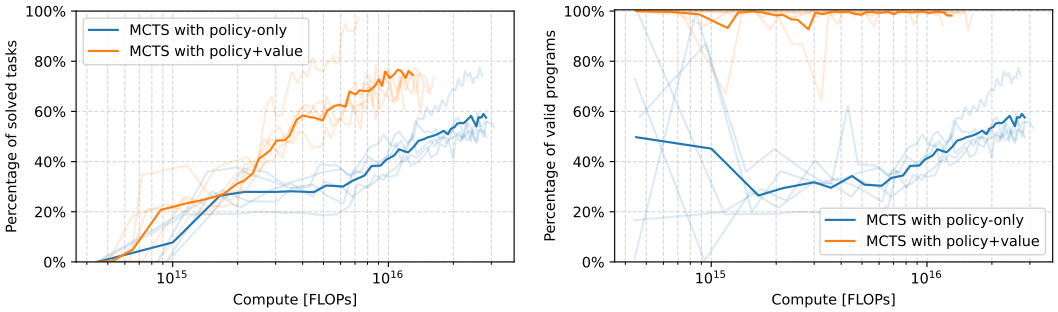


Figure 5: Learning actions versus values. We compare the effect of turning off either aspect on the performance of ExIt. Performance is measured as the percentage of solved tasks (left) and valid programs (right), over total compute used in FLOPs.

It was shown in Zhang et al. (2023) that an MCTS-aided decoding method can boost performance of pre-trained language models on code generation tasks. The proposed MCTS-based extensions differ from the one investigated in our work in several ways. Firstly, we employ Gumbel-MCTS Danihelka et al. (2021), a variant more suitable in settings with low simulation budget. Secondly, we do not use a rollout policy to estimate the value of a state but rather learn the associated value function. Lastly, we focus on from-scratch training and expert iteration dynamics rather than proposing a new decoding method for pre-trained foundation models.

Self-improvement methods Self-improvement methods involve training neural agents or systems on their outputs, offering an alternative to reliance on high-quality datasets. These methods have primarily been applied to policy improvement, reasoning, and planning within Reinforcement Learning contexts, exemplified by works such as (Anthony et al., 2017a; Silver et al., 2018; Schrittwieser et al., 2020; Mankowitz et al., 2023). All neural agents demonstrating genuine planning and reasoning, notably AlphaZero (Silver et al., 2018) and MuZero (Schrittwieser et al., 2020), employ self-improvement strategies.

Our work, aiming at exploring planning and reasoning, adopts a search algorithm akin to AlphaZero's but diverges by focusing on programming tasks outside the two-player game paradigm and not utilizing self-play. Unlike the game-specific approaches of (Silver et al., 2018) and (Schrittwieser et al., 2020), our agent is designed to solve multiple programming-by-example tasks through in-context specifications, leveraging Language Models (LMs). This approach, similar to Polu et al. (2022) in its single-player setting and LM reliance, also innovates in value and outcome prediction strategies. Additionally, our ExIt variation, resembling the method in (Gulcehre et al., 2023) for Machine Translation, differentiates itself through a heuristic reward function.

Our goal aligns with (Silver et al., 2018), (Schrittwieser et al., 2020), and (Mankowitz et al., 2023) in training agents from scratch without human demonstrations. Notably, (Mankowitz et al., 2023) also targets programming in assembly language, though focusing on a different language and a singular task. The work of (Jones, 2021) significantly influences our exploration of design choices and computational requirements for building self-improving intelligent agents.

5 Conclusion

We have shown that it is possible to learn to program progressively longer assembly programs from scratch through expert iteration. In an extensive empirical study we examined how to efficiently distribute the compute onto the different stages of expert iteration and how to adapt expert iteration to the challenging setting of code generation without any supervision. We observe that both search and training are essential to the success of expert iteration. Our empirical study was performed on simple datasets that allowed full control on the type of generalization that is assessed by the dataset. These datasets are limited in scope and it will be interesting to see how our findings generalize to other settings.

REFERENCES

Ferran Alet, Javier Lopez-Contreras, James Koppel, Maxwell Nye, Armando Solar-Lezama, Tomas Lozano-Perez, Leslie Kaelbling, and Joshua Tenenbaum. A large-scale benchmark for few-shot program induction and synthesis. In *International Conference on Machine Learning*, pp. 175–186. PMLR, 2021.

Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. *Advances in neural information processing systems*, 30, 2017a.

Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. *Advances in neural information processing systems*, 30, 2017b.

Xinyun Chen, Petros Maniatis, Rishabh Singh, Charles Sutton, Hanjun Dai, Max Lin, and Denny Zhou. Spreadsheetcoder: Formula prediction from semi-structured context. In *International Conference on Machine Learning*, pp. 1661–1672. PMLR, 2021.

Ivo Danihelka, Arthur Guez, Julian Schrittwieser, and David Silver. Policy improvement by planning with gumbel. In *International Conference on Learning Representations*, 2021.

Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, et al. Reinforced self-training (rest) for language modeling. *arXiv preprint arXiv:2308.08998*, 2023.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. Language models can teach themselves to program better. *arXiv preprint arXiv:2207.14502*, 2022.

Andy L. Jones. Scaling Scaling Laws with Board Games. *arXiv e-prints*, art. arXiv:2104.03113, April 2021. doi: 10.48550/arXiv.2104.03113.

Daniel Kahneman. *Thinking, fast and slow*. 2017.

Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *International conference on machine learning*, pp. 1238–1246. PMLR, 2013.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.

Wouter Kool, Herke Van Hoof, and Max Welling. Stochastic beams and where to find them: The gumbel-top-k trick for sampling sequences without replacement. In *International Conference on Machine Learning*, pp. 3499–3508. PMLR, 2019.

Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal Mathematics Statement Curriculum Learning. *arXiv e-prints*, art. arXiv:2202.01344, February 2022. doi: 10.48550/arXiv.2202.01344.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

Rajarshi Roy, Jonathan Raiman, Neel Kant, Ilyas Elkin, Robert Kirby, Michael Siu, Stuart Oberman, Saad Godil, and Bryan Catanzaro. Prefixrl: Optimization of parallel prefix circuits using deep reinforcement learning. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 853–858. IEEE, 2021.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

Tal Schuster, Ashwin Kalyan, Oleksandr Polozov, and Adam Tauman Kalai. Programming puzzles. *arXiv preprint arXiv:2106.05784*, 2021.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419): 1140–1144, 2018.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Saujas Vaduguru, Daniel Fried, and Yewen Pu. Generating pragmatic examples to train neural program synthesizers. *arXiv preprint arXiv:2311.05740*, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Linting Xue, Aditya Barua, Noah Constant, Rami Al-Rfou, Sharan Narang, Mihir Kale, Adam Roberts, and Colin Raffel. Byt5: Towards a token-free future with pre-trained byte-to-byte models. *Transactions of the Association for Computational Linguistics*, 10:291–306, 2022.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510*, 2023.

A MACHINE

ISA We program in an ISA based on the Little Man Computer (LMC) devised by Stuart Madnick [2]. We make the following modifications to it:

- We remove the program counter and the need to load the opcodes in memory (i.e. the assembly process). All branching instructions use line numbers rather than memory addresses.
- We reduce the size of the memory from 100 to 10 locations.
- Memory locations can contain any `int32` integer, and are not limited to numbers from 0 to 99 like in the original LMC.
- We prune the `INP` and `OUT` instructions. The inputs and outputs are loaded to, and read from, the machine by the environment, as we outline below.
- We add instructions for loading, adding and subtracting *literal* values, as opposed to memory addresses.

We report in table 1 the full ISA, with a description of each opcode.

Instruction	Description
`ADD`	Add content of subsequent memory address to accumulator
`SUB`	Subtract content of subsequent memory address from accumulator
`ADDL`	Add subsequent literal value to accumulator
`SUBL`	Subtract subsequent literal value from accumulator
`LDA`	Load content of subsequent memory address into accumulator
`LDL`	Load subsequent literal value into accumulator
`STA`	Store content of accumulator to subsequent memory address
`BRA`	Branch to subsequent line number
`BRZ`	If accumulator is zero, branch to subsequent line number
`BRP`	If accumulator is positive, branch to subsequent line number
`HLT`	Terminates the program.

Table 1: The full set of instructions available to our programmer.

For example, the program

```
LDL
1
STA
0
LDL
10
SUB
0
BRP
7
HLT
```

loads a literal one into the accumulator and then stores it into address zero, then loads a literal ten into the accumulator, and subtracts from it the content of memory address zero (i.e. a value of one) until the accumulator is zero. At that point the program halts.

Runtime A program like the one shown above can be easily executed sequentially, with each line counting as a single time step. Due to the presence of branching instructions, it is theoretically possible to write programs whose execution never terminates. A very simple example could be the program

[2] `https://elearning.algonquincollege.com/coursemat/dat2343/lectures.f03/12-LMC.htm`

```
ADDL
1
BRZ
7
BRA
1
HLT,
```

which will get stuck in an infinite loop unless the first element of the input array happens to be negative. In order to avoid such situations, we set a maximal number of executions steps $t_{\max} = 100$, above which the program is killed and a Runtime Error is produced.

Input/output management As mentioned above, we choose to have I/O operations performed by the environment directly. We remind the reader that in our problem specification inputs can be lists of integers, whilst outputs can only be single integers. When a program is ready to be run and has passed a preliminary syntax check step, the first element of the input list is loaded into the accumulator, and the subsequent ones are loaded into memory locations in increasing order. Once the program has finished running, the only output is read directly from the accumulator.

For example, given the set of 3 IO examples: `input=[[3,5],[4,8],[10,1]]output=[8,12,11]`, the program

```
ADD
0
HLT
```

is a perfect solution to its associated task (addition of the two inputs).

B ALGORITHM FOR GENERATION OF RANDOM ASSEMBLY PROGRAMS

We describe here the simple algorithm we implemented for generating random assembly programs together with their I/O specifications. The data so obtained can be used for two purposes: the programs can be used to pretrain the LLM before ExIt, and/or their I/O specifications can be used as the set of task to solve during ExIt; this provides us with a set of tasks alternative, and much more varied than, the add$N$ family.

Generating the inputs We start by generating a set of inputs. We do so by first generating a random integer between 2 and a pre-determined value $N_{\max}$ (both included), which will be the size of the input arrays. We then fill this input array with random integers between 1 and and maximal input value $x_{\max}$; we do so $2 * \mathcal{N}$ times, where $\mathcal{N}$ is the number of desired I/O pairs, and the factor of 2 is due to use generating both *training* I/O pairs, which function as task description and are used to prompt the LLM, and *testing* I/O pairs, which are not shown to the LLM and are used to compute the rewards.

Generating the program Once the inputs are ready, we proceed to generate the program. We simply alternate between picking an opcode from those in table 1, and picking an operand between 0 and a value corresponding to the length of the input array minus two, which corresponds to the memory address where the last element of the input array is loaded. We alternate these two actions for a number of times equal at least to the length of the input arrays minus one. This way, we have a reasonable chance that the program will in some way use all of the provided inputs; we do not however sample the operands without replacement, so the utilisation of all the inputs is not strictly required by the algorithm. Once the required number of operand/opcode pairs has been reached, the algorithm can take two actions: it can either, with probability ϵ, add one more pair to the program, or it can finally terminate it by adding the `HLT` instruction.

Checking and running the program Once the program is terminated, it is sent to the machine where, thanks to the simplicity of our ISA, it can be easily checked for syntax errors. If the program passes the check, it is finally run on the previously generated inputs. If the program halts and

successfully produces an output without generating runtime errors, then it is bundled together with its I/O pairs and added to the synthetic dataset. If the program fails the syntax check or produces a runtime error, it is discarded and a fresh one is generated. While this filtering procedure might appear wasteful, we find that programs are accepted often enough that the resulting dataset can be generated on the fly during each training run, without the need to store it. Of course, we saw to it that the procedure above is appropriately seeded, thereby ensuring that all our runs use the same, reproducible dataset. We report in table 2 the values of the parameters we used for the ExIt runs in the main text.

Parameter	**Value**
$N_{\max}$	8
$\mathcal{N}$	4
ϵ	0.001

Table 2: The values of the parameters of the random program generation algorithm defined in the text.

C HYPERPARAMETERS

C.1 SAMPLING AND FILTERING

To arrive at the hyper-parameters used in the main experiments we carried out small grid search on the learning rate, generation and evaluation temperatures. We found smaller learning rates to significantly reduce inter-seed variance. The generation temperature had a significantly more pronounced effect compared to the evaluation temperature, leading to ExIt saturating at lower rewards. The parameters used in the main experiments are presented in Table **??**.

Parameter	**Value**	
	Random Programs	Add 2-6
Learning Rate	5e-5	2e-4
Generation temperature	1.5	1.2
Evaluation temperature	1	0.8

Table 3: Parameter settings for Sampling & Filtering.