SHOW YOUR WORK: SCRATCHPADS FOR INTERMEDIATE COMPUTATION WITH LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large pre-trained language models perform remarkably well on tasks that can be done "in one pass", such as generating realistic text (Brown et al., 2020) or synthesizing computer programs (Chen et al., 2021; Austin et al., 2021). However, they struggle with tasks that require unbounded multi-step computation, such as adding integers (Brown et al., 2020) or *executing* programs (Austin et al., 2021). Surprisingly, we find that these same models are able to perform complex multi-step computations—even in the few-shot regime—when asked to perform the operation "step by step", showing the results of intermediate computations. In particular, we train Transformers to perform multi-step computations by asking them to emit intermediate computation steps into a "scratchpad". On a series of increasingly complex tasks ranging from long addition to the execution of arbitrary programs, we show that scratchpads dramatically improve the ability of language models to perform multi-step computations.

1 INTRODUCTION

Large Transformer-based language models exhibit surprisingly impressive capabilities (Devlin et al., 2019; Brown et al., 2020), including the ability to generate code that solves simple programming problems (Chen et al., 2021; Austin et al., 2021). However, these models struggle to perform multi-step algorithmic calculations, especially those that require precise reasoning and unbounded computation. For example, GPT-3 struggles to perform few-shot addition on numbers with greater than three digits (Brown et al., 2020). Similarly, large-scale language models struggle to predict the result of executing Python code, even code which is a solution to a programming task the model is able to solve (Austin et al., 2021). Likewise, standard recurrent and graph neural networks fail to systematically generalize when predicting the output of simple programs with loops (Bieber et al., 2020). So language models can to some extent *write* code, but do not seem to *understand* the code they write, because they cannot predict its execution. This has motivated research on networks that can perform algorithmic reasoning (Graves et al., 2014; Zaremba & Sutskever, 2014; Bieber et al., 2020). Neural networks that understand programs could enable a variety of downstream tasks, including program synthesis (Devlin et al., 2017), program analysis (Allamanis et al., 2018), and other algorithmic reasoning tasks (Velickovic & Blundell, 2021).

Why do large language models struggle with algorithmic reasoning tasks? We suggest that this is at least partly due to a limitation of the Transformer architecture and a limitation of task design: the model is asked to perform these tasks in one forward pass. Given a fixed number of layers and a fixed amount of computation time, the model cannot adapt the amount of compute spent on a problem to its difficulty before producing an output.[1] Prior work (Graves, 2016; Banino et al., 2021) has explored neural architectures that explicitly allow for dynamically chosen amounts of computation time to be dedicated to different sub-tasks. In this work, we propose a different approach—one that can exploit existing Transformer architectures and large few-shot-capable language models—we modify the *task design* rather than the model or training procedure.

[1]Transformers perform a computation which is quadratic in the length of the input sequence, so are theoretically unable to perfectly simulate algorithms which have greater time complexity than $O(n^2)$. However, it is unclear how relevant this theoretical bound is in practice; neural sequence prediction is approximate, and Transformers may be large enough in practice to effectively memorize the correct solutions for a relevant subspace of the possible inputs (e.g., all inputs up to a certain size).

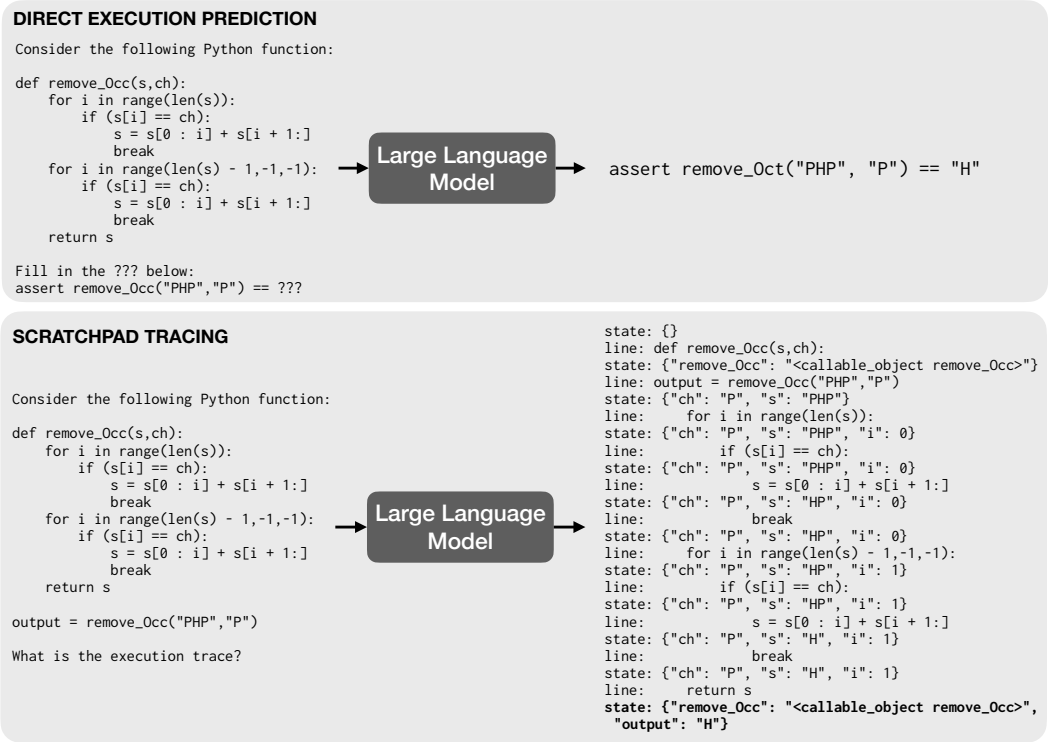

Figure 1: Overview of our scratchpad approach applied to predicting code execution and comparison to direct execution prediction. Top: Previous work has shown that large pre-trained models achieve poor performance when asked to directly predict the result of executing given computer code (Austin et al., 2021). Bottom: In this work, we show that training models to use a *scratchpad* and predict the program execution trace line-by-line can lead to large improvements in execution prediction performance.

Our proposal is simple: Allow the model to produce an arbitrary sequence of intermediate tokens, which we call a *scratchpad*, before producing the final answer. For example, on addition problems, the scratchpad contains the intermediate results from a standard long addition algorithm (see Figure 2). To train the model, we encode the intermediate steps of the algorithm as text and use standard supervised training.

This paper makes the following contributions:

- We introduce (Section 2) the notion of a "scratchpad" for Transformers, in order to make them better at performing complex discrete computations without modifying the underlying architecture.
- We show (Section 3) that scratchpads help Transformers learn to perform long addition in the fine-tuning regime, and in particular that they improve out-of-distribution generalization to larger problem instances.
- We also find (Section 4) that scratchpads help Transformers perform a somewhat higher level task: polynomial evaluation. This is true in both the few-shot and fine-tuning regimes.
- Finally, we move to a much more general context and show (Section 5) that training Transformers to emit full program traces line by line annotated with local variables dramatically improves their ability to predict the result of executing a given computer program on a particular input. This application in some sense subsumes the others.

2 METHOD

In this work we consider two related problems: algorithm induction (Graves et al., 2014; 2016; Kurach et al., 2016; Kaiser & Sutskever, 2016) and learning to execute (Zaremba & Sutskever,

2014; Bieber et al., 2020). The goal of both problems is for the neural network to learn to emulate a function f, which is "algorithmic" in the sense that it can be represented by a short program, such as addition or polynomial evaluation, from input-output behavior. In *neural algorithm induction*, the goal is to learn a single algorithm, and each training example gives a single input and desired output represented as strings. Therefore, the training data is $D = \{x_i, f(x_i)\}_{i=1}^N$. For *learning to execute*, we want the model to produce the result of a program, represented as source code, on some input. If each π_i is the source code of a program f_i, then the training data is $D = \{(\pi_i, x_i, f_i(x_i))\}_{i=1}^N$ (it is common for each f_i to have multiple input-output examples, but we omit this to lighten notation).

The main idea of this paper is that to solve a given algorithmic task, we simply encode the intermediate steps of the algorithm as text and train the model to emit them to a buffer that we call a "scratchpad." For example, let us consider the algorithmic induction task of learning long addition. To teach a model to add 29 to 57, a training example may look like the text in Figure 2, where the steps of the grade-school long addition algorithm are written out explicitly.

Learning to execute tasks can be encoded in a similar way, except now we add the source code π_i before the input, scratchpad, and desired output. An example of a training example for a learning to execute task is shown in Figure 1.

At training time, the model will be given the input plus target for standard likelihood-based training. At test time, the model will be given only the input and will be required to predict the target, e.g., by beam search or temperature sampling. In principle, any sequence model could be used for this. In this work, we choose to use decoder-only Transformer language models, but other sequence models could be effective, such as encoder-decoder models (Raffel et al., 2019), or recurrent networks.

```
Input:
2 9 + 5 7

Target:
<scratch>
2 9 + 5 7 ,  C: 0
2 + 5 , 6 C: 1  # added 9 + 7 = 6 carry 1
, 8 6 C: 0  # added 2 + 5 + 1 = 8 carry 0
0 8 6
</scratch>
8 6
```

Figure 2: Example of input and target for addition with a scratchpad. The carry is recorded in the digit following "C:". Comments (marked by #) are added for clarity and are not part of the target.

Adding a scratchpad has several potential advantages: First, the model has adaptive computation time, that is, it can now process the information for as long as needed, depending on the complexity of the task given the input. Second, the model can store the intermediate state of its computation in the scratch buffer and refer back to it by attending to its context. This removes the need to store all intermediate state in activations. Third, by forcing the model to output concrete intermediate states by sampling from the generative model, we aim to reduce the propagation and compounding of small errors, because states are quantized to token embeddings. Compounded errors can show up in methods—like Neural Turing Machines (Graves et al., 2014)—that use recurrence to support extended computations. Finally, examining a model's scratchpad output can help us identify common errors and correct them by revising the scratchpad format. We found this ability to interpret errors to be useful in this work.

For all experiments, we used pre-trained dense decoder-only Transformer language models, ranging in size from 2 million to around 100 billion parameters (with the largest denoted 100B+). These models were pre-trained on web documents and dialog data. We omit some details here—including exact model sizes—for double-blind review, but will include specific details in the published version.

3 ADDITION

As a first task, we consider integer addition. The baseline addition task presents two numbers as the input, and the target is their sum. For example:[2]

```
Input:  2 9 + 5 7
Target: 8 6
```

[2]We introduce spaces between the digits to ensure that each digit is mapped to a separate token.

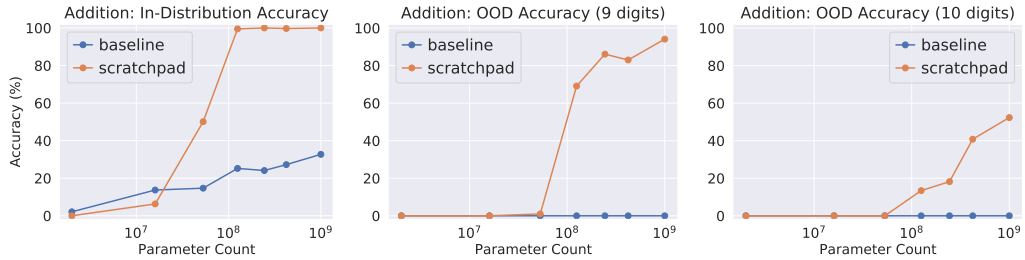

Figure 3: Using a scratchpad significantly improves the performance of pre-trained Transformer-based models on addition, including their ability to generalize out of the training distribution to numbers with more digits. Models were trained on 1-8 digit addition. The baseline models were trained without intermediate scratchpad steps.

We implement the scratchpad by including the intermediate steps of the long addition algorithm in the target, as in Figure 2. We train several models on integer addition problems with inputs that have 1-8 digits. We then test performance on in-distribution addition problems (with up to 8 digit inputs), and on out-of-distribution problems with 9 and 10 digit inputs. The models were fine-tuned on 100k examples for 5k steps with batch size 32. There are 10k in-distribution test examples, and 1k test examples for each out-of-distribution task. We examine the performance as a function of model size, ranging from 2M to 1B parameters. We compare performance to a baseline which includes the input and target numbers, but no intermediate scratchpad steps.

Results Figure 3 compares the performance of the scratchpad algorithm with the baseline. We see that beyond a critical model size, models are able to solve the addition task using the scratchpad, while models trained without a scratchpad fail to do so even at the largest tested scale. On the out-of-distribution tasks (9-10 digit addition), we find that models trained without scratchpad completely fail, while models trained with scratchpad show consistent improvement as a function of model size.

4 Polynomial Evaluation

Next we focus on a slightly higher-level task: evaluating polynomials. Inspired by the "polynomial evaluation" subproblem in Saxton et al. (2019), we generate a dataset of polynomials of degree less than or equal to three, with integer coefficients and inputs constrained to the range $[-10, 10]$. We also restrict outputs to the range $[-1000, 1000]$. We generate a training dataset of 10,000 polynomials and a test dataset of size 2,000. An example scratchpad target for this task is shown in Figure 4, with each term of the polynomial evaluated separately. As in the previous section, we compare the results of direct execution with the results of using the scratchpad. In this experiment, we evaluate in the few-shot regime using a 100B+ parameter pre-trained decoder-only model, as previous work indicates that very large models may be able to perform additions and multiplications with 3 or fewer digits few-shot (Brown et al., 2020). We use $n = 4$ example problems in the few-shot prompt. We also evaluate in the fine-tuning regime with an 8B parameter model fine-tuned for 2000 steps on the training set. The results of both evaluations are shown in Table 1. We find that scratchpad execution outperforms direct execution significantly in both the few-shot and fine-tuning regimes.

5 Executing Python Programs

We have shown that scratchpads can help algorithm induction, that is, they can help models learn to implement a particular algorithm with direct algorithm-specific supervision. But needing to hand-design the intermediate states for every new task is sub-optimal. In this section, we evaluate whether a model can learn to implement a new algorithm by executing arbitrary code. To test this capability, we follow the problem setup from Austin et al. (2021), in which language models are asked to predict the result of executing a given Python program on a particular input. Language models performed poorly at this task, even on programs which are solutions to a programming tasks the model is able to solve. Here we show that the scratchpad technique can dramatically improve the ability of language models to execute programs.

```
Input:
Evaluate -7*x**2 + 7*x + 5 at x = 1

Target:
<scratch>
-7*x**2: -7
7*x: 7
5: 5
</scratch>
total: 5
```

Figure 4: Example of polynomial evaluation with a scratchpad. Each term in the polynomial is computed separately and then added.

Table 1: Results for polynomial evaluation task. Scratchpad outperforms direct prediction whether using fine-tuning or few-shot.

	Few-shot	Fine-tuning
Direct prediction	8.8%	31.8%
Scratchpad	**20.1%**	**50.7%**

Direct execution prediction Our main baseline is the direct execution prediction procedure explored in Austin et al. (2021). Models are shown the source code for a function, and asked to predict the output of running the function on specific inputs. For example, the function in Figure 1 takes as input a string `s` and a character `ch`, and removes the first and last instances of the character `ch` from the string `s`. The direct execution prompt and target for this task are shown in the "Direct Execution Prediction" box in Figure 1. A task is considered solved under this regime if the model correctly outputs the target string.

Execution prediction via scratchpad tracing As discussed above, direct execution prediction requires the model to correctly output the result of executing the entire function in a single pass. Direct execution prediction has been shown to perform poorly on Python programs in Austin et al. (2021). We therefore design a scratchpad formulation of the execution task, in which models predict the output of a program by first predicting the sequence of intermediate states computed during the program's execution. Formally, we train models to predict an alternating sequence of 1) the ordered sequence of source code lines executed, and 2) the state of the local variables after each line is executed. We call this object the program's **trace**, and it allows us to track both the control flow—the sequence of operations executed—and how the state changes as a result of each operation. We represent the trace as a string, with the line of code reproduced directly, and the state information represented as a JSON dictionary.[3] For example, the "Scratchpad Tracing" box in Figure 1 contains the tracing prompt and trace target for the function discussed above.

Concretely, for each function to be traced, the prompt is formed by printing the function definition, followed by a line which calls the function on a particular input: `output = fn_name(input_value)`, where `fn_name` and `input_value` are replaced with the corresponding function name and input value. In Figure 1, note how the correct output of `remove_Occ("PHP","P")` is shown in the last line of the trace, assigned to the variable `"output"`. A tracing example is considered to have the correct execution output if the encoding of the value assigned to the variable `output` in the last line is a semantic match with the target output value (here, `"output": "P"`). We consider a *task* to be executed correctly if all given input-output examples are correctly executed. We can also test whether there is a "trace exact match" between the model prediction and the ground truth trace, by a) semantically comparing each state in the trace to the corresponding state in the ground truth trace, and b) comparing the sequence of source code lines predicted with the ground truth sequence.

Experimental setup As a proof-of-concept, we first show that scratchpad tracing greatly improves execution performance on synthetic Python data. Then, we compare scratchpad tracing and execution on the human-written Python problems from Austin et al. (2021). We find that a novel data augmentation technique that uses programs generated by the model as additional training data can significantly increase tracing performance on real data, whereas this augmentation technique hurts performance for direct execution. Finally, we show that incorporating tracing data from additional sources further improves tracing performance, indicating that the scratchpad tracing technique explored here may scale well with more data.

[3]Some objects cannot be represented using this JSON representation. Some objects (such as tuples) are represented by the closest JSON data type (in this case, lists). Other objects, such as user-constructed objects, are represented by a placeholder string, e.g., `"<object myObject>"`. Functions are also represented as strings, e.g., `"<callable_object f>"`.

```
def f(v0):
  v0 += 0
  v4 = 2
  while v4 > 0:
    v4 -= 1
    v0 *= 2
  return v0

output = f(8)
```

Figure 5: Example synthetic Python program.

Table 2: Synthetic tracing and execution results. Scratchpad outperforms direct prediction both for few-shot and fine-tuned.

	Few-shot	Fine-tuned
Direct prediction	11%	20%
Scratchpad	26.5%*	**41.5%**

For all experiments on Python code, we use a Transformer model with around 100 billion parameters (100B+), a context window of 1024 tokens and a limit of 512 generation tokens. Unless otherwise stated, all fine-tuning runs used a batch size of 8192 tokens and a learning rate of 3e-5, and model inference was performed with decoding temperature set to $T = 0$, equivalent to greedy decoding.

5.1 Scratchpad Beats Direct Execution for Synthetic Python Programs

In our first experiment, we test the few-shot and fine-tuned execution capabilities of our models on simple synthetic Python programs. This provides a proof-of-concept for our tracing technique.

We use a dataset of synthetic Python programs modified from Bieber et al. (2020). These programs include small integers (0, 1, and 2), simple `while` loops, and `if` statements. We construct a corpus of synthetic programs to mimic the size of the MBPP dataset in Austin et al. (2021), with 400 training programs, 100 validation programs, and 200 test programs. For each program, three random integer inputs are sampled from the range 0 to 9.

We test execution and scratchpad tracing under both few-shot and fine-tuning conditions. For few-shot experiments, the prompt contains three examples of previous tracing problems, as shown in Appendix A. For fine-tuned experiments, we fine-tune models to convergence on the training split, as judged by validation perplexity.

For the few-shot scratchpad experiment, we noticed that models would not assign the variable name `output` to the final value in the trace, and would instead continue using `v0` (the name of the variable returned in the function `f`) as the variable name for the final output line. We therefore modified the accuracy criterion from checking whether the value of `output` in the last line of the trace is correct, to checking whether the value of `v0` is correct. (Under naive scoring, the few-shot tracing accuracy is roughly zero.) An example of this behavior is shown in Appendix B.

Results Table 2 shows our results on synthetic Python problems. In both few-shot and fine-tuned settings, the scratchpad tracing technique leads to higher overall execution accuracy on the 200 test problems. Fine-tuning also improves performance more for the scratchpad tracing technique than it does for direct execution.

5.2 Scratchpad Beats Direct Execution for Real Programs

In our second set of experiments, we explore how well the scratchpad performs compared to execution on real data. Our main evaluation dataset is the MBPP dataset, introduced in Austin et al. (2021). MBPP consists of 1000 programming problems, each of which contains a natural language specification, a ground-truth Python program, and three input-output test cases. These programs involve computation using a large variety of types, including ints, strings, floats, dictionaries, tuples, and more, and include many language features and control-flow structures, such as loops, comprehensions, library imports, API calls and recursion. The evaluation split of the MBPP dataset contains 500 tasks. In order to separate out effects of the generation window size, we report all evaluation metrics on the subset of these tasks for which the ground-truth trace fits within the generation window of the model for all three of the input-output examples. This leaves a subset of 212 test tasks. Increasing generation and context window length is an important issue for Transformer-based models, but we view it as orthogonal and leave it for future work.

Table 3: Comparison of models fine-tuned on different data sets and evaluated on MBPP programs. We report "per-task" execution and tracing accuracies, which require all examples to be correctly executed/traced. We additionally report "per-example" accuracies, which correspond to the total fraction of test examples which are executed/traced correctly across the dataset. We find that training scratchpad models on an dataset augmented with samples from the model significantly improves performance for the scratchpad model, while it harms the direct execution model. Combining tracing training data from several sources further improves scratchpad model performance.

	Direct execution		Scratchpad				
	MBPP (§5.2.1)	MBPP-aug (§5.2.2)	MBPP (§5.2.1)	MBPP-aug (§5.2.2)	MBPP-aug +CodeNet +single line (§5.3)	MBPP-aug +CodeNet (§5.3)	MBPP-aug +single line (§5.3)
per-task execution acc:	10.3	5.1	5.1	17.3	**26.6**	25.2	23.4
per-task trace acc:	n/a	n/a	0.9	13.1	**24.6**	22.0	21.5
per-example execution acc:	22.0	12.3	24.6	35.5	**46.0**	45.3	43.5
per-example trace acc:	n/a	n/a	6.7	26.8	41.9	**42.1**	40.2

5.2.1 Performance is Poor in the Very-Low-Data Regime

In our first experiment with the MBPP data, we train a scratchpad tracing model on the 374 training tasks (3 examples per task, so 1122 overall examples). We discard all training examples which exceed the context window. We compare overall execution results against a model trained on the same 374 training tasks to perform direct execution. The columns labeled "MBPP" for Direct Execution and Scratchpad in Table 3 show the results of this experiment. Neither the scratchpad model or the direct execution model achieve good performance (5% and 10% output accuracy, respectively), and direct execution outperforms the scratchpad model.

5.2.2 Sampled Programs Make Good Scratchpad Training Data

Next, we employ a data augmentation technique to increase the size of the training dataset: We first run few-shot synthesis on the 374 MBPP training tasks using the pre-trained 100B+ model, as described in Austin et al. (2021). For each task, we sample and record 80 candidate programs $\{P_s\}$ from the model at temperature $T = 0.5$. We can then create a new execution datapoint using the candidate program P_s, the original three inputs for the task $\{x_i\}_{i=1,2,3}$, and the three new outputs which result from running the candidate program on the original three inputs: $\{y_{i_{\text{new}}}\}_{i=1,2,3}$, where $y_{i_{\text{new}}} = P_s(x_i)$. We discard any candidate programs for which execution results in an error. Note that the outputs of $y_{i_{\text{new}}}$ may or may not be equal to the original outputs, depending on the computation performed by the generated program P_s. Therefore, this augmented direct execution dataset has both additional new programs and new outputs compared to the original dataset. We can analogously create a tracing dataset for our scratchpad model by tracing the execution of each candidate program P_s on each x_i. This process produces much larger tracing and execution datasets with 17k new programs, which we refer to as **MBPP-aug**.

Conceptually, we have augmented the dataset using a combination of tools already available to us, namely a) the neural model, and b) program execution via a Python interpreter. We fine-tune direct execution and scratchpad models on this new augmented dataset MBPP-aug, using the same process as above.

The "MBPP-aug" columns in Table 3 show the results of this experiment. While the direct execution approach suffers a decrease in accuracy when trained on this additional data, the performance of the scratchpad model is greatly improved; the model trained on the augmented data solves more than three times the number of tasks as the model trained on only the original MBPP programs. We also note that if we measure the raw correctness across samples, the model already achieves 26.8% exact trace match, which is surprisingly high.

5.3 Scratchpad Training Makes Good Use of Large Datasets

```
state: b = 15; code: b = b // 2; output: b = 7;

state: g = 100; i = 1; l = [100, 100, 0, 0, -100, -100];
    code: g += l[i]; output: g = 200; i = 1; l = [100, 100, 0, 0, -100, -100];

state: s = 'aabbcd'; code: o = set(s); output: o = {'a', 'b', 'c', 'd'}; s = 'aabbcd';

state: f = 63; i = 11; j = 53; code: f = i ^ j; output: f = 62; i = 11; j = 53;
```

```
a, b, x = map(int, input().split())
if a // 2 < b:
    if x % 1000 == 0:
        print(a*(x//1000))
    else:
        if (x % 1000) / 500 > 1:
            print(min(a*(x//1000 + 1), a*(x//1000) + b*2))
        else:
            print(min(a*(x//1000 + 1), a*(x//1000) + b))
else:
    if x % 500 == 0:
        print(b*(x//500))
    else:
        print(b*(x//500 + 1))
```

Figure 6: Top: examples of single line data. Bottom: example CodeNet submission.

In this section, we examine whether collecting additional tracing data from human-written programs further improves tracing performance. This will allow us to understand whether the tracing procedure here is likely to scale well when slightly out-of-distribution tracing data is added to the fine-tuning set. We experiment using two datasets:

Single-line programs This dataset consists of roughly 9 million examples of single-line Python transformations. Figure 6 (Top) shows examples of these transformations. Each transformation consists of an initial set of variables and corresponding values, a single line of Python (together these form the input), and the new set of variables and values which results from running the line (the target). When training on single-line data, we do not introduce intermediate scratchpad steps. While this dataset does not provide examples of the high-level, multi-line control flow of a trace, the data provides good supervision for modeling the execution of individual lines of code, which is a key component of tracing. This data was collected by Fraser Greenlee, and can be accessed here.

CodeNet The Project CodeNet dataset (Puri et al., 2021) consists of millions of user submissions to approximately 4,000 coding problems. These submission include both correct and incorrect solutions to programming problems. However, from the experiment with MBPP-aug above, we know that incorrect or broken programs can still provide a useful training signal. We additionally improved our tracing technique to allow tracing programs with errors; when an error is reached, the error message is added to the end of the trace text and tracing is stopped. We extracted a total of 670,904 traces from the CodeNet data. For each dataset, we first fine-tune the model on these datasets, and then perform a second fine-tuning on MBPP-aug until convergence.

Results Results are shown in Table 3. As above, we report execution accuracy across tasks. We additionally report trace accuracy across tasks, to understand the extent to which the entire trace is accurately predicted. We also report the raw execution and trace accuracy across all test examples, as an additional metric to compare models.

Training on either the single-line dataset or the CodeNet dataset alone seem to both provide gains over MBPP-aug (23.4% and 25.2% tasks executed correctly, respectively). However, combining both CodeNet and the single-line dataset seems lead to the highest performance; tracing produces the correct final output for 26.6% of the tasks, and nearly a quarter of the tasks (24.6%) are traced perfectly for all three examples. These results seem promising: the neural network can often exactly trace programs. In particular, greedily decoding from the best model produces the exact correct trace for almost 42% of all traces.

6 Related Work

The tasks in this paper can be viewed as exploring one criticism of large language models, namely, to what extent do they simply rely on surface-level statistical correlations on text, without learn-

ing semantics or world knowledge (Bender & Koller, 2020)? In response, Li et al. (2021) provide evidence that pre-trained language models do indeed construct approximate representations of the semantics of the situations they describe in text. In the context of programs, Austin et al. (2021) approach this question by exploring the *learning to execute* task on MBPP, which we consider in Section 5.2. The idea behind this task was to explore whether neural models for synthesis that generate code could also execute it. While that work finds existing models perform poorly at predicting execution, we show that adding a scratchpad allows these models to perform better.

Work in *learning to execute* has considered whether off-the-shelf recurrent neural networks (Zaremba & Sutskever, 2014) or more specialized architectures (Dehghani et al., 2018; Bieber et al., 2020) have an inductive bias that is sufficiently well suited for executing arbitrary code. The related problem of *neural algorithm induction* has attracted considerable interest (Graves et al., 2014; Kurach et al., 2016; Kaiser & Sutskever, 2016; Graves et al., 2016; Reed & de Freitas, 2016). This work proposes new neural architectures, inspired by theoretical models of computation, whose inductive bias allows them to more easily learn algorithm induction tasks. Several methods for algorithm induction specifically add adaptive computation time to sequence models (Graves, 2016; Dehghani et al., 2018; Banino et al., 2021). In particular, universal transformers include adaptive computation time, and are evaluated both on algorithm induction and on learning to execute tasks (Dehghani et al., 2018). In contrast, a scratchpad is a simple way both to provide a model with adaptive computation time, and also to provide supervision about how to use that additional computation, without requiring modification to the underlying architecture.

Algorithm induction has also been connected to pre-trained models. Lu et al. (2021) show that Transformers can be used to some extent as universal computation engines, by pre-training them on natural language, and fine-tuning a small fraction of the weights on non-language tasks, including simple algorithm induction tasks. Finally, supervised approaches to semantic parsing (Zelle & Mooney, 1996; Zettlemoyer & Collins, 2005; Kwiatkowksi et al., 2010; Wong & Mooney, 2006) predict the text of a database query, which can then be executed to answer a natural language question.

7 Limitations and Future Work

Context window size In this work, we limit our experiments to problems where the scratchpad text fits within the model generation window. However, many problems require very long scratchpad generations. Therefore, fully realizing the potential of the scratchpad technique may require further improvements in transformer generation window size. This is an active area of research in NLP (Tay et al., 2020), and improvements would be beneficial for the scratchpad technique.

Learning to use the scratchpad without supervision A clear next step is to try to learn to use the scratchpad without direct supervision. A simple method would be to use reinforcement learning (RL) techniques: models would be rewarded for correctly answering questions, with reward inversely proportional to the number of scratchpad tokens used. We would hope that learning to use the scratchpad would be a transferable skill; for example, a model could potentially use the algorithm it learned to perform long addition to succeed at polynomial evaluation.

8 Conclusion

In this work we showed—through experiments on long addition, polynomial evaluation, and Python code execution—that allowing models to read from and write to a simple scratchpad can improve their performance on algorithmic tasks. Such models may be a first step toward combining the knowledge-compression capabilities of large language models with reasoning capabilities, in order to build models that *understand* code as well as write it. This could be useful for a variety of applications that require both working with natural language and reasoning about program semantics, such as program synthesis, neural-guided program analysis, and interactive programming assistants. The scratchpad technique presented here might not take us all the way toward that goal, but we hope it is an important step.

REFERENCES

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations (ICLR)*, February 2018.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Andrea Banino, Jan Balaguer, and Charles Blundell. Pondernet: Learning to ponder. In *8th ICML Workshop on Automated Machine Learning (AutoML)*, 2021.

Emily M. Bender and Alexander Koller. Climbing towards NLU: On meaning, form, and understanding in the age of data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 5185–5198, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.463. URL https://aclanthology.org/2020.acl-main.463.

David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. Learning to execute programs with instruction pointer attention graph neural networks. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 8626–8637. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/62326dc7c4f7b849d6f013ba46489d6c-Paper.pdf.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. URL https://arxiv.org/abs/2005.14165.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, Will Guss, Alex Nichol, Igor Babuschkin, Suchir Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, July 2021. URL http://arxiv.org/abs/2107.03374.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. July 2018.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. *CoRR*, abs/1703.07469, 2017. URL http://arxiv.org/abs/1703.07469.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019.

Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626): 471–476, 2016.

Lukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. In *International Conference on Learning Representations, (ICLR)*, 2016.

Tom Kwiatkowksi, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pp. 1223–1233, October 2010.

Belinda Z. Li, Maxwell Nye, and Jacob Andreas. Implicit representations of meaning in neural language models. *ArXiv*, abs/2106.00737, 2021.

Kevin Lu, Aditya Grover, Pieter Abbeel, and Igor Mordatch. Pretrained transformers as universal computation engines. March 2021.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. Project CodeNet: A Large-Scale AI for code dataset for learning a diversity of coding tasks. May 2021. URL `http://arxiv.org/abs/2105.12655`.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019. URL `http://arxiv.org/abs/1910.10683`.

Scott Reed and Nando de Freitas. Neural programmer-interpreters. In *International Conference on Learning Representations (ICLR)*, 2016. URL `http://arxiv.org/pdf/1511.06279v3`.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. *CoRR*, abs/1904.01557, 2019. URL `http://arxiv.org/abs/1904.01557`.

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers. *CoRR*, abs/2011.04006, 2020. URL `https://arxiv.org/abs/2011.04006`.

Petar Velickovic and Charles Blundell. Neural algorithmic reasoning. *CoRR*, abs/2105.02761, 2021. URL `https://arxiv.org/abs/2105.02761`.

Yuk Wah Wong and Raymond J Mooney. Learning for semantic parsing with statistical machine translation. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics* -, Morristown, NJ, USA, 2006. Association for Computational Linguistics.

Wojciech Zaremba and Ilya Sutskever. Learning to execute. *ArXiv*, abs/1410.4615, 2014.

J Zelle and R Mooney. Learning to parse database queries using inductive logic programming. In *National Conference on Artificial Intelligence (AAAI)*, 1996.

Luke S Zettlemoyer and Michael Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Uncertainty in Artificial Intelligence*, July 2005.

A EXAMPLE FEW-SHOT PROMPT FOR SYNTHETIC PYTHON EXPERIMENTS

Below is an example of a prompt for few-shot synthetic Python synthesis problems:

```
Consider the following Python function:

def f(v0):
  v0 += 0
  v4 = 2
  while v4 > 0:
    v4 -= 1
    v0 *= 2
  return v0

output = f(6)

What is the execution trace?

[BEGIN]

state: {}
line: def f(v0):
state: {"f": "<callable_object f>"}
line: output = f(6)
state: {"v0": 6}
line:   v0 += 0
state: {"v0": 6}
line:   v4 = 2
state: {"v0": 6, "v4": 2}
line:   while v4 > 0:
state: {"v0": 6, "v4": 2}
line:     v4 -= 1
state: {"v0": 6, "v4": 1}
line:     v0 *= 2
state: {"v0": 12, "v4": 1}
line:   while v4 > 0:
state: {"v0": 12, "v4": 1}
line:     v4 -= 1
state: {"v0": 12, "v4": 0}
line:     v0 *= 2
state: {"v0": 24, "v4": 0}
line:   while v4 > 0:
state: {"v0": 24, "v4": 0}
line:   return v0
state: {"f": "<callable_object f>", "output": 24}

[DONE]

Consider the following Python function:

def f(v0):
  v0 -= 0
  v0 += 2
  v0 -= 0
  return v0

output = f(4)

What is the execution trace?

[BEGIN]

state: {}
line: def f(v0):
state: {"f": "<callable_object f>"}
line: output = f(4)
state: {"v0": 4}
line:   v0 -= 0
state: {"v0": 4}
line:   v0 += 2
state: {"v0": 6}
line:   v0 -= 0
state: {"v0": 6}
line:   return v0
state: {"f": "<callable_object f>", "output": 6}

[DONE]

Consider the following Python function:
```

```
def f(v0):
  v0 -= 0
  v8 = 2
  while v8 > 0:
    v8 -= 1
    v0 *= 1
  return v0

output = f(4)

What is the execution trace?

[BEGIN]

state: {}
line: def f(v0):
state: {"f": "<callable_object f>"}
line: output = f(4)
state: {"v0": 4}
line:   v0 -= 0
state: {"v0": 4}
line:   v8 = 2
state: {"v0": 4, "v8": 2}
line:   while v8 > 0:
state: {"v0": 4, "v8": 2}
line:     v8 -= 1
state: {"v0": 4, "v8": 1}
line:     v0 *= 1
state: {"v0": 4, "v8": 1}
line:   while v8 > 0:
state: {"v0": 4, "v8": 1}
line:     v8 -= 1
state: {"v0": 4, "v8": 0}
line:     v0 *= 1
state: {"v0": 4, "v8": 0}
line:   while v8 > 0:
state: {"v0": 4, "v8": 0}
line:   return v0
state: {"f": "<callable_object f>", "output": 4}

[DONE]

Consider the following Python function:

def f(v0):
  v0 += 2
  v0 *= 2
  v0 *= 1
  return v0

output = f(8)

What is the execution trace?

[BEGIN]
```

B INCORRECT OUTPUT FOR SYNTHETIC PYTHON PROBLEMS

Below is an example of the failure of the few-shot synthetic experiments discussed in Section 5.1. Given a problem such as:

```
Consider the following Python function:

def f(v0):
  v0 += 2
  v0 *= 2
  v0 *= 1
  return v0

output = f(8)

What is the execution trace?
```

The model produces the following trace:

```
state: {"v0": 8}
line: def f(v0):
```

```
state: {"v0": 8, "f": "<function f at 0x00BFC1C0>"}
line: v0 += 2
state: {"v0": 10}
line: v0 *= 2
state: {"v0": 20}
line: v0 *= 1
state: {"v0": 20}
line: return v0
state: {"f": "<function f at 0x00BFC1C0>", "v0": 20}
```

Note that this trace is exactly correct, except for the last line: where the model predicted `"v0": 20`, the correct output is `"output": 20`. Because this type of error consistently occurs in the few-shot synthetic Python experiments, we modified the evaluation script slightly to consider this output to be correct.