# OccamLLM: Fast and Exact Language Model Arithmetic in a Single Step

**Owen Dugan**[*] [†]
Department of Physics
Massachusetts Institute of Technology
Cambridge, MA
odugan@mit.edu

**Donato M. Jiménez-Benetó**[*]
Department of Physics
Massachusetts Institute of Technology
Cambridge, MA
donatojb@mit.edu

**Charlotte Loh**
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA
cloh@mit.edu

**Zhuo Chen**
Department of Physics
Massachusetts Institute of Technology
Cambridge, MA
chenzhuo@mit.edu

**Rumen Dangovski**
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA
rumenrd@mit.edu

**Marin Soljačić**
Department of Physics
Massachusetts Institute of Technology
Cambridge, MA
soljacic@mit.edu

## Abstract

Despite significant advancements in text generation and reasoning, Large Language Models (LLMs) still face challenges in accurately performing complex arithmetic operations. Language model systems often enable LLMs to generate code for arithmetic operations to achieve accurate calculations. However, this approach compromises speed and security, and fine-tuning risks the language model losing prior capabilities. We propose a framework that enables exact arithmetic in *a single autoregressive step*, providing faster, more secure, and more interpretable LLM systems with arithmetic capabilities. We use the hidden states of a LLM to control a symbolic architecture that performs arithmetic. Our implementation using Llama 3 with OccamNet as a symbolic model (OccamLlama) achieves 100% accuracy on single arithmetic operations ($+, -, \times, \div, \sin, \cos, \log, \exp, \sqrt{}$), outperforming GPT 4o with and without a code interpreter. Furthermore, OccamLlama outperforms GPT 4o with and without a code interpreter on average across a range of mathematical problem solving benchmarks, demonstrating that OccamLLMs can excel in arithmetic tasks, even surpassing much larger models. Code is available at https://github.com/druidowm/OccamLLM.

## 1 Introduction

Since the release of GPT 3, Large Language Models (LLMs) have dramatically improved in their text generation and reasoning capabilities. This has enabled success in downstream applications including machine translation [1, 2], sentiment analysis [3, 4, 5], and interactive dialogue generation

[*]Equal contribution
[†]Corresponding author

38th Conference on Neural Information Processing Systems (NeurIPS 2024).


https://doi.org/10.52202/079017-1125

Table 1: OccamLLM is the only approach to improving the arithmetic capabilities of a pretrained LLM which 1) enables single-pass arithmetic, 2) does not risk catastrophic forgetting from finetuning, 3) does not require arbitrary code execution, and 4) provides an interpretable process.

| | Single Pass | No Catastrophic Forgetting | No Arbitrary Code Execution | Interpretable |
|---|---|---|---|---|
| Fine Tuning | ✓ | ✗ | ✓ | ✗ |
| Tool Use | ✗ | ✗ | ✗ | ✓ |
| **OccamLLM** | ✓ | ✓ | ✓ | ✓ |

[6], with language models even surpassing human experts on some academic benchmarks that require reading comprehension, reasoning and coding [7]. However even industry-leading LLMs such as GPT 4 cannot reach 100% accuracy on simple arithmetic [8], limiting their ability to perform basic mathematical tasks. This hinders potential applications of LLMs ranging from chat-bot physics tutors to LLM-powered automated research that could accelerate scientific discovery and technological innovation. The poor arithmetic performance of LLMs is particularly acute for small LLM agents, limiting their usage in smartphone or in multi-agent applications.

To enable accurate calculations, language model systems often resort to running code written by a LLM. However, this comes at the cost of speed; the model must perform multiple autoregressive steps to generate code that performs the appropriate arithmetic operations. This increased decoding time may negatively impact applications such as multi-agent workflows [9, 10] where speed is essential. At the same time, code-based LLM arithmetic mechanisms may increase system vulnerability by providing a mechanism for arbitrary LLM-generated code execution.

We propose an alternative, a framework which enables exact and interpretable LLM arithmetic in *a single autoregressive step*, providing faster and more secure arithmetic capabilities in LLM systems. Our framework uses the hidden states of a LLM to control a symbolic architecture that performs arithmetic. Although our method can in principle work with any symbolic architecture, in this paper we use an interpretable neurosymbolic architecture known as OccamNet [11, 12] because of its interpretability and scalability. Therefore, we term our method OccamLLM, or OccamLlama when using a Llama model as the LLM.

Our core contributions are as follows:

1. We develop a framework for exact and interpretable LLM arithmetic in a single autoregressive step without catastrophic forgetting [13] or vulnerability from code generation. We explore how to train OccamLlama, including data generation, decoder architecture, and loss function.
2. We benchmark OccamLlama on arithmetic tasks, demonstrating that OccamLlama achieves 100% accuracy on arbitrary single arithmetic operations ($+, -, \times, \div, \sin, \cos, \log, \exp, \sqrt{}$), more than double the accuracy of GPT 4o. OccamLlama performs slightly better than GPT 4o with Code Interpreter while answering in on average more than 50x fewer generation tokens.
3. We benchmark on mathematical problem solving tasks, showing that OccamLlama can sustain long generations. OccamLlama outperforms both GPT 4o and GPT 4o with code interpreter on average across the benchmarks we tested.

## 2 Related Work

**Arithmetic Performance in LLMs.** Prior research has trained models on synthetic data, finding that such models can achieve near-perfect accuracy on addition [14, 15], subtraction [15], multiplication [14, 15], division [15], and raising to powers [15]. These prior models have been tested only on arithmetic datasets, so their generality has not been assessed. Other work focuses on finetuning LLMs which are already trained on large amounts of general-purpose data on math datasets. Both full-parameter [16, 17] and parameter-efficient (PEFT) [18] finetuning strategies have been applied. However, finetuning on a single dataset carries the risk of catastrophic forgetting of an LLM's previously acquired linguistic skills [19]. While PEFT techniques have been shown to partially mitigate this effect, this area is still one of active research [20, 21].

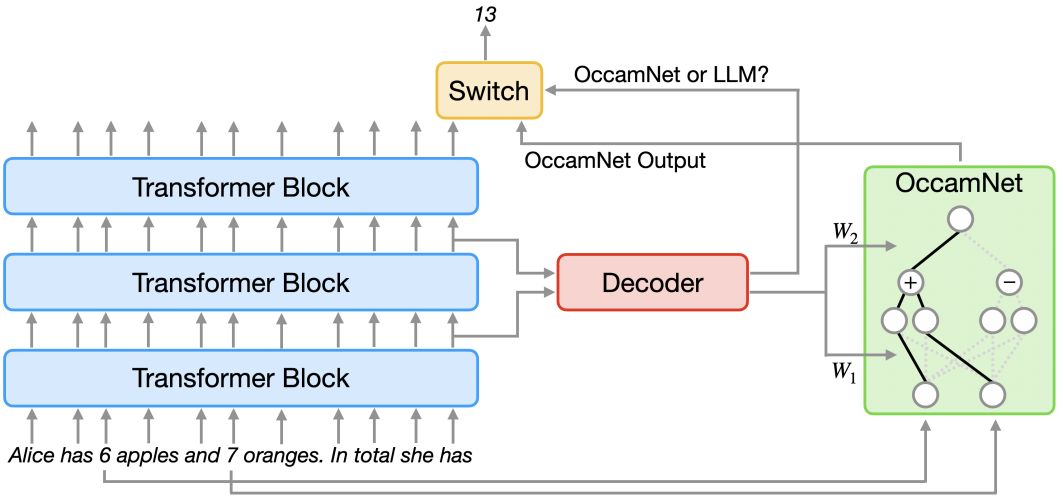


Figure 1: The OccamLLM system. For each autoregressive step, the language model hidden states for that token are fed into a decoder block which assigns weights to OccamNet. The system feeds the most recent numbers from the text into OccamNet, which then evaluates the sparse function specified by its weights. The decoder then determines whether to use the LLM output or the OccamNet output.

**LLMs with Tool Use.** Another thrust of prior research has focused on LLM tool use, which we believe is most directly related to our methods. *Calc-X* [22] introduces a technique to offload arithmetic computations to an external tool like a calculator. The authors curated a large dataset of arithmetic problems and trained a language model that learns to interact with a calculator through the use of tags to signify the calling of the external tool. Several other works [23, 24, 25] follow a similar idea, using crowd workers to annotate tool calls and using this data to train language models to interact with external tools such as a web searching tool, a calculator, or a translation system. These approaches can be prohibitively expensive in annotation costs; *Toolformer* [26] overcomes this cost by using in-context learning and a language model to generate datasets containing the necessary 'API' tool calls via a self-supervised loss. Further, the above methods all require finetuning of the LLM, placing the LLM at risk of losing generality and its original language modelling abilities through catastrophic forgetting. In contrast, our approach does not involve training the language model. Our 'external tool' is a symbolic model which can be trained to correctly use the hidden states of the language model to perform the required arithmetic computations. The language model is kept frozen throughout this process. Unlike other tool-calling approaches, where the cost of data annotation to train for tool-calling interaction can be prohibitively expensive, in our method, each task only requires manually annotating tens of prompts, a high annotation efficiency. Other prior methods leverage prompt engineering to improve arithmetic performance of LLMs; this is done either through chain-of-thought [27], or to encourage LLMs to use a code interpreter [28, 29, 30]. Contrary to these methods, our approach does not use code kernels; this provides several advantages: 1) it enables tool use without expending compute on autoregressive steps for token generation, and 2) it avoids running potentially incorrect or malicious code generated by language models.

## 3 Methods

### 3.1 OccamLLM: Combining a Language Model with a Symbolic Model

In short, the OccamLLM system combines a language model with a symbolic model, namely OccamNet, that can perform arithmetic operations like addition and subtraction. For each token, the corresponding internal hidden states of the language model are fed into a decoder module which initializes the symbolic model such that it executes the operation required by the task described in the input text. A string parser feeds the necessary numbers from the text into OccamNet, which evaluates the desired expression. Finally, a decoder determines whether to use the language model output or the OccamNet output for generating the next token.


https://doi.org/10.52202/079017-1125

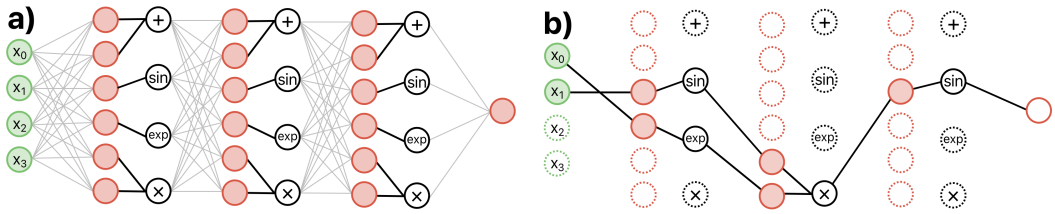


Figure 2: a) A schematic of the OccamNet architecture, with softmax layers in grey and their outputs in red. b) A Directed Acyclic Graph (DAG) (with edges not connected to the output removed for clarity) formed by sampling from OccamNet. This DAG corresponds to the function $\sin(\sin(x_1) \cdot \exp(x_0))$. Modified from [11].

In the example shown in Figure 1, a decoder determines how to initialize OccamNet from the language model hidden states, choosing to have OccamNet perform addition. The text parser then feeds the numbers 6 and 7 into OccamNet, which adds the numbers, returning 13. Finally, a decoder decides to use the OccamNet output instead of the language model output, so the system outputs 13. The new sentence, including the 13, is tokenized and fed back to the LLM to continue autoregressive generation. The language model might later generate "Since she ate two apples, she now has," at which point the switch will again trigger OccamNet, this time implementing $13 - 2$ and returning 11.

In the subsections below, we describe the OccamLLM system which from our experiments we find to be most performant, even oupterforming GPT 4o in several benchmarks. For an analysis of alternate architectures and losses, see Appendix D.

### 3.1.1 OccamNet

OccamNet is a symbolic architecture that provides an interpretable way of parametrizing probability distributions over a space of functions [11]. We leave a more thorough explanation of OccamNet to [11] and Appendix E, describing only the relevant components here.

An $l$-layer OccamNet with primitives $\mathcal{P}$ and $n$ inputs is an architecture that defines a probability distribution over the space of functions representable as compositions of the primitives in $\mathcal{P}$ up to depth $l$. For example, a two-layer OccamNet with primitives $\mathcal{P} = \{\sin, \cos\}$ and one input represents a probability distribution over the set

$$\mathcal{F} = \{x, \sin(x), \cos(x), \sin(\sin(x)), \sin(\cos(x)), \cos(\sin(x)), \sin(\sin(x))\}.$$

OccamNet has the structure of an $n$-input, $l$-internal-activation-layer multilayer perceptron with the biases removed and the activations in each layer replaced by the primitives $\mathcal{P}$, as shown in Figure 2a. Activation functions may have multiple inputs. We rename the linear layers *softmax layers*, denote the weights of the $i$th softmax layer as $\mathbf{W}^{(i)}$, and denote the combined weights of OccamNet as $\mathbf{W}$.

We define the probability distribution which OccamNet parametrizes by specifying how to sample from it. For each softmax layer output node (shown in red in Figure 2), we select a single connection to that node from a softmax layer input node by sampling from the distribution given by the softmax of the weights of the connections to the different inputs. This process produces a directed acyclic graph (DAG) defining a computational path through the OccamNet activations, such as the one shown in Figure 2b. In this way, each DAG represents a function on the inputs of OccamNet.

To ensure that OccamNet can represent all possible compositions of functions in $\mathcal{P}$ up to depth $l$, we include the following modifications to the OccamNet architecture: 1) for each softmax layer, we concatenate its inputs with the previous softmax layer's inputs to enable the representation of functions with fewer than $l$ compositions, and 2) we repeat primitives in the $i$th activation layer $A^{l-i}$ times, where $A$ is the maximum number of inputs of any of the primitives, to ensure that a sufficient number of each primitive is available at each layer. We refer to this modified architecture as *complete OccamNet* as it can represent the complete set of desired functions. The resulting architecture is shown in Figure 7 in the appendix.

In principle, OccamLLM can work with any symbolic model, i.e., any model that can parameterize a set of symbolic functions or a distribution over such functions. We choose OccamNet as opposed to, for example, a transformer [31] or recurrent neural network [32], for two reasons: 1) OccamNet

is interpretable, which we hypothesize makes controlling OccamNet an easier task for a decoder to learn, and 2) OccamNet is parallelizable over multiple samples, allowing for scalable training.

#### 3.1.2 OccamLLM Decoder

The OccamLLM decoder takes the hidden states of a language model and outputs an initialization for OccamNet. This gives the LLM control over which function to apply on the inputs. The decoder acts on each input token separately, producing a different OccamNet initialization for each. Therefore, the arithmetic operations predicted may change along an input sequence, allowing OccamNet's use for different computations in a single multi-token generation. This is crucial in multi-step reasoning scenarios where OccamNet is employed several times for different purposes.

Many decoder architectures are possible. We choose to parameterize the weights of each softmax layer of OccamNet independently, as $(\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(l)}) = (\text{Decoder}_1(\mathbf{h}), \ldots, \text{Decoder}_l(\mathbf{h}))$, where $\mathbf{h}$ are the hidden states of the language model. We choose

$$\text{Decoder}_i(\mathbf{h}) = \text{MLP}_i\left(\sum_{j=1}^{L} w_{i,j}\mathbf{h}_j\right) + \mathbf{W}^{*(i)} \tag{1}$$

where $\mathbf{h}_j$ are the hidden states of the $j$th layer of the language model, $w_{i,j}$ are trainable weights, $\text{MLP}_i$ are two-layer multilayer perceptrons (MLPs), and $\mathbf{W}^{*(i)}$ are untrained weights which initialize all functions to have approximately equal probabilities according to the initialization scheme described in [11] and explained in Appendix E.4.

#### 3.1.3 OccamLLM Switch

We similarly train a decoder for a switch that, for each input token, is fed the hidden states of the language model and selects whether to use the output of OccamNet or the output of the language model. The decoder outputs a single number from 0 to 1, where all numbers less than or equal to 0.5 correspond to using the output of the language model and all numbers greater than 0.5 correspond to using the output of OccamNet. We choose the following architecture for the switch decoder:

$$\text{Decoder}_{\text{switch}}(\mathbf{h}) = \text{sigmoid}\left(\text{MLP}_{\text{switch}}\left(\sum_{j=1}^{L} w_{\text{switch},j}\mathbf{h}_j\right)\right). \tag{2}$$

### 3.2 Data Generation

We create synthetic datasets to train the OccamLLM decoders, which contain instruction prompts for diverse arithmetic tasks. To generate datasets of arbitrary size, we create prompts with placeholders for numbers. Each prompt includes a question with number placeholders, the sampling value range for each number, and a function that computes the answer to the query given the sampled input numbers. The prompts fall into two main categories: purely arithmetic tasks and reasoning problems.

Purely arithmetic prompts are formed by expressions including only symbols, without any natural language added, such as "$3 + 85 =$." We create prompts using the following operations: $+(\cdot,\cdot)$, $-(\cdot,\cdot)$, $\times(\cdot,\cdot)$, $\div(\cdot,\cdot)$, sqrt$(\cdot)$, power$(\cdot,\cdot)$, $\log_e(\cdot)$, $\exp(\cdot)$, $\sin(\cdot)$, and $\cos(\cdot)$.

We also include word problems that require one or two reasoning steps. We generated 150 single step word problems and 40 multi-step reasoning problems which we modified from examples in the MultiArith training dataset [33].

#### 3.2.1 OccamNet Decoder Training Data

For training the decoder that controls the weights of OccamNet, we created two types of examples, single queries and concatenated queries. For single queries, we select a single prompt from the problems generated as discussed in Section 3.2. We use the Llama 3 Instruct chat template and fill in the query as the user input and the result as the assistant response, prepending "Answer = " to the later in randomly selected samples (see Appendix A.1.1 for further details). For the concatenated queries of examples, we select a random number of prompts and concatenate the query-response pairs without using the Llama 3 Instruct chat template. The OccamNet decoder is trained to predict only

 https://doi.org/10.52202/079017-1125

the results of the last query in the sequence. This strategy helps OccamLLM to learn which operation to perform without becoming confused by earlier text, which is useful for continuous generation. To create the training dataset, each example is sampled by first randomly selecting whether to create a single or concatenated query, then randomly selecting the type(s) of prompt(s) used, and finally randomly sampling the input values from the range corresponding to each selected prompt.

#### 3.2.2 OccamLLM Switch Training Data

To train the switch, we generate examples of possible LLM outputs for given input expressions and label the outputs with sequences of 0s or 1s corresponding to whether the language model or the OccamNet output should be used for the next token. Some examples correspond to the prompts described in Section 3.2. For such examples, the LLM output is set to "The answer is" or "Answer = " and the label sequence is all 0s with a 1 at the last token to indicate the system should use OccamNet only to compute the answer. We also manually created and labeled several other examples for diverse scenarios to explicitly teach the system in which cases it should or should not use OccamNet (see Appendix A.1.2 for further details).

To create the training dataset, we concatenate a random number of the above user input - assistant output pairs in a conversational fashion, using the Llama 3 Instruct chat template.

### 3.3 OccamLLM Training

We train the OccamLLM decoder and the switch separately, as they do not share weights. In all cases, the weights of the LLM are kept frozen. In the first step, we train the system to predict the answer to examples generated by the method explained in Section 3.2.1. The OccamNet decoder processes the hidden states corresponding to the last token of the response and sets the weights of OccamNet such that the correct arithmetic expression is sampled. In this step, we use a rescaled REINFORCE [34] loss, which can also be interpreted as a Monte-Carlo estimate of the cross-entropy loss (see Appendix D.2):

$$\mathcal{L}(x, y; W) = -\frac{\sum_{f\sim p_W} R(f(x), y) \log p_W[f]}{\sum_{f\sim p_W} R(f(x), y)}, \tag{3}$$

where $p_W[f] \equiv \text{ON}(f; \text{Decoder}_W(\mathbf{h}(x)))$ is the probability distribution represented by the decoder-initialized OccamNet.

Minimizing this loss steers the decoder towards assigning higher probabilities to the functions that maximize the reward $R(f(x), y)$, which measures the similarity between the correct answer $y$ and the prediction of OccamNet $f(x)$. We find setting $R(f(x), y) = 1$ if $f(x) = y$, and 0 otherwise, most effective. We discuss the OccamNet loss in more detail in Appendix D.

The second step involves training the decoder to route the outputs to OccamNet when needed. We train the switch decoder alone, freezing the weights of the OccamNet decoder of the previous step and minimizing the binary cross-entropy loss between the switch output and the desired output for each token. The OccamLLM switch decoder learns when to route the output to OccamNet in diverse contexts.

## 4 Experiments

For all OccamLLM results, we use Llama 3 8B Instruct and Llama 3 70B Instruct [35] as the underlying language models. As such, we call our models OccamLlama 8B and OccamLlama 70B, respectively. We use a 1 layer Complete OccamNet with primitives

$$\mathcal{P} = \{+(\cdot,\cdot), -(\cdot,\cdot), \times(\cdot,\cdot), \div(\cdot,\cdot), \text{sqrt}(\cdot), \text{power}(\cdot,\cdot), \log_e(\cdot), \exp(\cdot), \sin(\cdot), \cos(\cdot)\}.$$

This single layer OccamNet can be invoked by the LLM several times during generation to perform complex arithmetic operations accurately. To use the trained OccamLlama for inference, we sample the highest probability function from OccamNet as described in Appendix E.3.

We benchmark our methods against unmodified Llama 2 7B Chat (Llama 2 7B) [36], unmodified Llama 3 8B Instruct (Llama 3 8B) [35], unmodified Llama 3 70B Instruct (Llama 3708B) [35], gpt-3.5-turbo-0125 (GPT 3.5 Turbo) [37], gpt-4o-2024-05-13 (GPT 4o) [38], and gpt-4o-2024-05-13 with Code Interpreter (GPT 4o + Code) [39]. To reduce costs, for GPT 4o with Code Interpreter, we test a random subset of 200 datapoints for each dataset.

Table 2: Accuracy on arithmetic tasks, in percentages. The OccamLlama column corresponds to the results of both OccamLlama 8B and OccamLlama 70B. Higher is better. Bold indicates best performance for each row.

| | OccamLlama 8B / 70B | Llama 2 7B Chat | Llama 3 8b Instruct | GPT 3.5 Turbo | GPT 4o | GPT 4o Code |
|---|---|---|---|---|---|---|
| Addition | **100.0**$\pm$**0.0** | 19.2$\pm$1.2 | 44.9$\pm$1.6 | 65.2$\pm$1.5 | 95.7$\pm$0.6 | **100.0**$\pm$**0.0** |
| Subtraction | **100.0**$\pm$**0.0** | 8.7$\pm$0.9 | 34.4$\pm$1.5 | 59.8$\pm$1.6 | 85.6$\pm$1.1 | 99.5$\pm$0.5 |
| Multiplication | **100.0**$\pm$**0.0** | 0.0$\pm$0.0 | 0.0$\pm$0.0 | 0.0$\pm$0.0 | 0.0$\pm$0.0 | 99.0$\pm$0.7 |
| Division | **100.0**$\pm$**0.0** | 2.8$\pm$0.5 | 35.3$\pm$1.5 | 10.7$\pm$1.0 | 38.6$\pm$1.5 | **100.0**$\pm$**0.0** |
| Square Root | **100.0**$\pm$**0.0** | 0.0$\pm$0.0 | 0.0$\pm$0.0 | 0.9$\pm$0.3 | 18.6$\pm$1.2 | **100.0**$\pm$**0.0** |
| Exponential | **100.0**$\pm$**0.0** | 0.3$\pm$0.2 | 3.1$\pm$0.5 | 12.5$\pm$1.0 | 23.2$\pm$1.3 | **100.0**$\pm$**0.0** |
| Logarithm | **100.0**$\pm$**0.0** | 0.1$\pm$0.1 | 0.0$\pm$0.0 | 17.1$\pm$1.2 | 21.3$\pm$1.3 | **100.0**$\pm$**0.0** |
| Sine | **100.0**$\pm$**0.0** | 7.6$\pm$0.8 | 7.0$\pm$0.8 | 13.4$\pm$1.1 | 39.3$\pm$1.5 | **100.0**$\pm$**0.0** |
| Cosine | **100.0**$\pm$**0.0** | 0.8$\pm$0.3 | 1.5$\pm$0.4 | 6.7$\pm$0.8 | 32.8$\pm$1.5 | **100.0**$\pm$**0.0** |
| AVERAGE | **100.0**$\pm$**0.0** | 4.4$\pm$0.2 | 14.0$\pm$0.4 | 20.7$\pm$0.4 | 39.5$\pm$0.5 | 99.8$\pm$0.1 |

To determine if a model output is correct, we parse all numbers in the model output and if one of them "matches" the correct answer, we determine that the result is correct. We mark each correct result as 100% accuracy and each incorrect result as 0% accuracy. For each model on each dataset, we report the mean accuracy and the standard error of the mean. To determine if a number matches the result, we first determine how many places after the decimal $d$ the number should be accurate to. If the number is an integer, we set $d$ to 2. Otherwise, we set $d$ to the number of places after the decimal in the model output, clipped between 2 and 5. Finally we state that a number "matches" the result if the number and the result differ by less than $10^{-d}$. We present further experiment details, including additional experiments, hyperparameters, and prompts in Appendix A.

### 4.1 Simple Arithmetic Problems

To evaluate OccamLlama and the baselines on purely arithmetic expressions, we create several synthetic datasets. For each of the operations in $\{+, -, \times, \div\}$, the inputs are random 7-digit positive or negative integers. For $\sqrt{}$, the inputs are random 7-digit positive integers. For the logarithms, the examples are log-uniformly sampled in the interval $(10^{-10}, 10^{10})$; for the exponentials, they are uniformly sampled in the interval $(-10, 10)$, and for sines and cosines they are uniformly sampled in the interval $(-2\pi, 2\pi)$.

The results of these evaluations are shown in Table 2. More detailed results, including relative error and results for 3- and 5-digit arithmetic, are shown in Appendix A.5.

Both OccamLlama 8B and 70B have $100.0 \pm 0.0\%$ accuracy on all tasks, missing 0 out of 9000 problems. On the other hand, we tested GPT 4o with Code Interpreter on fewer problems to save cost, and it missed 3 out of the 1800 problems it faced, achieving an accuracy of $99.8 \pm 0.1\%$.

Furthermore, GPT 4o with Code Interpreter generates on average more than 54 tokens to answer these problems, whereas our model uses OccamNet on the first forward pass. This means that, barring advanced decoding techniques such as speculative decoding [40], GPT 4o would need to be more than 50x faster than OccamLlama per forward pass to be comparable in answer generation speed on these tasks.

Table 2 demonstrates that arithmetic with LLMs is still challenging; state-of-the-art proprietary language models like GPT 4o achieve less than 40% accuracy on 7-digit division and fail to perform any 7-digit multiplications correctly. Open source LLMs fall farther behind, with Llama 3 8B achieving below 50% on relatively simple tasks such as 7-digit addition.

### 4.2 Mathematical Problem Solving

To test the performance of OccamLlama on more general mathematical problem solving tasks, we evaluate our method and baselines on the following six benchmarks: AddSub [41], GSM8K [42], MultiArith [33], MATH401 [8], Single Eq [43], and SVAMP [44]. All but MATH401 are word

 https://doi.org/10.52202/079017-1125

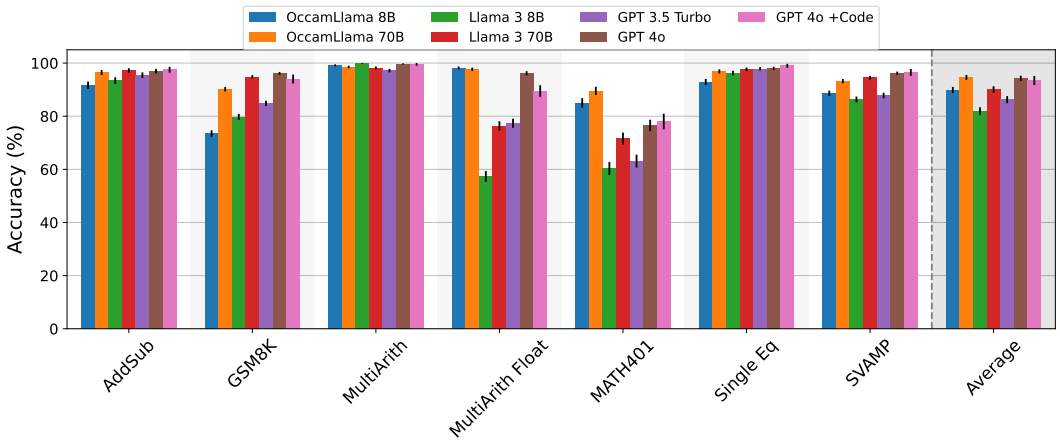


Figure 3: Accuracy of OccamLlama and baselines on mathematical problem solving tasks. Higher is better. OccamLlama 8B achieves accuracy comparable to Llama 3 8B on benchmarks with simple arithmetic, higher accuracy than GPT 4o and GPT 4o + Code on on tasks with challenging arithmetic, and accuracy above Llama 3 8B and similar to GPT 3.5 Turbo on average. OccamLlama 70B outperforms GPT 4o and GPT 4o + Code on average.

problems requiring longer generation and a mix of reasoning and arithmetic capabilities. MATH401 also includes multistep arithmetic problems which require more than one call to OccamLlama. We selected these datasets (including the MultiArith Float dataset described below) before testing any methods on them to ensure unbiased selection of benchmarks.

Because many of the arithmetic operations required in these datasets are relatively simple, we also create MultiArith Float, a modification of MultiArith in which we select problems which are arithmetically more challenging, while requiring similar levels of reasoning. To this end, we select prompts having input numbers that can be replaced with floats. For instance, $3.5$ feet or $\$39.95$ are reasonable but $3.5$ people is not. Furthermore, we sample input values from ranges larger than those appearing in the MultiArith dataset, in cases where it is reasonable. Float operations and larger additions and multiplications are more difficult for the baseline LLMs but do not make a difference for OccamLLM, so this dataset is particularly useful to show the advantages of the system we propose. Figure 3 shows the results of these evaluations. More detailed results are shown in Appendix A.5.

OccamLlama 70B outperforms both GPT 4o and GPT 4o + Code on average across the benchmarks, demonstrating OccamLlama's strong mathematical problem solving capability. We also note that GPT 4o + Code does not outperform GPT 4o on average, suggesting that existing implementations of LLMs with code generation may not help with mathematical problem solving.

We now consider the performance of OccamLlama 8B, the smaller OccamLlama model. On MultiArith Float and MATH401, two datasets requiring challenging arithmetic, OccamLlama 8B outperforms not only Llama 3 8B but also GPT 4o and GPT 4o + Code. At the same time, most other datasets in this benchmark do not involve challenging arithmetic, meaning that Llama 3 8B is well suited to solve these tasks without assistance; most of the difficulty of these tasks lies in the reasoning rather than in the arithmetic computations. This is further supported by the fact that GPT 4o with Code Interpreter never substantially outperforms and sometimes underperforms GPT 4o on these tasks. As such, it is remarkable that OccamLlama 8B can achieve comparable accuracy to Llama 3 8B even when it is trained on very different data and evaluated on tasks without challenging arithmetic.

The only datasets for which OccamLlama 8B performs noticeably worse than Llama 3 8B are GSM8K and Single Eq, but we believe this results from an imperfect OccamLlama switch, likely stemming from text which is outside of the switch training distribution (see Section 4.3). Fortunately, in Appendix C, we find that the OccamNet decoder is quite robust to out of distribution data and that both the OccamNet and switch decoders generalize well to unseen languages. This suggests that, with relatively little data, it should be possible to teach the switch to handle these unseen cases, something we leave for future work.

In Figure 4, we show example generations from OccamLlama 8B for both arithmetic and reasoning tasks. These generations demonstrate how the OccamLlama switch learns to balance OccamNet

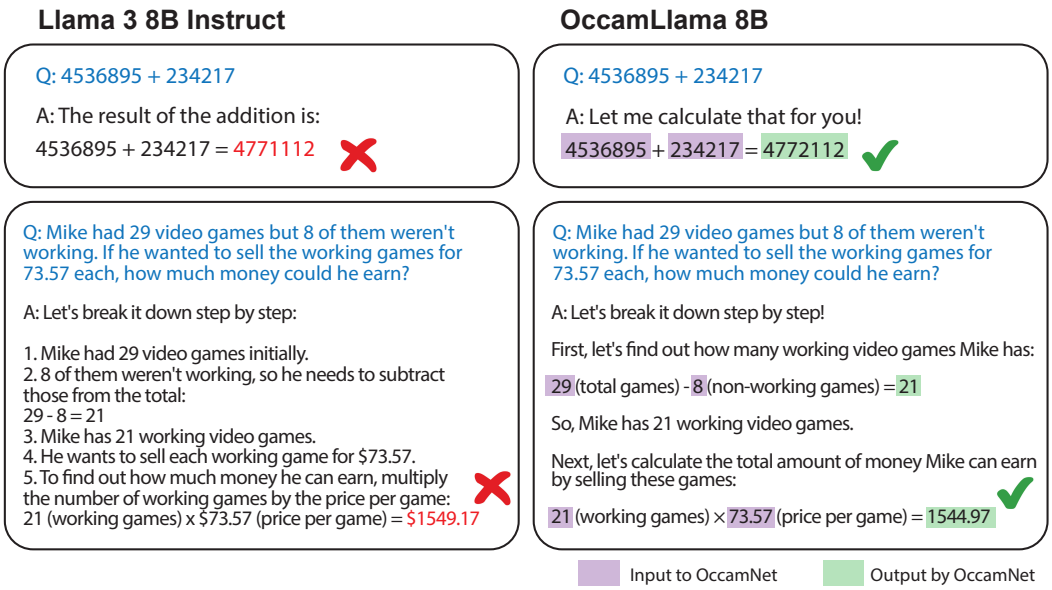


Figure 4: Examples from Llama 3 8B Instruct and OccamLlama 8B on (top) an arithmetic problem and (bottom) a mathematical reasoning problem from the MultiArith Float dataset. In OccamLlama, the LLM performs reasoning, the switch predicts when to use OccamNet, and OccamNet performs arithmetic operations. OccamNet's inputs and outputs are highlighted in purple and green, respectively.

outputs with LLM outputs, effectively distributing the work between a reasoner (Llama) and a calculator (OccamNet). Because the language model is unaware of the OccamLlama system, its generations behave as if it possesses an interior calculator even though it is actually using a tool. In this way, we combine the benefits of a language model finetuned on arithmetic with the benefits of a language model finetuned to use code for arithmetic, all *without any finetuning*.

### 4.3 Limitations

In our experiments, we use a single-layer OccamNet as the symbolic network, enabling evaluation of *single-operation* arithmetic problems. This sometimes poses a challenge on reasoning problems when the base language model generates compound expressions requiring more than one operation to evaluate, such as $3 + 5 + 7 =$. A single-layer OccamNet cannot evaluate these expressions. We attempted to overcome this by prompting Llama to break down compound expressions into multiple steps, but we find it difficult to coerce Llama to follow these instructions. Another challenge is that Llama often generates expressions in fractions or percentages, which also constitute compound expressions that are not properly handled by the OccamLLM system. Fortunately, we observed that these compound expressions were typically simple enough for the LLM to evaluate without OccamNet. Therefore, in our experiments, we trained the OccamLLM switch to avoid using OccamNet for compound operations, largely mitigating this issue. Future work could explore other solutions such as integrating a two-layer OccamNet as the symbolic network. We found that these issues are particularly acute in the GSM8K and Single Eq datasets, where the expressions generated by Llama are not prevalent in the switch training data, causing it to sometimes incorrectly trigger OccamNet and degrade performance, as discussed more in Appendix A.5.

Furthermore, we found that the language model sometimes appends further digits to OccamLlama outputs, defeating the purpose of OccamLlama generations. To address this issue, we append "\n\n." to every number computed with OccamNet, emulating the usual behavior of Llama.

These techniques demonstrate a design paradigm of OccamLlama: by tuning the behaviors of OccamNet and the switch, we can often avoid finetuning the LLM.


https://doi.org/10.52202/079017-1125

## 5 Discussion

We presented OccamLLM, a system enabling exact and interpretable language model arithmetic in a single autoregressive step. Our method does not require modifying the weights of the underlying language model, thereby avoiding risks of catastrophic forgetting. Furthermore, our method avoids security risks arising from running code generated by a language model while outperforming top LLM code generation methods (GPT 4o + Code) on average across our benchmarks.

We benchmarked our method on challenging arithmetic tasks, achieving 100% accuracy where GPT 4o achieves only 40% performance on average. We also benchmarked our method on mathematical problem solving tasks, demonstrating that the OccamLlama switch can accurately balance the LLM for reasoning and OccamNet for arithmetic, outperforming even GPT 4o and GPT 4o with Code Interpreter on average.

Our work could enable smaller LLMs to be as performant as much larger LLMs in arithmetic. Moreover, integrating OccamLLM with larger LLMs like GPT 4o could further improve their arithmetic abilities without requiring a code interpreter. Furthermore, at present, OccamLLM may not integrate with more advanced decoding techniques such as speculative decoding [40, 45]. We hope to explore these avenues in future work.

## 6 Broader Impact

We believe that, in addition to enabling fast, safe, and interpretable arithmetic, OccamLLM demonstrates a new paradigm for tool use. As a proof of concept for more complex tool use, we further train OccamLlama 8B with a two layer Complete OccamNet with the primitives

$$\mathcal{P} = \{\mathsf{Addition}(\cdot,\cdot), \mathsf{Subtraction}(\cdot,\cdot), \mathsf{Multiplication}(\cdot,\cdot), \mathsf{Division}(\cdot,\cdot)\},$$

which enables OccamLlama to perform up to three arithmetic operations (e.g., $2 \cdot 7 + 3/2$) in a single autoregressive step. We find that this two-layer OccamLlama can reach near 100% accuracy, even when performing three arithmetic operations in a single autoregressive step, as shown in Table 3. This demonstrates that OccamLLM can be used to perform more complex operations, including composing multiple different tools.

For future work, we plan to explore integrating other tools beyond calculators through a similar technique. This is facilitated by the fact that there are no restrictions on OccamNet's activations; in principle, tools could be placed inside activations of OccamNet, enabling OccamNet to serve as a sort of a mixture of experts for tools. While some tools, like querying a search engine, may still be most effective when integrated into language model systems through language, we believe this work demonstrates that some tools are more effective when they can be more tightly integrated into the language model.

Table 3: Accuracy on multistep arithmetic.

| | OccamLlama | Llama 3 8b Instruct |
|---|---|---|
| One-Step | **99.9**$\pm$**0.1** | 78.1$\pm$1.3 |
| Two-Step | **98.2**$\pm$**0.4** | 57.8$\pm$1.6 |
| Three-Step | **96.1**$\pm$**0.6** | 40.2$\pm$1.6 |
| AVERAGE | **98.1**$\pm$**0.3** | 58.7$\pm$0.9 |

## Acknowledgements

We would like to thank Andrew Ma and Di Luo for their thoughtful discussions.

The authors acknowledge the MIT SuperCloud and Lincoln Laboratory Supercomputing Center for providing HPC resources that have contributed to the research results reported within this paper.

Research was sponsored by the Department of the Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of the Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

This work is also supported in part by the National Science Foundation under Cooperative Agreement PHY-2019786 (The NSF AI Institute for Artificial Intelligence and Fundamental Interactions, http://iaifi.org/).

## References

[1] Biao Zhang, Barry Haddow, and Alexandra Birch. Prompting large language model for machine translation: A case study, 2023.

[2] Wenhao Zhu, Hongyi Liu, Qingxiu Dong, Jingjing Xu, Shujian Huang, Lingpeng Kong, Jiajun Chen, and Lei Li. Multilingual machine translation with large language models: Empirical results and analysis, 2023.

[3] Xiang Deng, Vasilisa Bashlovkina, Feng Han, Simon Baumgartner, and Michael Bendersky. Llms to the moon? reddit market sentiment analysis with large language models. In *Companion Proceedings of the ACM Web Conference 2023*, WWW '23 Companion, page 1014–1019, New York, NY, USA, 2023. Association for Computing Machinery.

[4] Zengzhi Wang, Qiming Xie, Yi Feng, Zixiang Ding, Zinong Yang, and Rui Xia. Is ChatGPT a Good Sentiment Analyzer? A Preliminary Study. *arXiv e-prints*, page arXiv:2304.04339, April 2023.

[5] Wenxuan Zhang, Yue Deng, Bing Liu, Sinno Jialin Pan, and Lidong Bing. Sentiment analysis in the era of large language models: A reality check, 2023.

[6] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, ..., and Barret Zoph. Gpt-4 technical report, 2024.

[7] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy Lillicrap, Angeliki Lazaridou, Orhan Firat, ..., and Oriol Vinyals. Gemini: A family of highly capable multimodal models, 2024.

[8] Zheng Yuan, Hongyi Yuan, Chuanqi Tan, Wei Wang, and Songfang Huang. How well do large language models perform in arithmetic tasks?, 2023.

[9] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023.

[10] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges, 2024.

[11] Owen Dugan, Rumen Dangovski, Allan Costa, Samuel Kim, Pawan Goyal, Joseph Jacobson, and Marin Soljačić. OccamNet: A Fast Neural Model for Symbolic Regression at Scale. *arXiv e-prints*, page arXiv:2007.10784, July 2020.

[12] Julia Balla, Sihao Huang, Owen Dugan, Rumen Dangovski, and Marin Soljacic. AI-Assisted Discovery of Quantitative and Formal Models in Social Science. *arXiv e-prints*, page arXiv:2210.00563, October 2022.

[13] Michael McCloskey and Neal J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. volume 24 of *Psychology of Learning and Motivation*, pages 109–165. Academic Press, 1989.

[14] Davide Maltoni and Matteo Ferrara. Arithmetic with Language Models: from Memorization to Computation. *arXiv e-prints*, page arXiv:2308.01154, August 2023.

[15] Zhen Yang, Ming Ding, Qingsong Lv, Zhihuan Jiang, Zehai He, Yuyi Guo, Jinfeng Bai, and Jie Tang. GPT Can Solve Mathematical Problems Without a Calculator. *arXiv e-prints*, page arXiv:2309.03241, September 2023.

[16] Yixin Liu, Avi Singh, C. Daniel Freeman, John D. Co-Reyes, and Peter J. Liu. Improving large language model fine-tuning for solving math problems, 2023.

 https://doi.org/10.52202/079017-1125

[17] Ke Wang, Houxing Ren, Aojun Zhou, Zimu Lu, Sichun Luo, Weikang Shi, Renrui Zhang, Linqi Song, Mingjie Zhan, and Hongsheng Li. Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning, 2023.

[18] Zhiqiang Hu, Lei Wang, Yihuai Lan, Wanyu Xu, Ee-Peng Lim, Lidong Bing, Xing Xu, Soujanya Poria, and Roy Ka-Wei Lee. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models, 2023.

[19] Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

[20] Haolin Chen and Philip N. Garner. Bayesian parameter-efficient fine-tuning for overcoming catastrophic forgetting, 2024.

[21] Shuo Liu, Jacky Keung, Zhen Yang, Fang Liu, Qilin Zhou, and Yihan Liao. Delving into parameter-efficient fine-tuning in code change learning: An empirical study, 2024.

[22] Marek Kadlčík, Michal Štefánik, Ondřej Sotolář, and Vlastimil Martinek. Calc-x and calcformers: Empowering arithmetical chain-of-thought through interaction with symbolic systems, 2023.

[23] Mojtaba Komeili, Kurt Shuster, and Jason Weston. Internet-augmented dialogue generation. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8460–8478, Dublin, Ireland, May 2022. Association for Computational Linguistics.

[24] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback, 2022.

[25] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Vincent Zhao, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Pranesh Srinivasan, Laichee Man, Kathleen Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed Chi, and Quoc Le. Lamda: Language models for dialog applications, 2022.

[26] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023.

[27] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[28] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models, 2023.

[29] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, 2023.

[30] Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, and Hongsheng Li. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification, 2023.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *arXiv e-prints*, page arXiv:1706.03762, June 2017.

[32] Robin M. Schmidt. Recurrent Neural Networks (RNNs): A gentle Introduction and Overview. *arXiv e-prints*, page arXiv:1912.05911, November 2019.

[33] Subhro Roy and Dan Roth. Solving general arithmetic word problems, 2016.

[34] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, May 1992.

[35] AI@Meta. Llama 3 model card. 2024.

[36] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv e-prints*, page arXiv:2307.09288, July 2023.

[37] OpenAI. Gpt-3.5 turbo, 2024. OpenAI API Documentation.

[38] OpenAI. Hello gpt-4o. 2024.

[39] OpenAI. Code interpreter, 2024. OpenAI Assistants API Documentation.

[40] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast Inference from Transformers via Speculative Decoding. *arXiv e-prints*, page arXiv:2211.17192, November 2022.

[41] Mohammad Javad Hosseini, Hannaneh Hajishirzi, Oren Etzioni, and Nate Kushman. Learning to solve arithmetic word problems with verb categorization. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 523–533, Doha, Qatar, October 2014. Association for Computational Linguistics.

[42] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *ArXiv*, abs/2110.14168, 2021.

[43] Rik Koncel-Kedziorski, Hannaneh Hajishirzi, Ashish Sabharwal, Oren Etzioni, and Siena Dumas Ang. Parsing algebraic word problems into equations. *Transactions of the Association for Computational Linguistics*, 3:585–597, 2015.

[44] Arkil Patel, Satwik Bhattamishra, and Navin Goyal. Are NLP models really able to solve simple math word problems? In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2080–2094, Online, June 2021. Association for Computational Linguistics.

[45] Benjamin Spector and Chris Re. Accelerating LLM Inference with Staged Speculative Decoding. *arXiv e-prints*, page arXiv:2308.04623, August 2023.

[46] Freda Shi, Mirac Suzgun, Markus Freitag, Xuezhi Wang, Suraj Srivats, Soroush Vosoughi, Hyung Won Chung, Yi Tay, Sebastian Ruder, Denny Zhou, Dipanjan Das, and Jason Wei. Language models are multilingual chain-of-thought reasoners, 2022.

[47] Georg Martius and Christoph H. Lampert. Extrapolation and learning equations. *arXiv e-prints*, page arXiv:1610.02995, October 2016.

[48] Subham Sahoo, Christoph Lampert, and Georg Martius. Learning equations for extrapolation and control. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4442–4450, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[49] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv e-prints*, page arXiv:1707.06347, July 2017.

 https://doi.org/10.52202/079017-1125

# Appendix

## A Further Experiment Details and Results

For training and evaluation OccamLlama 8B, we used a single 32 GB NVIDIA Tesla V100 GPU. For OccamLlama 70B, we used two 80 GB NVIDIA A100 GPU. Each training run takes less than 48 hours.

In the experiments presented in Section 4, for each of the weight decoders and the switch, we used two-layer MLPs of input size 4096/8192 (Llama 3 8B/70B Instruct hidden size), intermediate size 64 and final size equal to the number of weights in the corresponding OccamNet layer or switch.

In the two-layer experiments presented in Section 6, for each of the weight decoders, we used two-layer MLPs of input size 4096 (Llama 3 8B Instruct hidden size), intermediate size 512, and final size equal to the number of weights in the corresponding OccamNet layer. We did not train a switch for this experiment as we did not test long-form generations.

### A.1 Training Dataset

#### A.1.1 OccamNet Decoder

To train the OccamNet decoder, we created a training dataset consisting of a 80,000 examples split in 40,000 single queries and 40,000 sequences of concatenated queries. In the first case, we sampled a single prompt of those described in 3.2 and formatted it using the Llama 3 Instruct chat template. In the second case, we concatenated multiple prompts described in 3.2 without the chat template.

40% of the sampled prompts correspond to simple arithmetic, concretely $+, -, \times$, and $\div$. We sampled from various input value ranges, chosen at random: integers in $[-10, 10]$, integers in $[-100, 100]$, integers in $[-1000, 1000]$, integers in $[-20000, 20000]$, floating numbers in $[-1, 1]$, and floating point numbers in $[-1000, 1000]$.

Another 40% corresponds to complex arithmetic involving square roots, logarithms, exponentials, trigonometric functions and computing one number to the power of another. For the square root and the logarithm, we sampled integers uniformly in either $[1, 100]$ or $[1, 20000]$ and floats uniformly in either $[0.01, 100]$ or $[0.01, 20000]$. For the exponential, we sampled integers and floats in $[-10, 10]$. For the powers, we sampled the base as either an integer in $[1, 25]$ or a float in $[0.1, 25]$ and the exponent as an integer in $[-6, 6]$.

The remaining 20% corresponds to single or multi step problems reasoning prompts. The inputs were sampled with various ranges, sometimes as floats and sometimes as integers, depending on the context of the problem. Because a single-OccamNet-layer OccamLlama cannot solve a multi-step reasoning problem in a single step, we never end the multiple-query examples with a multi-step reasoning problem.

We first iterated the 80,000 examples, prepending "Answer = " to the assistant response, thus training OccamNet to predict the result after the "=". Next, we validated the model on out-of-distribution examples where "Answer = " was not appended. We noticed that the accuracy on this task was improving during training, but after the full dataset was iterated it still didn't perform as well as when evaluated in-distribution. Therefore, we continued to train the model using examples of the same dataset but with no "Answer = " at the beginning of the assistant response. The model rapidly learned the new task. We stopped at 28,000 iterations of this second stage.

For the two-layer OccamNet run, we generated a large set of programmatically generated prompts of the form $3 + 97 \cdot -4 =$, with the Llama 3 Instruct chat template applied.

#### A.1.2 Switch Decoder

To train the switch decoder, we created a dataset of 50,000 examples (80,000 for OccamLlama 80B). For each example, the tokens previous to the numbers that should be computed using OccamNet, which are the ones that the switch should not route to the LLM, are labeled with a 1, and all the rest are labeled with a 0.

Half of the examples consist of a single prompt corresponding to a simple arithmetic expression as the ones described in Section 3.2. The token immediately at the beginning of the assistant response is labeled with a 1. Therefore, the trained system will answer directly to simple arithmetic queries that OccamNet can compute.

The remaining 25,000 examples consist each of a series of prompts which are formatted in the Llama 3 Instruct chat template in a conversational style. The input-output pairs used to create each sequence of prompts are distributed in the following way:

- 25% of these pairs are created by taking one of the simple arithmetic expressions as input. The output is selected randomly between answering directly at the beginning of the assistant response, adding "Answer = " before the answer, or repeating the input expression before the answer. These examples train the switch to trigger OccamNet in different scenarios where the LLM needs to compute an answer.
- 70% of the pairs come from a collection of 43 manually created and labeled examples, which illustrate in which cases the switch should route to OccamNet and, importantly, in which cases it shouldn't. This collection was designed to cover a wide variety of situations where the LLM might need to use OccamNet for computations. Furthermore, it includes cases where the LLM should avoid calling OccamNet because doing so would produce a bad prediction. This is the case, for example, of instances where the LLM attempts to add three numbers simultaneously. If it were to use the 1-layer OccamNet, which can take 2 inputs at most, the result would be incorrect.
- The remaining 5% of the prompts come from multi-step reasoning problems. We set the output for these not to a full response, but only "The answer is ". In such cases, a single-layer OccamNet cannot compute the answer, so the output tokens are labeled with a 0. This trains the system to avoid routing to OccamNet when the later cannot compute the answer.

Below, we present some manually created examples and explain what case they cover. The tokens of the numbers in **bold** were marked with a 1 to signal that the switch should route to OccamNet, and the rest of the numbers and all of the text tokens were marked with a 0 to signal that OccamNet should not be called in these cases.

The cases we cover with these prompts are:

1. Single-layer OccamNet should be called to perform two-input arithmetic operations.
2. OccamNet should not be called to write numbers that already appeared and do not require further arithmetic.
3. OccamNet should not be called to round numbers.
4. OccamNet should not be called to perform inline operations nested within a larger expression because it might contradict the assumption that the last two numbers are involved in the operation.
5. OccamNet should not be used to convert magnitudes when the LLM does not explicitly write the conversion factor and relies on its internal knowledge; without the numeric value of the factor present in the text, OccamNet can't give the correct result.
6. OccamNet can't compute percentages, as this operation was not included in the training set of operations. It could be done in two steps.
7. Single-layer OccamNet can't compute a fraction of a number, as this would require two operations.
8. Single-layer OccamNet can't perform operations with more than 2 arguments.

| **User Query** | **Assistant Response** | **Cases** |
|---|---|---|
| An author writes 7.6 pages per session. After 9 sessions, the total pages written are | The author writes 7.6 pages per session. After 9 sessions, the author will have written 7.6 ×9 = **68.4** pages. The answer is 68. | 1, 2, 3 |


https://doi.org/10.52202/079017-1125

| User Query | Assistant Response | Cases |
|---|---|---|
| I have 10 oranges and 6 apple trees, each of them with 3 apples. How much fruit do I have? | The total number of pieces of fruit is 10 oranges + 6 trees times 3 apples = 10 + (6 ×3) = 10 + 18 = **28**. The answer is 28. | 1, 2, 4 |
| Sally has 7 pennies and 8 nickels. How many pennies does she have in total? | 1. Convert the number of nickels to pennies: 8 nickels is 40 pennies.<br>2. Add total number of pennies 40 + 7 = **47**.<br>The answer is 47. | 1, 2, 5 |
| What is 84 percent of 230.77? | 84% of 230.77 = 193.85<br>Explanation:<br>84 / 100 = **0.84**<br>0.84 ×230.77 = **193.85** | 1, 2, 6 |
| There were 39 cats on a rock. 3 boats came and carried away 5 cats each. How many cats were left? | There were originally 39 cats. 3 boats came and each took away 5 cats. So 3 ×5 = **15**.<br>39 - 15 = **24**.<br>Then 3/7 of them ran away. 3/7 of 24 is 9.<br>24 - 9 = **15**.<br>So there were 15 cats left. | 1, 2, 7 |
| 11 people have 2 apples, 2 oranges, and 2 peaches each. How many pieces of fruit do they have? | Each person has 2 + 2 + 2 = 6 pieces of fruit.<br>In total, they have 6 ×11 = **66**.<br>The answer is 66. | 1, 2, 8 |

### A.2 Training Hyperparameters

For all 1-layer OccamNet training runs, we used a batch size of 1, a learning rate of $6e-4$ and a weight decay parameter of $0.01$. We use gradient accumulation to achieve an effective batch size of $8$. We used a constant learning rate scheduler. We take 1000 samples from OccamNet per token.

For the 2-layer OccamNet run, we used a batch size of 1, a learning rate of $1e-4$ and a weight decay parameter of $0.01$. We use the gradient accumulation technique to achieve an effective batch size of $8$. We used a constant learning rate scheduler. We take 50,000 samples from OccamNet per token.

### A.3 Prompting

For the division arithmetic tasks, we found that the language models often did not return decimals. As such, we appended “Give the answer in decimals.” to these prompts. Similarly, for the trigonometric functions evaluations, we explicitly ask the language models to take the input as radians, by formatting the prompts as "cos(X rad) =".

For some models, we provide system prompting to guide the model toward the correct behavior. We break down prompting by model below:

**Llama 2/3:** We did not provide a system prompt for the arithmetic tasks. For the reasoning tasks, we used the system prompt “Solve step by step.”

**GPT 3.5 Turbo:** We do not use a system prompt for GPT 3.5 Turbo.

**GPT 4o:** We did not use a system prompt, except for the MATH401 dataset, where we noticed that GPT 4o was returning fractions instead of decimals. As such, on MATH401 we used the system prompt “Give your answer in decimals.”

**GPT 4o + Code:** We used the system prompt “Write and run code to answer math questions. Do not format numbers. Give all answers in decimals.”

**OccamLlama:** We experimented with OccamLlama prompts, but discovered that not including a system prompt was most effective.

### A.4 Generation parameters

For OccamLlama, Llama 2 7B and Llama 3 8B, we use the default values of $T = 0.6$ and Top-P $= 0.9$. For GPT 3.5 Turbo, GPT 4o, and GPT 4o with Code Interpreter, we use the default values of $T = 1.0$ and Top-P $= 1.0$.

### A.5 Experimental Results

Tables 5 and 6 show in more detail the accuracy of OccamLlama and other baselines on arithmetic and mathematical problem solving tasks. We measure accuracy as described in the main text.

We note here that on datasets with challenging arithmetic, in particular Multiarith Float and MATH401, OccamLlama 8B outperforms even GPT 4o and GPT 4o Code. In fact, on MultiArith Float, OccamLlama 8B is nearly 10 percentage points more accurate than GPT 4o + Code and and more than 40 percentage points more accurate than Llama 3 8B. Similarly, on MATH401, OccamLlama 8B is 7 percentage points more accurate than GPT 4o + Code and nearly 25 percentage points more accurate than Llama 3 8B. Although MATH401 does not include word problems, it does include some arithmetic expressions that require multiple calls to OccamNet to solve, meaning it requires both reasoning (to determine how to break up the arithmetic expression) and arithmetic capabilities.

The only datasets on which OccamLlama 8B performs substantially worse than Llama 3 8B are GSM8K [42] and Single Eq [43]. We believe a contributor to this is that these datasets include many problems that involve either fractions and percentages, which Llama does not convert to decimal format, or equations with unknown variables. As such, Llama often calls OccamNet with expressions such as "multiplying by 3/4 gives," "5% of this gives," or "adding 5 to both sides of x-5 = 11 gives." Because the switch is not trained on many examples like these in which the number is not in decimal format, it does not realize that OccamNet should not be used in these cases. Therefore, the switch triggers OccamNet, which is not capable of performing the correct operation (these types of operations are not acheivable with a 1-layer OccamNet). Future work could address this issue by training the switch with more data on this type of situation or by training an OccamLlama with a two layer OccamNet.

Finally, as noted in the main text, OccamLlama 70B achieves significant performance improvement over OccamLlama 8B across a number of benchmarks and outperforms GPT 4o and GPT 4o + Code on average. This demonstrates that OccamLLM improves with the base language model and suggests that combining OccamLLM with more capable models such as GPT 4o could be a promising avenue for future research.

Relative error is another important metric that complements accuracy. It measures by how much the answer differs from the true result. For two models with a similar accuracy metric, the relative error they achieve can be very different. Table 7 shows the relative error for the arithmetic experiments. An answer marked correct can have a nonzero relative error because of machine precision limits and because the answer does not report an infinite number of digits.

Interestingly, Llama 2 performs exceptionally poorly on division. By examining outputs, we see that this is because Llama 2 produces an approximately correct output but with the decimal place in the wrong position, leading to a result that is off by many orders of magnitude.

## B Example OccamLlama Generations

In this section, we include example OccamLlama 8B generations from the MATH401 and MultiArith-Float datasets. We randomly selected three examples for each dataset. OccamNet outputs are included in green. We omit prompt formatting to save space. Similarly, although outputs from OccamNet are always followed by "\n\n," we omit these newlines to save space, instead adding a period and space after each OccamNet generation.

By chance, all six responses happen to be correct.

 https://doi.org/10.52202/079017-1125

Table 5: Percent accuracy on arithmetic tasks. Higher is Better. Bold indicates best performance.

| | OccamLlama | Llama 2 7B Chat | Llama 3 8b Instruct | GPT 3.5 Turbo | GPT 4o | GPT 4o Code |
|---|---|---|---|---|---|---|
| Addition (3) | **100.0**±0.0 | 70.9±1.4 | 97.1±0.5 | 98.8±0.3 | **100.0**±0.0 | |
| Addition (5) | **100.0**±0.0 | 55.9±1.6 | 77.1±1.3 | 92.5±0.8 | 99.2±0.3 | |
| Addition (7) | **100.0**±0.0 | 19.2±1.2 | 44.9±1.6 | 65.2±1.5 | 95.7±0.6 | **100.0**±0.0 |
| Subtraction (3) | **100.0**±0.0 | 49.7±1.6 | 95.2±0.7 | 94.0±0.8 | 98.7±0.4 | |
| Subtraction (5) | **100.0**±0.0 | 22.9±1.3 | 58.8±1.6 | 86.3±1.1 | 92.6±0.8 | |
| Subtraction (7) | **100.0**±0.0 | 8.7±0.9 | 34.4±1.5 | 59.8±1.6 | 85.6±1.1 | 99.5±0.5 |
| Multiplication (3) | **100.0**±0.0 | 4.6±0.7 | 16.8±1.2 | 49.2±1.6 | 76.9±1.3 | |
| Multiplication (5) | **100.0**±0.0 | 0.0±0.0 | 0.1±0.1 | 0.4±0.2 | 4.6±0.7 | |
| Multiplication (7) | **100.0**±0.0 | 0.0±0.0 | 0.0±0.0 | 0.0±0.0 | 0.0±0.0 | 99.0±0.7 |
| Division (3) | **100.0**±0.0 | 20.8±1.3 | 71.7±1.4 | 50.5±1.6 | 78.2±1.3 | |
| Division (5) | **100.0**±0.0 | 7.4±0.8 | 48.1±1.6 | 15.7±1.2 | 51.0±1.6 | |
| Division (7) | **100.0**±0.0 | 2.8±0.5 | 35.3±1.5 | 10.7±1.0 | 38.6±1.5 | **100.0**±0.0 |
| Square Root (3) | **100.0**±0.0 | 1.2±0.3 | 14.8±1.1 | 47.1±1.6 | 69.3±1.5 | |
| Square Root (5) | **100.0**±0.0 | 0.2±0.1 | 1.3±0.4 | 11.9±1.0 | 23.6±1.3 | |
| Square Root (7) | **100.0**±0.0 | 0.0±0.0 | 0.0±0.0 | 0.9±0.3 | 18.6±1.2 | **100.0**±0.0 |
| Exponential | **100.0**±0.0 | 0.3±0.2 | 3.1±0.5 | 12.5±1.0 | 23.2±1.3 | **100.0**±0.0 |
| Logarithm | **100.0**±0.0 | 0.1±0.1 | 0.0±0.0 | 17.1±1.2 | 21.3±1.3 | **100.0**±0.0 |
| Sine | **100.0**±0.0 | 7.6±0.8 | 7.0±0.8 | 13.4±1.1 | 39.3±1.5 | **100.0**±0.0 |
| Cosine | **100.0**±0.0 | 0.8±0.3 | 1.5±0.4 | 6.7±0.8 | 32.8±1.5 | **100.0**±0.0 |
| AVERAGE | **100.0**±0.0 | 14.4±0.3 | 32.0±0.3 | 38.6±0.4 | 55.2±0.4 | 99.8±0.1 |

We also include three miscellaneous prompts and OccamLlama 8B generations to demonstrate OccamLlama's ability to handle more challenging prompts.

Table 6: Percent accuracy on reasoning tasks. Higher is Better. Bold indicates best performance.

| | OccamLlama 8B | OccamLlama 70B | Llama 3 8b Instruct | Llama 3 70B Instruct | GPT 3.5 Turbo | GPT 4o | GPT 4o Code |
|---|---|---|---|---|---|---|---|
| AddSub | 91.6±1.4 | 96.5±0.9 | 93.4±1.2 | 97.2±0.8 | 95.4±1.1 | 97.0±0.9 | **97.5±1.1** |
| GSM8K | 73.5±1.2 | 90.1±0.8 | 79.8±1.1 | 94.8±0.6 | 84.8±1.0 | **96.1±0.5** | 94.0±1.7 |
| MultiArith | 99.2±0.4 | 98.5±0.5 | **99.8±0.2** | 98.2±0.5 | 97.2±0.7 | 99.7±0.2 | 99.5±0.5 |
| MultiArith Float | **98.2±0.5** | 97.7±0.6 | 57.3±2.0 | 76.3±1.7 | 77.3±1.7 | 96.2±0.8 | 89.5±2.2 |
| MATH401 | 85.0±1.8 | **89.5±1.5** | 60.3±2.4 | 71.6±2.3 | 63.1±2.4 | 76.6±2.1 | 78.0±2.9 |
| Single Eq | 92.9±1.1 | 96.9±0.8 | 96.3±0.8 | 97.6±0.7 | 97.8±0.6 | 98.0±0.6 | **99.0±0.7** |
| SVAMP | 88.6±1.0 | 93.2±0.8 | 86.3±1.1 | 94.5±0.7 | 87.8±1.0 | 96.2±0.6 | **96.5±1.3** |
| AVERAGE | 89.9±1.2 | **94.6±0.9** | 81.9±1.5 | 90.0±1.2 | 86.2±1.4 | 94.2±1.0 | 93.4±1.7 |

### B.1 MATH401

**Example 1**

Prompt:

*7^4=*

Response:

*2401. 7 ^4 = 7 × 7 × 7 × 7 = 2401*

**Example 2**

Prompt:

*6**3=*

Response:

*216. 6 ^3 = 6 × 6 × 6 = 216*

**Example 3**

Prompt:

*63.0+0.074=*

Response:

*63.074. 63.0 + 0.074 = 63.074. So the answer is 63.074.*

### B.2 MultiArith-Float

**Example 1**

Prompt:

*Mike had 30 video games but 6 of them weren't working. If he wanted to sell the working games for 85.53 each, how much money could he earn?*

Response:

*Mike had 30 video games. 6 weren't working, so he had 30 - 6 = 24. He can sell 24 games for 85.53 each. 24 x 85.53 is 2052.720. So Mike could earn 2052.72 dollars.*


https://doi.org/10.52202/079017-1125

Table 7: Relative error (%) on arithmetic tasks. Lower is Better. Bold indicates best performance.

| | OccamLlama | Llama 2 7B Chat | Llama 3 8b Instruct | GPT 3.5 Turbo | GPT 4o | GPT 4o Code |
|---|---|---|---|---|---|---|
| Addition (3) | **0.0±0.0** | 50.5±10.9 | 3.2±1.9 | 0.3±0.1 | **0.0±0.0** | |
| Addition (5) | **0.0±0.0** | 113.0±21.1 | 23.7±4.0 | 4.6±1.8 | 0.0±0.0 | |
| Addition (7) | **0.0±0.0** | 310.3±97.0 | 78.1±16.2 | 4.0±1.4 | 1.0±0.9 | **0.0±0.0** |
| Subtraction (3) | **0.0±0.0** | 66.2±18.7 | 4.1±0.8 | 3.8±0.7 | 0.4±0.1 | |
| Subtraction (5) | **0.0±0.0** | 173.6±67.1 | 29.4±4.3 | 38.3±16.5 | 3.5±0.6 | |
| Subtraction (7) | **0.0±0.0** | 222.3±54.4 | 65.6±12.9 | 44.6±31.3 | 5.4±0.7 | 0.3±0.3 |
| Multiplication (3) | **0.0±0.0** | 7.6±0.7 | 1.9±0.5 | 1.8±0.4 | 0.1±0.1 | |
| Multiplication (5) | **0.0±0.0** | 84.9±0.9 | 46.7±1.7 | 19.2±3.7 | 1.8±0.4 | |
| Multiplication (7) | **0.0±0.0** | 98.9±0.2 | 74.9±1.8 | 90.1±24.2 | 4.4±0.6 | 1.0±0.7 |
| Division (3) | 0.1±0.0 | 1346.2±275.4 | 1.3±0.9 | 1.1±0.3 | **0.0±0.0** | |
| Division (5) | 0.2±0.1 | 174156.6±31687.6 | 9.5±1.8 | 0.7±0.2 | **0.1±0.0** | |
| Division (7) | 0.1±0.0 | 22032920.9±3642549.7 | 225.3±142.1 | 0.3±0.1 | 0.0±0.0 | **0.0±0.0** |
| Square Root (3) | **0.0±0.0** | 8.9±1.1 | 1.1±0.3 | 0.2±0.0 | 0.0±0.0 | |
| Square Root (5) | **0.0±0.0** | 72.8±4.8 | 12.6±1.7 | 0.1±0.0 | 0.0±0.0 | |
| Square Root (7) | 0.0±0.0 | 207.8±21.8 | 15.4±1.4 | 8.8±0.9 | 4.8±2.0 | **0.0±0.0** |
| Exponential | 0.3±0.0 | 422.6±82.6 | 11.7±0.7 | 2.3±0.9 | 0.1±0.0 | **0.0±0.0** |
| Logarithm | 0.0±0.0 | 138.3±11.9 | 40.2±1.3 | 6.4±4.0 | 0.1±0.0 | **0.0±0.0** |
| Sine | 0.1±0.0 | 515.5±68.1 | 279.2±21.9 | 125.4±12.3 | 3.2±0.7 | **0.1±0.0** |
| Cosine | 0.1±0.0 | 581.9±199.7 | 346.1±80.4 | 188.5±22.5 | 2.8±0.5 | **0.0±0.0** |
| AVERAGE | **0.1±0.0** | 1169210.8±194953.4 | 66.9±8.8 | 28.4±2.7 | 1.5±0.1 | 0.2±0.1 |

**Example 2**

Prompt:

> *While playing a trivia game, Adam answered 6 questions correct in the first half and 3 questions correct in the second half. If each question was worth 59.08 points, what was his final score?*

Response:

> *Adam got 6 questions correct in the first half, and 3 questions correct in the second half. So he got 6 + 3 = 9. Each question was worth 59.08 points. So his score is 9 * 59.08 = 531.720. The answer is 531.72.*

**Example 3**

Prompt:

*Lana's favorite band was holding a concert where tickets were 181.92 dollars each. Lana bought 5 tickets for herself and her friends and 4 extra tickets in case anyone else wanted to go. How much did she spend?*

Response:

*Lana bought 5 tickets for herself and her friends. Then 4 extra tickets. So in total she bought 5 + 4 = 9. Each ticket was 181.92 dollars. So 9 tickets will be 9 x 181.92 = 1637.280. The answer is 1637.280.*

### B.3 Miscellaneous prompts

These miscellaneous prompts demonstrate OccamLlama's ability to handle more challenging prompts. Examples 1 and 2 show that OccamLlama can handle prompts where the numbers are provided as words and not as decimals. This works because we can choose to parse strings such as "three" and "four" as 3 and 4, respectively, when parsing inputs to OccamNet. Example 3 shows that OccamLlama can handle prompts in which many irrelevant numbers are included. This works because OccamLlama uses Llama to reason and select the correct numbers to use before feeding these numbers into OccamNet.

**Example 1**

Prompt:

*Six minus seven =?*

Response:

*-1. Six minus seven is equal to -1.*

**Example 2**

Prompt:

*Three + four = ?*

Response:

*7. 3 + 4 = 7. Let me know if you have any other questions!*

**Example 3**

Prompt:

*In 1932, 100 planets were discovered and in 1933, another 200 were found. What is the total number of planets discovered between 1932 and 1933?*

Response:

*The number of planets discovered in 1932 was 100. The number of planets discovered in 1933 was 200. So the total number of planets discovered is 100 + 200 = 300. 300.*

## C OccamLLM Robustness Experiments

We find that OccamLlama displays remarkable generalization capabilities on out-of-distribution problems. To demonstrate this, we show below two out-of-distribution tasks on which OccamLlama performs remarkably well.

### C.1 Non-textual Training

First, we train the OccamNet decoder *from scratch*, using *only numeric expressions* and *absolutely no text at all*. This means that any problem with text, such as a word problem, is far out-of-distribution


https://doi.org/10.52202/079017-1125

Table 8: Accuracy on reasoning tasks. Higher is Better.

| | OccamLlama 8B | OccamLlama 8B Arith | Llama 2 7B Chat | Llama 3 8b Instruct | GPT 3.5 Turbo | GPT 4o | GPT 4o Code |
|---|---|---|---|---|---|---|---|
| AddSub | 91.6±1.4 | 92.7±1.3 | 78.0±2.1 | 93.4±1.2 | 95.4±1.1 | 97.0±0.9 | **97.5±1.1** |
| GSM8K | 73.5±1.2 | 71.6±1.2 | 36.0±1.3 | 79.8±1.1 | 84.8±1.0 | **96.1±0.5** | 94.0±1.7 |
| MultiArith | 99.2±0.4 | 98.5±0.5 | 76.0±1.7 | **99.8±0.2** | 97.2±0.7 | 99.7±0.2 | 99.5±0.5 |
| MultiArith Float | **98.2±0.5** | 95.3±0.9 | 23.3±1.7 | 57.3±2.0 | 77.3±1.7 | 96.2±0.8 | 89.5±2.2 |
| MATH401 | 85.0±1.8 | **85.8±1.7** | 43.9±2.5 | 60.3±2.4 | 63.1±2.4 | 76.6±2.1 | 78.0±2.9 |
| Single Eq | 92.9±1.1 | 92.1±1.2 | 79.1±1.8 | 96.3±0.8 | 97.8±0.6 | 98.0±0.6 | **99.0±0.7** |
| SVAMP | 88.6±1.0 | 88.8±1.0 | 61.5±1.5 | 86.3±1.1 | 87.8±1.0 | 96.2±0.6 | **96.5±1.3** |
| AVERAGE | 89.9±1.1 | 89.3±1.1 | 56.8±1.8 | 81.9±1.3 | 86.2±1.2 | **94.2±0.8** | 93.4±1.5 |

of the OccamNet decoder's training data. We test this model (using the standard router), which we denote OccamLlama 8B Arith, on the mathematical reasoning benchmarks and obtain remarkably good results, shown in Table 8.

The OccamLlama 8B Arith performs on par with the model trained with both numbers and text, even achieving higher accuracy on some benchmarks. This shows that the OccamLLM framework is robust, and points towards the fact that the representations of arithmetic that are built in the transformer body of the LLM and extracted by the OccamLLM Decoder are very general.

In contrast, we expect that finetuning Llama to perform arithmetic using only numeric examples and no text whatsoever would lead to extreme catastrophic forgetting and poor arithmetic performance on word problems. As such, we believe this data shows a remarkable generalization and robustness of OccamLLM.

### C.2 Multilingual Reasoning

To further demonstrate OccamLlama's generalization capabilities and also show that OccamLlama can handle non-English generation, we tested OccamLlama on the Multilingual Grade School Math Benchmark (MGSM) [46], a dataset consisting of GSM8K translated into 10 languages (Bengali, Chinese, French, German, Japanese, Russian, Spanish, Swahili, Telugu, and Thai). For these experiments, we prompted the LLMs to write their answers in the same language as the problem statement. Otherwise, the LLM would typically respond always in English, defeating the purpose of the experiment. We compute the drop in accuracy when switching from English to another language, given by the accuracy of a model on the English dataset minus the accuracy of the model on the dataset in a given language. The results are shown in Figure 5.

The table above shows that OccamLlama and Llama have similar performance drops between the English dataset and each non-English language dataset. On most languages and on average, OccamLlama has a smaller performance drop than Llama. The fact that OccamLlama (the decoders for which have never been trained on other languages) has a smaller average out-of-distribution performance drop than Llama (a model trained on over 750 billion tokens of non-English text) is in our opinion quite remarkable.

We believe that this test demonstrates OccamLlama's ability to handle many languages and its robustness against out-of-distribution data.

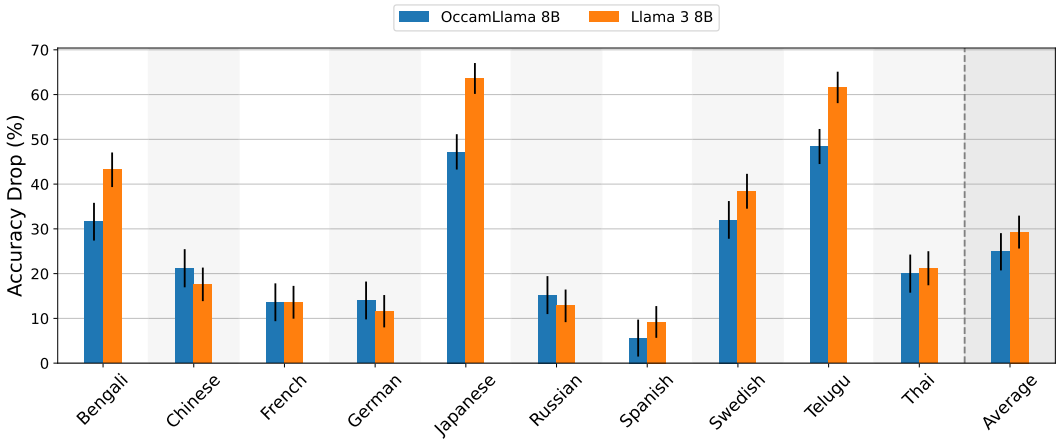


Figure 5: Model performance degradation for each language relative to English in the MGSM dataset. OccamLlama 8B's performance degradation is considerably less than Llama 3 8B's performance degradation, demonstrating strong multilingual and generalization capabilities.

# D Alternative Architectures and Losses

## D.1 Alternative Architectures

As discussed in the main text, although OccamLLM works most naturally with OccamNet, it can also work with other symbolic architectures such as the EQL network [47, 48], or architectures that can represent probability distributions over symbolic expressions, such as transformers [31] or recurrent neural networks (RNNs) [32].

However, in practice we believe OccamNet is the most effective architecture for this use case. We find that because EQL does not represent a probability distribution over functions, it easily gets stuck in local minima.

Regarding transformers and RNNs, we believe that OccamNet possesses a key advantage of being interpretable; simply by looking at the weights, it is possible for a human to determine which functions OccamNet assigns a high probability. We believe that this interpretability will make OccamNet easy for a decoder to initialize with the desired distribution. On the other hand, an RNN or transformer have substantially more complex relations between the weights and corresponding probability distribution, which we hypothesize would make learning a decoder for such models difficult.

This leads us to a key point: transformers and RNNs are effective for modeling complex multimodal distributions, but for this problem, we want to select a single function for each token, so the extra expressivity of these models is unneeded and likely detrimental to performance. We believe that OccamNet, a much simpler architecture, enables better parameter efficiency and performance.

## D.2 Alternative Losses

In this section we discuss alternative possible losses and how we arrived at the loss in Equation 3.

We considered two loss functions which are natural when optimizing a probability distribution: 1) a cross-entropy loss, and 2) a REINFORCE [34] loss. Each of these requires only a slight modification to reach Equation 3. This discussion thus illustrates how our loss combines benefits from both the cross-entropy and the reinforcement-learning losses.

**Cross-Entropy Loss** The cross-entropy loss is effective at modeling probability distributions. Given a ground truth distribution $q_x[f]$ conditioned on the input text $x$, the cross-entropy loss is given by

$$\mathcal{L}(x, y; W) = -\sum_f q_x[f] \log p_W[f]. \tag{4}$$


https://doi.org/10.52202/079017-1125

Unfortunately, for OccamLLM, the ground-truth distribution $q_x[f]$ is not uniquely specified. In particular the only constraints on $q_x[f]$ are that it is is normalized and satisfies $q_x[f] = 0$ if $f$ is not the desired function (i.e., $f(x) \neq y$). Since the same function can be represented in many ways in the OccamNet network (a property true of many function representations), multiple $f$ may satisfy $f(x) = y$, so $q_x$ is underparametrized.

The most natural choice for $q_x$ is to weight each valid function equally:

$$q_x[f] = \begin{cases} c_x & \text{if } f(x) = y \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where $c_x$ is a constant chosen such that $q_x$ is normalized, given by the inverse of the number of functions $f$ satisfying $f(x) = y$. However, determining $c_x$ requires testing every possible function $f$, which may be infeasible for large OccamNet networks. Further, this $q_x$ requires OccamNet to learn a superposition of functions, which may be challenging given its relatively low parameter count.

Another option is to choose a canonical form $f^*$ for each function and to set $q_x$ to be a 1-hot distribution that is nonzero only at $f^*$. Although this removes the challenge of learning a superposition, it still requires sampling nearly all functions in OccamNet due to the sparsity of $q_x$.

Ideally, we would like to find a $q_x$ with the following conditions:

- It enables the cross-entropy loss to be calculated by sampling from OccamNet. This allows us to avoid needing to iterate through and evaluate every $f(x)$ each time we compute the loss, since we can instead obtain a Monte-Carlo estimate.
- It is minimized when $p_W$ is a 1-hot probability distribution. This ensures that OccamNet can represent the optimal distribution.
- It has $q_x[f] \neq 0$ for all $f$ satisfying $f(x) = y$. This improves sample-efficiency by increasing the probability of sampling an $f$ with $q_x[f] > 0$.

A solution is to set

$$q_x[f] = \begin{cases} c_x p_W[f] & \text{if } f(x) = y \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

where $c_x$ is chosen such that $q_x$ is normalized. This gives a loss

$$\begin{aligned} \mathcal{L}(x, y; W) &= -\sum_f q_x[f] \cdot \log p_W[f] \\ &= -\sum_f c_x p_W[f] \cdot \delta(f(x) - y) \cdot \log p_W[f] \\ &\approx -\frac{c_x}{N} \sum_{f \sim p_W} \delta(f(x) - y) \cdot \log p_W[f] \\ &\approx -\frac{\sum_{f \sim p_W} \delta(f(x) - y) \cdot \log p_W[f]}{\sum_{f \sim p_W} \delta(f(x) - y)}, \end{aligned}$$

where

$$\delta(f(x) - y) = \begin{cases} 1 & \text{if } f(x) = y \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

and in the last step we used the fact that $c_x$ can be approximated as

$$c_x = \frac{1}{\sum_f p_W[f] \delta(f(x) - y)} \approx \frac{N}{\sum_{f \sim p_W} \delta(f(x) - y)}.$$

This loss is easily computed by sampling from $p_W$, it satisfies $q_W > 0$ for all $f$ satisfying $f(x) = y$, and it is minimized when $p_W$ is a delta function centered at any $f$ satisfying $f(x) = y$, as desired.

Note that

$$\mathcal{L}(x, y; W) = -\frac{\sum_{f \sim p_W} \delta(f(x) - y) \cdot \log p_W[f]}{\sum_{f \sim p_W} \delta(f(x) - y)} \tag{8}$$

is exactly the loss given in Equation 3 with $R(f(x), y) = \delta(f(x) - y)$. Thus, we have shown how Equation 3 can be interpreted as a cross-entropy loss. Equation 3 with general $R(f(x), y)$ can be seen as a cross-entropy loss with a "smoothed" ground truth distribution $q_x$ given by $q_x \propto p_W[f] \cdot R(f(x), y)$.

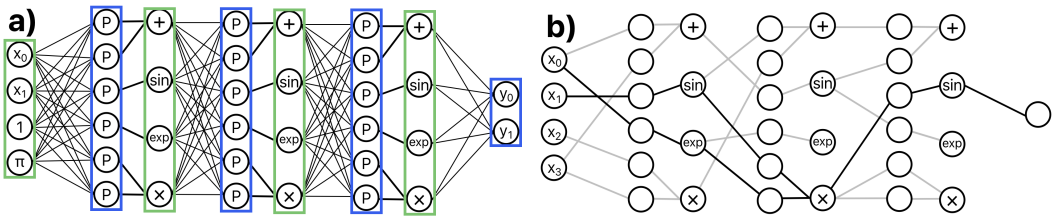


Figure 6: $a)$ An example OccamNet, with image layers boxed in green and arguments layers boxed in blue. We denote the inputs as the 0th image layer and the outputs as the $(L+1)$th arguments layer. Nodes in the arguments layers are represented with a P because of their probabilistic nature. $b)$ A demonstration of the dropped connections from sampled paths in OccamNet. All light grey paths are dropped from the final symbolic form of the sampled function because they are not directly connected to the outputs.

**REINFORCE Loss** Reinforcement-learning losses are effective for exploring large search spaces. We use a modification of the REINFORCE [34] loss because it is relatively simple to implement. Future work could explore more sophisticated variants of this algorithm, such as Proximal Policy Optimization [49].

The standard REINFORCE loss applied to OccamLLM gives

$$\mathcal{L}(x, y; W) = -\frac{1}{N} \sum_{f \sim p_W} R(f(x), y) \cdot \log p_W[f].$$

Note that for sparse $R$, there will be very few nonzero $R(f(x), y)$ sampled, so, since we are dividing by $N$, the gradient signal will be small. We modify REINFORCE by dividing by the the sum of the rewards for all samples instead of by $N$ to ensure that correct functions sampled only a few times still receive a large training step update. This once again produces Equation 3.

We find that using a delta function for our reward is most effective because it most accurately represents the sparse reward of the problem. Further, as shown above, this loss provides a Monte-Carlo estimate of the the cross entropy loss. Due to the sparse reward, many samples may initially be required to obtain an accurate estimate of the loss. However, as OccamNet approaches the desired distribution, the loss's sample efficiency will improve.

## E Background on OccamNet

This section is heavily modified from [11].

We divide this section into the following subsections:

1. In Section E.1, we describe OccamNet's architecture in more detail.
2. In Section E.2, we describe OccamNet's sampling process.
3. In Section E.3, we describe OccamNet's probability distribution.
4. In Section E.4, we describe OccamNet's initialization process.

### E.1 OccamNet Architecture

As described in the main text, we start from a predefined collection of $N$ primitive functions $\mathcal{P}$. OccamNet represents a distribution over compositions of functions in $\mathcal{P}$. From now on, we denote the $i$th primitive function in the $l$th layer as $\phi_i^{(l)}$. We begin indexing the primitives from 0 and the layers from 1, because we treat the inputs as the 0th layer. So, for example, in Figure 6a, $\phi_2^{(1)} = \phi_2^{(2)} = \phi_2^{(3)} = \sin$.

Each OccamNet layer consists of two sublayers, which we denote the *arguments* and *image* sublayers, shown in Figure 6a. For an $L$-layer OccamNet, each of these sublayers is reproduced $L$ times. The $l$th softmax layer connects the $(l-1)$th image layer with the $l$th arguments layer. For $1 \leq l \leq L$, we


https://doi.org/10.52202/079017-1125

denote the $l$th arguments sublayer hidden state as $\widetilde{\mathbf{h}}^{(l)}$ and the $l$th image sublayer hidden state as $\mathbf{h}^{(l)}$. So, $\widetilde{\mathbf{h}}^{(2)}$ would represent the middle layer of nodes labeled $P$ in Figure 6a. We further write

$$\widetilde{\mathbf{h}}^{(l)} = \left[\widetilde{h}_1^{(l)}, \ldots, \widetilde{h}_{M^{(l)}}^{(l)}\right]^\top, \quad \mathbf{h}^{(l)} = \left[h_1^{(l)}, \ldots, h_{N^{(l)}}^{(l)}\right]^\top, \tag{9}$$

where

$$M^{(l)} = \sum_{0 \le k < N^{(l)}} \alpha\left[\phi_k^{(l)}\right],$$

$N^{(l)}$ is the number of primitives in layer $l$, and $\alpha[\phi]$ is the arity of function $\phi$. We also define $\mathbf{h}^{(0)}$ to be the input layer (an image sublayer) and $\widetilde{\mathbf{h}}^{(L+1)}$ to be the output layer (an arguments sublayer).

In a standard OccamNet layer, each primitive is repeated exactly once in each layer. However, in Complete OccamNet, each primitive in the $l$th layer is repeated $A^{L-l}$ times, where $A$ is the maximum arity of the primitives. This is shown in Figure 7 in the transition from 7a to 7b. Complete OccamNet also concatenates each image layer to the next image layer, as shown in Figure 7c.

### E.2 Sampling from OccamNet

In this section, we more carefully describe OccamNet's sampling process. We sample a connection to each arguments layer node from the distribution given by the softmax of the softmax-layer weights leading to that node. In particular, if $\mathbf{w}_i^{(l)}$ are the weights of the $l$th softmax layer leading to the $i$th node of the $l$th argument's layer, when we sample we produce a sparse matrix

$$\mathsf{SAMPLE}\left(\begin{bmatrix} \mathsf{softmax}(\mathbf{w}_1^{(l)}) \\ \vdots \\ \mathsf{softmax}(\mathbf{w}_{M^{(l)}}^{(l)}) \end{bmatrix}\right) \tag{10}$$

where the SAMPLE function samples a one-hot row vector for each row based on the categorical probability distribution defined by $\mathsf{softmax}(\mathbf{w})$. To evaluate this sample, we simply evaluate a forward pass through the network, treating the sampled sparse matrices from the softmax layers as the weights of linear layers:

$$\widetilde{\mathbf{h}}^{(l)} = \begin{bmatrix} \widetilde{h}_1^{(l)} \\ \vdots \\ \widetilde{h}_{M^{(l)}}^{(l)} \end{bmatrix} \equiv \mathsf{SAMPLE}\left(\begin{bmatrix} \mathsf{softmax}(\mathbf{w}_1^{(l)}) \\ \vdots \\ \mathsf{softmax}(\mathbf{w}_{M^{(l)}}^{(l)}) \end{bmatrix}\right) \mathbf{h}^{(l-1)}, \tag{11}$$

To complete the picture of the forward pass, we formalize how we deal with activations accepting multiple inputs. We define the action of the activation functions as follows:

$$h_i^{(l)} = \phi_i^{(l)}\left(\widetilde{h}_j^{(l)}, \ldots, \widetilde{h}_{j+\alpha[\phi_i^{(l)}]-1}^{(l)}\right), \quad j = \sum_{0 \le k < i} \alpha\left[\phi_k^{(l)}\right]. \tag{12}$$

### E.3 OccamNet's Probability Distribution

OccamNet parametrizes a probability distribution over all functions which it can sample. In particular, when OccamNet samples a function, it is really sampling a directed acyclic graph (DAG) which defines a computational path to compute a function. The probability of sampling a computational graph is equal to the product of the probabilities of the connections in the DAG which are connected to the output node.

Note that multiple computational graphs can correspond to the same function. In this paper, when we refer to a function sampled from OccamNet or the probability of a function according to OccamNet, we use function as a shorthand for a *particular* computational graph corresponding to that function.

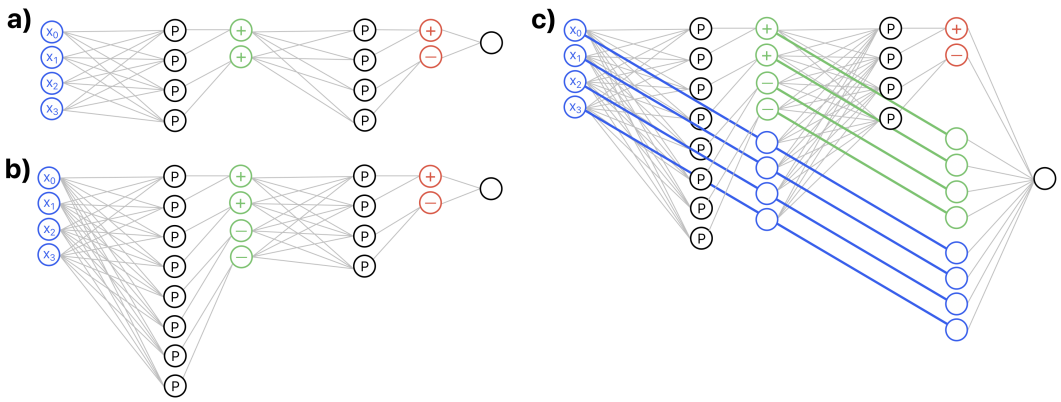


Figure 7: The progression of enhancements leading to a Complete OccamNet from a standard OccamNet. a) A standard OccamNet without repeated activations or skip connections. b) The same OccamNet as in a) with activations repeated in earlier layers. c) The same OccamNet as in b) with added skip connections. This is a Complete OccamNet.

Although this underspecifies the computational graph in question, this is never an issue because we always refer to functions in abstract.

When using OccamLlama for inference, we select the maximum probability function by sampling 100 functions from OccamNet, evaluating their probabilities as described above and selecting the maximum one.

### E.4 Initialization

This section describes how we calculate $\mathbf{W}^*$ from the main text. We wish to initialize $\mathbf{W}^*$ such that $p_{W^*}[f_1] = p_{W^*}[f_2]$ for all $f_1$ and $f_2$. Below, we assume that skip connections do not exist. However, the algorithm also works for skip connections, requiring only a small modification to Equation 14.

Unfortunately, such an initialization is impossible for any OccamNet with two or more layers containing primitives with more than one argument. However, it is possible to initialize OccamNet such that a lower bound $q_{W^*}$ of the true probability $p_{W^*}$ is independent of $f$.

Define the probability of a function $f$ up to a given node as the product of the probabilities of the edges that lead to that node in the DAG of $f$. Intuitively, $q_W[f]$ approximates $p_W[f]$ by maintaining a lower bound on the probability of $f$ up to each node of an OccamNet and propagating that lower bound through the computational graph given by $f$.

To define $q_W$ more precisely, let $q_i^{(l)}[f]$ and $\widetilde{q}_i^{(l)}[f]$ be the probability bounds corresponding to the $i$th node of the $l$th image or arguments sublayer. We have suppressed the dependence on $W$ for notational convenience. We compute these probabilities starting with the inputs, for which we set $q_i^{(0)} = 1$. We then propagate probabilities to the arguments layers according to

$$\widetilde{q}_i^{(l+1)} = \mathsf{softmax}(\mathbf{w}_i^{(l+1)})_j q_j^{(l)}, \tag{13}$$

where $j$ is the node in the $l$th image layer which $f$ connects to the $i$th node of $(l+1)$th arguments layer. Similarly, we propagate probabilities to the image layers according to

$$q_i^{(l)} = \prod_{k=n}^{n+\alpha[\phi_i^{(l)}]-1} \widetilde{q}_k^{(l)}, \quad n = \sum_{j=1}^{i-1} \alpha\left[\phi_j^{(l)}\right]. \tag{14}$$

Finally, we define $q_W[f] = q_0^{(L+1)}[f]$.

In practice $q_W[f] \leq p_W[f]$, where equality holds for many functions. In fact, $q_W[f] < p_W[f]$ only when part of the DAG of $f$ is used as input to two different arguments nodes. In cases such as these, the portion of the DAG that is used twice multiplicatively contributes the probability of its edges to $q_W[f]$ twice, artificially suppressing its value. However, because $q_W[f]$ is a lower bound, initializing


https://doi.org/10.52202/079017-1125

$W^*$ to equalize $q_{W^*}$ still has the desired effect of ensuring adequate coverage for each $f$ in the initial probability distribution of OccamNet.

With this primer, we can now define the algorithm to initialize $W^*$ such that $q_{W^*}[f]$ is uniform.

The algorithm traverses through OccamNet layer by layer and establishes as an invariant that, after assigning the weights up to the $l$th layer, $\widetilde{q}_i^{(l)}[f]$ are equal for all $i$ and $f$. This implies that, after assigning the weights up to the $l$th layer, $q_i^{(l)}[f]$ are equal for all $f$, but not necessarily for all $i$. We denote the common value of $\widetilde{q}_i^{(l)}[f]$ as $\widetilde{q}^{(l)}$ and the common value of $q_i^{(l)}[f]$ as $q_i^{(l)}$.

The algorithm starts with input layer, where $q_i^{(0)} = 1$ automatically. Once the invariant above is true for a given $l$, the algorithm sets

$$\left(\mathbf{w}_i^{*(l+1)}\right)_j = \log\left(\frac{\min_k\left(q_k^{(l)}\right)}{q_j^{(l)}}\right) \tag{15}$$

for all $i, j$, where $\left(\mathbf{w}_i^{*(l+1)}\right)_j$ denotes the weight connecting the $j$th node in the $l$th image layer to the $i$th node in the $(l+1)$th arguments layer. This establishes the invariant for $l+1$ because

$$\begin{aligned} q_j^{(l)}\mathsf{softmax}(\mathbf{w}_i^{*(l+1)})_j &= \frac{q_j^{(l)}\exp\left[\left(\mathbf{w}_i^{*(l+1)}\right)_j\right]}{\sum_k \exp\left[\left(\mathbf{w}_i^{*(l+1)}\right)_k\right]} \\ &= \frac{q_j^{(l)}\min_k\left(q_k^{(l)}\right)/q_j^{(l)}}{\sum_k \min_m\left(q_m^{(l)}\right)/q_k^{(l)}} \\ &= \frac{1}{\sum_k 1/q_k^{(l)}}, \end{aligned}$$

which is a constant over both $i$ and $j$, so $\widetilde{q}_i^{(l+1)}[f]$ is a constant over both $i$ and $f$. The algorithm repeats the above procedure until it has traversed the entire network.

In summary, the algorithm involves the following steps:

1. Set $l = 0$ and $q_i^{(l)} = 1$.
2. Increment $l$ by 1.
3. Set $\mathbf{W}^{*(l)}$ according to Equation 15.
4. If $l < L+1$, Compute $\widetilde{q}^{(l+1)}$ and $q_i^{(l+1)}$.
5. Return to step 2 until $l = L+1$.

## NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Claims made in the abstract and introduction are supported by experiments in the results section.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We address the limitations of our method in the limitations section. We also provide some discussion on room for improvement in the discussion section.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory Assumptions and Proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?


https://doi.org/10.52202/079017-1125

Answer: [NA]

Justification: The paper does not have theoretical results.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental Result Reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Details to the dataset generation (where neccessary) and the experiments are provided in the main text and appendix. We also provide code to reproduce results.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We include code and data at `https://github.com/druidowm/OccamLLM`, as highlighted in the abstract. We also describe our experiments in detail in the appendix.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental Setting/Details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [Yes]

Justification: All experiment hyperparameters and configuration are detailed in Appendix A and referenced in the main text.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment Statistical Significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: All results are reported with error bars. They are defined in the main text.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).


https://doi.org/10.52202/079017-1125

- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Compute resources used (particularly, the GPUs used) are detailed in Appendix A.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Code of Ethics were reviewed and conformed to.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: This is discussed in the broader impacts section of the paper.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: Our work does not release a pre-trained model and new datasets created in this work are purely arithmetic.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All data and models used are properly cited and their license terms were properly respected.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.


https://doi.org/10.52202/079017-1125

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: New data created in this work and trained models are documented in the paper and/or in the public GitHub repository.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: Our work does not involve human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing or research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.


https://doi.org/10.52202/079017-1125