Full Length Article

Arithmetic with language models: From memorization to computation

Davide Maltoni, Matteo Ferrara *

Department of Computer Science and Engineering, University of Bologna, Italy

ARTICLE INFO

ABSTRACT

A better understanding of the emergent computation and problem-solving capabilities of recent large language models is of paramount importance to further improve them and broaden their applicability. This work investigates how a language model, trained to predict the next token, can perform arithmetic computations generalizing beyond training data. Binary addition and multiplication constitute a good testbed for this purpose, since they require a very small vocabulary and exhibit relevant input/output discontinuities making smooth input interpolation ineffective for novel data. We successfully trained a light language model to learn these tasks and ran a number of experiments to investigate the extrapolation capabilities and internal information processing. Our findings support the hypothesis that the language model works as an Encoding–Regression–Decoding machine where the computation takes place in the value space once the input token representation is mapped to an appropriate internal representation.

1. Introduction

Large Language Models (LLMs) based on Transformer architecture (Vaswani et al., 2017) have recently demonstrated surprising problem-solving capabilities that require logic reasoning, advanced information processing and common sense (Bubeck et al., 2023; Wei et al., 2022, 2022). Their huge storage capacity combined with a massive training on terabytes of heterogeneous data could suggest that the memorization of an enormous amount of knowledge is enough to perform well on similar test data. However, validations on carefully selected Out-of-Distribution (OoD) data proved their reasoning capabilities on novel examples requiring non-trivial generalizations. Unfortunately, the depth and width of such models is so high that decoding and understanding the internal information processing is very challenging.

Focusing on arithmetic calculations, some studies (Yuan et al., 2023) demonstrate that recent LLMs (such as GPT-4) can perform additions and multiplications with long-digit operands, for which the number of variants is so high to exclude the exhaustive memorization of the training set. Nevertheless, the computational approach put in place by LLMs, as well as the interpolation/extrapolation capabilities remain unexplained.

In this work we design some controlled experiments, consisting of simple computation tasks such as binary addition and multiplication, and solve them with two Language Models (LMs) based on Transformer architecture: (i) the original encoder–decoder architecture by Vaswani et al. (2017) and (ii) a more recent decoder-only architecture denoted as nanoGPT (Karpathy, 2022). In spite of their simplicity, these tasks cannot be solved by pure memorization or smooth interpolation and investigating how an LM learn them can improve our understanding of the underlying mechanisms. In particular, using a tiny vocabulary of just 5 tokens and a small training set allows to operate with a light (non-pretrained) LM and use interpretability techniques to investigate internal information processing.

Other studies addressed the ability of LLMs to perform arithmetic computation and train small LMs to learn these tasks from scratch (see related works in Section 2). However, our aim is different: we are not interested in finding the best LM architecture and setup to maximize accuracy on arithmetic operations, but we look for a simple architecture and setup that allow to effectively solve the task in order to be able to investigate the underlying computational approach. The main novelty and contribution of this work are the formalization of the hypothesis that our LM works as an Encoding–Regression–Decoding machine and the design of a number of experiments to support and validate this hypothesis (see Table 1).

After presentation of related works in Section 2, in Section 3 we introduce the experimental testbed and the architecture of the LM used. Section 4 presents the results achieved and introduces control experiments and elaborations to shed light on the computation approach used to solve the tasks. In Section 5 an ablation study is presented and, finally, in Section 6 we include a final discussion and draw some conclusions.

---

* Corresponding author.
 E-mail addresses: davide.maltoni@unibo.it (D. Maltoni), matteo.ferrara@unibo.it (M. Ferrara).

Table 1

The main contributions of this work.

Step	Aim	Where
Training an LM on addition and multiplication	Demonstrating these arithmetic problems can be solved with a simple LM trained form scratch	Section 4.1
Manipulating training set by excluding specific regions of the input space	Evaluating interpolation/extrapolation capabilities and making hypothesis on internal regression	Section 4.4
Correlation analysis of internal values (embeddings)	Support hypothesis that LM works as an ERD machine	Section 4.5
Amnesic probing	Prove that the "value" information is crucial to properly compute the output	Appendix D

2. Related works

2.1. LM and LLM capabilities on arithmetic tasks

In Yuan et al. (2023) recent LLMs have been benchmarked in arithmetic tasks, including long-digits sum and multiplication, showing that LLMs such as ChatGPT and GPT-4 can perform reasonably well on these tasks even with no specific tuning. On the other hand, the accuracy of smaller models is markedly lower, and in general they are not able to work with long operands and generalize to OoD data.

Goat (Liu & Low, 2023) a LLaMA model specifically fine-tuned on arithmetic tasks performed even better than GPT-4 on large-number additions and subtractions, probably due to the consistent (digit level) tokenization of numbers in LLaMA models. However, it was able to perform multi-digits multiplication and division only forcing a Chain of Thought (CoT) (Wei et al., 2022) decomposition of such tasks during instruction tuning.

Nogueira et al. (2021) tuned a T5-based pre-trained LM on additions and subtractions, and argued that tokenization and input representation are critical to achieve good accuracy. In particular, in their experiments character-based tokenization works better than sub-word tokenization, and making explicit the digit position in the input string (i.e., inserting after each digit a marker to denote its position in the sequence) generally leads to better accuracy. They also trained a vanilla non-pretrained LM on smaller numbers and found that classical sinusoidal-based positional embedding does not perform well, so they proposed a tailored position-wise masked embedding. Their paper contains other interesting findings such as the impact of the digit order (plain or reverse) and the size of the training set.

Muffo et al. (2023) tuned pre-trained GPT-2 models on 5-digit additions and 2-digit multiplications. They also found that making explicit the digit position in the input sequence helps to improve accuracy. While good accuracy is reported for addition, the tuned models struggle to learn multiplication even on two-digit operands.

Lee et al. (2023) trained small LMs to learn arithmetic tasks, mainly focusing on addition, but also experimenting with subtraction, multiplication, sine and square root. The authors carefully ablated different aspects of the training data to isolate the factors that contribute to the appearance of arithmetic capabilities. In particular, they studied the impact of the input order (plain or reverse) and the utility of providing intermediate information about the decomposition of the task in steps to promote CoT reasoning. Some results and findings included in Lee et al. (2023) will be further discussed throughout this paper.

All the above works provide useful contributions to understand the capabilities and limitations of large and small LMs to deal with arithmetic tasks, but none of them focus on the computational approach used to solve them, which is the main purpose of the present work (see Table 1).

2.2. Interpretability techniques

A large number of techniques can be used to investigate the internal working mode of deep neural networks, including Transformers and LMs: see Räuker et al. (2023) for a recent survey. Weights, single neurons, subnetworks/circuits, and activations can be the target of *intrinsic* approaches (implemented during training) or *post-hoc* approaches (implemented after training).

Probing is a common technique used to investigate the representations learned by pre-trained LMs: it typically involves training a simple model (denoted as *probe*) on top of the LM embeddings to predict a given property (Belinkov, 2022). Moreover, structural probing can be used to check whether internal representations encode discrete structures such as syntax trees (Hewitt & Manning, 2019; White et al., 2021). However, a certain criticism emerged on probing analyses which is believed to disconnect the probing task from the original one and/or to reveal correlations instead of causations. Therefore, instead of focusing on the presence of information on internal encoding, some researchers proposed to check whether the removal of some knowledge from embeddings (e.g., amnesic probing Elazar et al., 2021) negatively influences the model ability to perform a task (Elazar et al., 2021; Lasri et al., 2022). Other interesting approaches to interpretability are mechanistic interpretability (Elhage et al., 2021) and causal abstraction (Geiger et al., 2021): the former is aimed at reverse engineering the algorithm that a model uses to solve a task and to map it to neural circuits; the latter constructs an interpretable causal model and aligns it with neural representations.

In this work we use a mix of intrinsic and post-hoc interpretability techniques: in particular through the experiments we manipulate the training set, change the input representation and the architecture components, perform correlation analyses of embeddings and apply amnesic probing.

2.3. Interpretability of arithmetic reasoning with LMs

Stolfo et al. (2023) introduced a causal mediation analysis to point out the LM components (e.g., attention heads, Multi-Layer Perceptrons — MLPs) involved in the information processing of simple arithmetic operations, focusing on the flow of numerical information throughout the model layers/columns. The main outcomes of this study are that the model: (i) processes the representation of numbers and operators with the first layers; (ii) information is then conveyed (by attention heads) to the last part of the sequence (i.e., output column), where (iii) it is numerically processed by late MLPs.

Nanda et al. (2023) carefully studied the algorithmic approach put in place by a small Transformer to implement modular addition of small numbers. They discovered that the internal algorithmic implementation is based on discrete Fourier transforms and trigonometric identities to convert addition to rotation on a circle. While the outcomes are somewhat surprising, here the term algorithm must be taken with care: even if the experiments prove that internal processing well approximates given equations, the approach is a numerical approximation (based on weight encoded values) that does not generalize to different moduli (as a symbolic implementation of the equations could do).

Both these studies adopted a simplified setting where numbers are presented as single token, and the output is expected at the last position of the sequence. So the models are not operated in autoregressive manner and the multi-token encoding/decoding stages are simplified. In Section 6 we discuss how the above findings are compatible with our findings.

3. Experiment design

3.1. The tasks

We focused on two simple computation tasks: binary addition and binary multiplication. Using binary encoding allows keeping the vocabulary very compact, since we need to encode only the symbols '0', '1' and a few other tokens. The selected tasks have other nice properties such as computing input similarities by Hamming distance and easily generating all combinations. Of course, a classical artificial neural network can be trained to learn to sum and multiply two integers or floating-point numbers, but adding/multiplying strings of tokens with an LM is trickier.

More formally, given two integers A, B (both in the range $[0, 127]$) our input sequence (or prompt) is a 15-token string taking the form:

$$a_0 a_1 a_2 a_3 a_4 a_5 a_6 \langle op \rangle b_0 b_1 b_2 b_3 b_4 b_5 b_6$$

where $a_i, b_i \in \{\text{'0'}, \text{'1'}\}$ are the symbols corresponding to bits in the ith position in the binary representation of A and B respectively, and $\langle op \rangle$ can be either '+' or '×'.

The expected output string (or input completion) is:

$$R = r_0 r_1 ... r_{m-1}$$

where r_i is the ith bit in the binary representation of $A \langle op \rangle B$, and m is the number of bits of the expected output string R (8 and 14 for addition and multiplication, respectively).

It is worth noting that:

- we are using a fixed-length input/output representation (with zero padding for unused most significant bits) to make the digit positions more explicit.
- in both the input and output the Least Significant Bits (LSBs) are provided before the Most Significant Bits (MSBs) (a.k.a., reverse or little-endian order) since this was supposed to simplify the model learning.[1] As discussed in Appendix C this assumption leads to a much faster training.

If we consider the sequence-to-sequence mapping underlying the proposed tasks we note that even in a simple binary addition a slight change in the input (i.e., a single bit) can produce a relevant change in the output because of the carries propagation. In the example below a single bit modification in the input produces an 8 bit modification in the output:

$$1000000 + 0111111 \rightarrow 11111110$$
$$1000000 + 1111111 \rightarrow 00000001$$

Such input–output discontinuity is made more explicit for addition in Appendix A.

3.2. The architecture

A non-pretrained encoder–decoder Transformer based on the original architecture introduced in Vaswani et al. (2017) was used as primary LM. Table 2 reports the model setup and parametrization. The small vocabulary used allows us to keep the model small (just 701 K learnable parameters) and trainable from scratch with a limited number of examples.

The LM was trained to learn separately the addition/multiplication tasks. For both problems, we exhaustively generated all the $2^{14} = 16384$ input/output combinations, which were then randomly split into training ($3/4 \rightarrow 12288$) and validation ($1/4 \rightarrow 4096$) sets. In our

---

[1] In binary arithmetic the addition/multiplication algorithms start processing the LSBs in order to correctly propagate the intermediate carries.

Table 2
Details of the LM model used in our experiments. The total number of learnable parameters is just 701K, which is several orders of magnitudes smaller than recent billion-parameters LLMs.

vocabulary size	5
vocabulary	0: unused, 1: <start>, 2: '+' or '×', 3: '0', 4: '1'
token embedding	learned
positional encoding	fixed (sinusoidal)
d_{model}	64
d_{ff}	$d_{model} \times 4$
num_heads h	8
encoder layers	6
decoder layers	6
dropout	0.1
learnable parameters	701K

experiments we do not need a separate dataset to tune hyperparameters so our validation set coincides with the test set.

An additional control experiment was run where the input sequences were the same of the addition experiment but the output completion was randomly generated (with the same length as the addition, i.e., 8 tokens). In this case, the lack of any dependencies between input and output makes it impossible to learn an algorithmic approach (or smooth mapping) to solve the problem and the only strategy to learn the training set is memorizing all the sequences.

When the trained LM is used in inference mode, we always pick the most probable token from the logit outputs (i.e., greedy decoding). Two metrics can be used to denote the LM accuracy: *token accuracy* refers to the probability of generating the next token correctly, while *sequence accuracy* refers to the probability of generating the whole output string correctly in autoregressive mode (i.e., generating one token at a time and appending it to the current prompt).

Most of the experiments have been repeated with a second LM (nanoGPT by Karpathy, 2022) which is a good representative of the decoder-only family. Details are reported in Appendix E.

All the experiments included in this paper can be easily reproduced by running the code available on Github.

4. Results

4.1. Learning addition and multiplication

Fig. 1 shows that our simple LM is able to learn addition in less than 50 epochs, and multiplication in about 250 epochs.[2] As expected multiplication is more complex and requires more training: this is due to the high non-linearity of this operation (more on this later) and to the higher length of the output (14 vs 8 tokens). The accuracy on the validation set is very close to the training set, denoting almost perfect generalization on numbers never seen before. This is a somewhat surprising result, especially considering the limited size of the training data. No grokking[3] was observed (Nanda et al., 2023). Similar results were obtained with nanoGPT (see Fig. E.7 in Appendix E.)

Unlike Nogueira et al. (2021) (see their Appendix B for a similar setup), we were able to learn addition with the native sinusoidal positional encoding. Moreover, in Lee et al. (2023) additions can be effectively learnt by a simple LM, but to reach 100% accuracy the training set had to be balanced in terms of the operand magnitude (i.e., number of digits) and carry propagation. The effectiveness of our training procedure is probably due to the lower complexity determined

---

[2] We used the standard CrossEntropy loss, the Adam optimizer with the learning rate of 0.0001 and betas = 0.9 and 0.98, and a minibatch size of 128.

[3] Grooking refers to the case where validation accuracy, much smaller than training accuracy at initial stages, suddenly increases after a certain number of epochs.
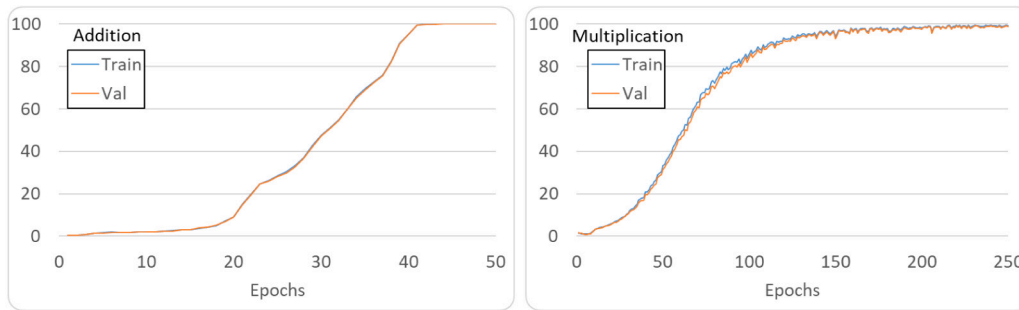
Fig. 1. Sequence accuracy. From the left: addition and multiplication. Results are averaged over five runs. Note that, training and validation curves are almost overlapped. At the end of training the Mean Absolute Error (MAE) on the validation set, between the real and generated operation results, is 0 and 1.3 for addition and multiplication, respectively.
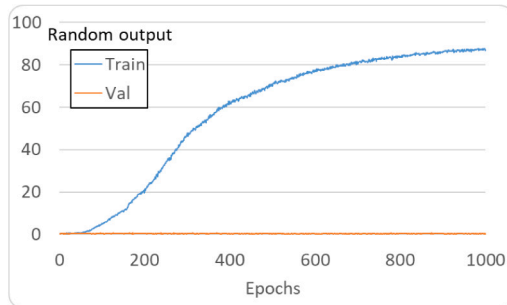

Fig. 2. Sequence accuracy using random output in the training set. Results are averaged over five runs.

by a small vocabulary and fixed-length representation. As to multiplication, Muffo et al. (2023) were not able to effectively learn two (decimal) digits multiplication, while Lee et al. (2023) and Liu and Low (2023) had to provide extra intermediate steps in the prompt (denoted as *detailed scratchpad*) or during instruction tuning, respectively. On the contrary our model effectively learnt multiplication of 7 binary digit operands: again the simplified setup may have been the key.

On the workstation used (with a single Titan RTX GPU) training can be completed in just 8 and 46 min for addition and multiplication, respectively. An estimation of the training complexity C of an LLM in term of floating point operations is $6 \times N \times T$ (Kaplan et al., 2020), where N is the number of model parameters (about 701 K as reported in Table 2) and T the number of training tokens. T can be obtained as the product of the training set size (12288 in our experiments — see Section 3.2), the sequence length in tokens (23 and 29 for addition and multiplication, respectively — see Section 3.1) and the number of epochs (50 and 250 for addition and multiplication, respectively). Hence, for addition T is 14M ($12288 \times 23 \times 50$) and therefore C is about 59×10^{12} operations while for multiplication T is 89M ($12288 \times 29 \times 250$) and C is about 374×10^{12} operations.

4.2. Control experiment: random output

If the output is randomly generated and therefore there is no relation with the input, the only possibility of learning the training set is by memorizing the whole data. Fig. 2 shows the training results: a much larger number of epochs (i.e., 1000) were necessary to reach a sequence accuracy of 87.8%, and, as expected, the validation accuracy did not increase over the epochs. The difficulty of memorizing the training set (many more epochs) is due to the high discontinuity of the input–output mapping. In fact, because of the random output generation, very similar input sequences can be associated to completely different outputs.

Therefore, even if we only consider the accuracy on the training set, this result shows that an exhaustive memorization of the input is much more complex for the LM than solving the addition and multiplication

tasks. This leads us to assume that, to efficiently solve the above computation tasks, the LM has found a computational approach (or algorithm) to simplify the output prediction. Now the question is: what is the approach?

4.3. The computational approach

Let us consider two alternative approaches:

Symbolic Manipulation (SM): a first idea is that the LM could learn the binary integer addition/multiplication algorithms used by an ALU inside a CPU (see Appendix B for a short reminder). Indeed, the addition algorithm is not complex and can be solved by using a 3-bit truth table (to sum each pair of corresponding bits with the carry-in) and iterative carry-out propagations. However, multiplication (by iterative additions) is much more complex and trickier to learn by using a symbolic manipulation approach. Furthermore, as shown in Lee et al. (2023), a simple LM can also learn complex operations such as the sine function or the square root, whose mathematical (and algorithmic) decomposition is very complex since they require Taylor expansion and Newton method, respectively.

Encoding–Regression–Decoding (ERD): if we consider the model architecture (Transformer) used for the LM and the underlying word embedding by vector representations, it is more likely that the LM solves the problem by decomposing it in the following three phases:

1. Encoding (token to value): maps the input sequence (i.e., $a_0 a_1 a_2 a_3 a_4 a_5 a_6 \ \langle op \rangle \ b_0 b_1 b_2 b_3 b_4 b_5 b_6$) to a suitable vector representation. In principle, two vectors $\mathbf{v}_A$ and $\mathbf{v}_B$ representing the values (or magnitudes) of A and B are enough.
2. Regression: learns the computation as a supervised regression problem in the vector space: $\mathbf{v}_R = regress(\mathbf{v}_A, \mathbf{v}_B)$. Actually this regression formulation is an oversimplification of the problem since in the next-token-prediction training the LM works incrementally. In Appendix C this discussion will be expanded.
3. Decoding (value to token): maps the value vector $\mathbf{v}_R$ back to token representation (i.e., $r_0 r_1 ... r_m$).

It is worth noting that the above Encoding and Decoding phases do not need to be mapped onto the Transformer encoder and decoder (more on this later). The experiments reported in Sections 4.4 and 4.5 support the ERD assumption. The capability of capturing number magnitudes by pretrained embedders was also investigated by Wallace et al. (2019) who successfully trained a simple external regressor to compute the sum of two numbers starting from their embeddings. Other interesting studies on capturing numeracy with embedding were carried out by Naik et al. (2019) and Sundararaman et al. (2020).

4.4. Interpolation vs extrapolation

The random training/validation split performed for the experiments reported in Section 4.1 constitutes a somewhat simplified testbed to
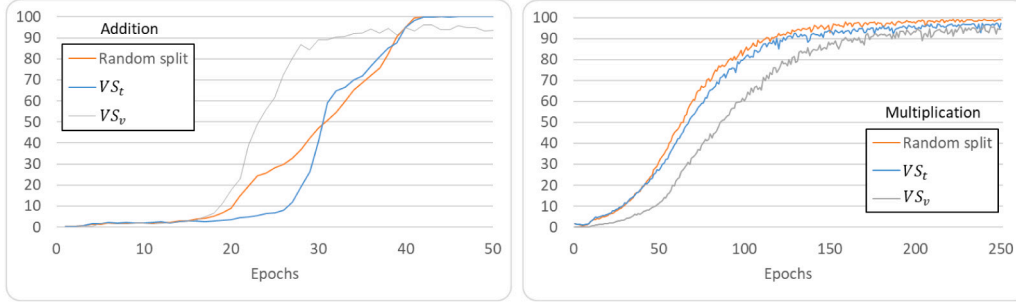
Fig. 3. Sequence accuracy on Random, VS_t, and VS_v validation subsets for addition (left) and multiplication (right). Results are averaged over five runs. VS_t reaches 100% accuracy on additions (the same of Random split) and 97.5% accuracy on multiplication (just 1.4% less than random split); VS_v reaches 93.7% on addition and 94.3% on multiplication (6.3% and 4.6% less than Random split, respectively).

learn the two tasks. In fact, random split leads to a complete (even if sparse) coverage of the input space by both the training and validation sets, where each example in the validation set has high chance to be close to a training set example, and interpolation is enough to fill the gaps.

Hereafter, we exploit the well-known difficulty of a numerical regressor to work in the extrapolation regime to get insights about the computational approach of the LM. In particular, we considered two different criteria to isolate specific portion of the input space for the validation set, in order to better investigate extrapolation capabilities:

- $VS_t = \{(A, B)|(A, B) \in NN_{4096}((A^*, B^*))\}$
  where $NN_{4096}((A^*, B^*))$ is the set of 4096 pairs (A, B) which are the nearest neighbors to a centroid (A^*, B^*) according to the Hamming distance between the corresponding token representations (i.e., number of different tokens at corresponding positions). As centroid (A^*, B^*) in the token space we used: 1010101 $\langle op \rangle$ 0101010.
- $VS_v = \{(A, B)|32 \leq A < 96 \quad \text{and} \quad 32 \leq B < 96\}$
  here the centroid is located in the middle of the value space (64, 64), so VS_v is a squared region (of side 64) centered in the value space.

Both VS_t and VS_v isolate a contiguous data region of 4096 samples to be included in the validation set, but in the former the samples are close in the token representation space, while in latter are close in the value space. Being such contiguous portions of space excluded from the training set, we can expect a worse generalization. From the results (see Fig. 3) we note that VS_t is very marginally affecting LM training and generalization while VS_v has a major impact: in fact, in the second case, for both addition and multiplication the final sequence accuracy is from 4% to 6% points lower. This result strengthens the ERD hypothesis, since: (i) using VS_v leads to the exclusion of a specific contiguous portion of value space during phase 2 and does not allow to properly train the regressor in this region; (ii) the encoding performed during phase 1 makes irrelevant the selection performed according to VS_t because, after encoding, the corresponding data point remains scattered in the value space and the regressor can easily interpolate among them. Similar results were obtained with nanoGPT (see Fig. E.8 in Appendix E.)

4.5. Looking at internal representations

Understanding the internal representation (embeddings in the vector space) in a trained Transformer is not an easy task. However, in the specific setting considered we can gain some hints by looking at the distances between the embedding of different data points (at different layers) and correlating them with the corresponding distances at input/output levels.

Given an LM trained on addition (or multiplication) we consider the dataset S including the 128 input pairs where the two operands have identical values[4]:

$$S = \{(A, A)|0 \leq A < 128\}$$

At the input level (*in*) we can compute two ordered sets of 8128 ($128 \times 127/2$) distances each:

$$d_{in,t} = \{hdist(X, Y)|(X, X), (Y, Y) \in S, X < Y\}$$
$$d_{in,v} = \{|X - Y| \quad |(X, X), (Y, Y) \in S, X < Y\}$$

where $hdist(X, Y)$ is the Hamming distance between the token representation of X and Y, and the subscript letters t and v denote token and value levels, respectively.

At the output level (*out*) we can compute the two corresponding sets of distances as:

$$d_{out,t} = \{hdist(P, Q)|(X, X), (Y, Y) \in S, X < Y\}$$
$$d_{out,v} = \{|P - Q| \quad |(X, X), (Y, Y) \in S, X < Y\}$$

where ($P = X + X$ and $Q = Y + Y$) for addition, and ($P = X \times X$ and $Q = Y \times Y$) for multiplication.

Finally, for each intermediate level of the Transformer encoder (*enc*) or decoder (*dec*) we can compute the Euclidean distances among the corresponding embedding vectors.

$$d_{enc_i} = \{\|enc_i(X, X) - enc_i(Y, Y)\| \quad |(X, X), (Y, Y) \in S, X < Y\}$$
$$d_{dec_i} = \{\|dec_i(X, X) - dec_i(Y, Y)\| \quad |(X, X), (Y, Y) \in S, X < Y\}$$

where enc_i and dec_i are the output vectors obtained by concatenating all the token embeddings (each of dimensionality 64) after the ith encoder and decoder layer, respectively. For example enc_i has dimensionality $960 = 64 \times 15$ where 15 is the number of tokens in the encoder.

Even if the distances in the different sets have different ranges, we can use correlation to find out similarities. If two sets of distances are correlated we can expect that the corresponding representations/embeddings are correlated as well. Since both Pearson and Spearman correlations (Schober et al., 2018) provided similar outputs, for simplicity in Fig. 4 we report only Pearson correlations.

The yellow cells in the tables of Fig. 4 confirm the low correlation between the token and value representation at both input and output level. The blue cells show that correlation remains quite similar across the encoder layers as if the encoder was not performing any significant computation (this is confirmed in Section 5 where we achieve similar results by totally removing all intermediate attention and MLP layers in the encoder). More interesting is the trend of correlations across the decoder layers (green cells). In particular, for the addition the token

---

[4] Since the input prompt contains two operands, we select only the cases with identical values ($A = B$) in order to easily determine the "magnitude" of the input, and thereafter compute meaningful distances.

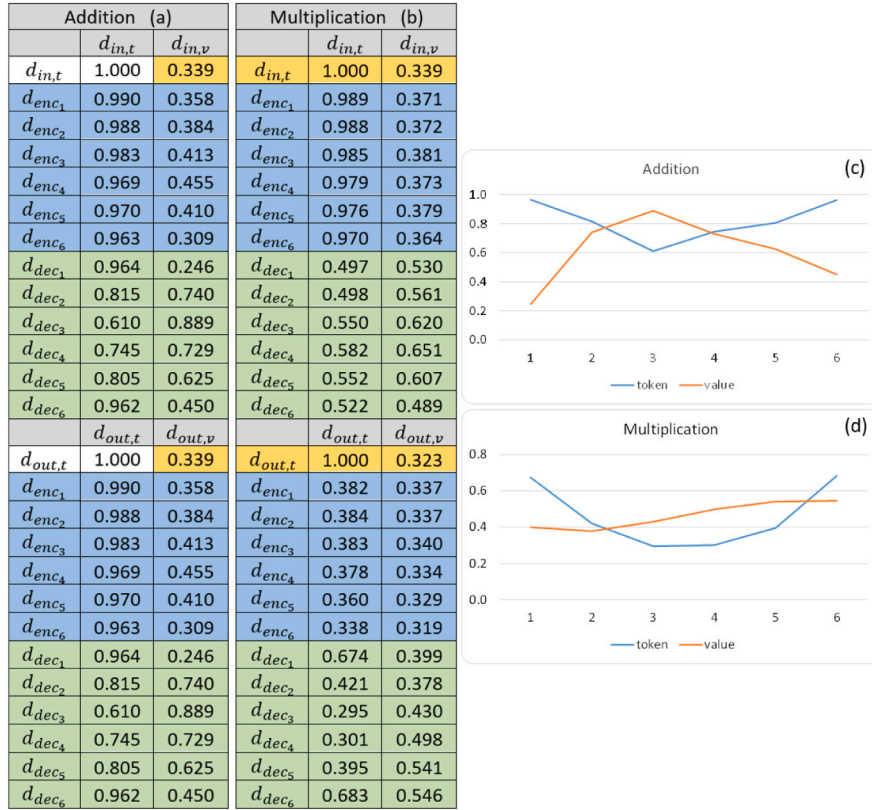| Addition (a)			Multiplication (b)		
	$d_{in,t}$	$d_{in,v}$		$d_{in,t}$	$d_{in,v}$
$d_{in,t}$	1.000	0.339	$d_{in,t}$	1.000	0.339
d_{enc_1}	0.990	0.358	d_{enc_1}	0.989	0.371
d_{enc_2}	0.988	0.384	d_{enc_2}	0.988	0.372
d_{enc_3}	0.983	0.413	d_{enc_3}	0.985	0.381
d_{enc_4}	0.969	0.455	d_{enc_4}	0.979	0.373
d_{enc_5}	0.970	0.410	d_{enc_5}	0.976	0.379
d_{enc_6}	0.963	0.309	d_{enc_6}	0.970	0.364
d_{dec_1}	0.964	0.246	d_{dec_1}	0.497	0.530
d_{dec_2}	0.815	0.740	d_{dec_2}	0.498	0.561
d_{dec_3}	0.610	0.889	d_{dec_3}	0.550	0.620
d_{dec_4}	0.745	0.729	d_{dec_4}	0.582	0.651
d_{dec_5}	0.805	0.625	d_{dec_5}	0.552	0.607
d_{dec_6}	0.962	0.450	d_{dec_6}	0.522	0.489
	$d_{out,t}$	$d_{out,v}$		$d_{out,t}$	$d_{out,v}$
$d_{out,t}$	1.000	0.339	$d_{out,t}$	1.000	0.323
d_{enc_1}	0.990	0.358	d_{enc_1}	0.382	0.337
d_{enc_2}	0.988	0.384	d_{enc_2}	0.384	0.337
d_{enc_3}	0.983	0.413	d_{enc_3}	0.383	0.340
d_{enc_4}	0.969	0.455	d_{enc_4}	0.378	0.334
d_{enc_5}	0.970	0.410	d_{enc_5}	0.360	0.329
d_{enc_6}	0.963	0.309	d_{enc_6}	0.338	0.319
d_{dec_1}	0.964	0.246	d_{dec_1}	0.674	0.399
d_{dec_2}	0.815	0.740	d_{dec_2}	0.421	0.378
d_{dec_3}	0.610	0.889	d_{dec_3}	0.295	0.430
d_{dec_4}	0.745	0.729	d_{dec_4}	0.301	0.498
d_{dec_5}	0.805	0.625	d_{dec_5}	0.395	0.541
d_{dec_6}	0.962	0.450	d_{dec_6}	0.683	0.546

Fig. 4. Pearson correlation between ordered sets of distances for addition (a) and multiplication (b). Each cell denotes the correlation between the two ordered set of distances specified in the corresponding row and column. Note that since for addition in this experiment the output value is always twice the input, the correlation values (blue and green cells) are the same for $d_{in,\_}$ and $d_{out,\_}$ block of values. Graphs (c) and (d) show the correlations of output distances $d_{out,t}$ (at token level — blue curves) and $d_{out,v}$ (at value level — orange curves) with the embedding distances d_{dec_i} across the 6 decoder layers for addition and multiplication, respectively.

representation has high correlation with the first and last layers and low with central layers, while the value representation has an opposite trend (see also Fig. 4.c). These results support the ERD hypothesis and in particular that the initial and final layers in the decoder transform from token to value representation (and vice versa) while the central layers perform regression in the value space. In particular, at layer 3, the correlation at token level is minimum while the correlation at value level is maximum.

For multiplication the low-high-low trend at value level is less evident (Fig. 4.d orange curve), probably because the quadratic dependence of the output from the input (at value level) does not allow to learn a simple regressor smoothly working in the whole vector space, and the mapping is performed by piecewise linear approximation in different space regions, which introduces discontinuities that make global distances in the vector space unsuitable to quantify the representation similarity.

As discussed in Section 2.2, correlation analyses might be insufficient to prove that the presence of a certain information in the embeddings is really necessary to compute the output (direct causation). So to further strengthen our hypothesis we applied an amnesic probing technique (Elazar et al., 2021) and proved that, upon removal of value information from the embeddings, the LM is no longer capable of performing the right computation. Details are reported in Appendix D.

5. Ablation study

This section presents the results of an ablation study where the LM architecture was simplified, to understand which components are necessary to learn the addition/multiplication computation. Squeezing the encoder (i.e., removing all intermediate attention and MLP layers)

Table 3
Epochs necessary to reach 95% accuracy on the validation set. A dash is used when 95% accuracy is not achieved in 1K epochs: in such case the accuracy reached is reported within brackets.

Configuration	Addition	Multiplication
Full (see Table 2)	39	137
Squeezing the encoder (see main text)	60	426
num_heads h=1	25	225
Reduced dimensionality ($d_{model} = 32$)	66	309
No positional embedding	— (2.4%)	— (1.8%)
No attention layers	— (0.9%)	— (1.7%)
No fully connected layers	56	398

does not have a relevant impact; this is consistent with other works claiming that a decoder only architecture (Liu et al., 2018) can achieve similar results with respect to an encoder–decoder Transformer, and further confirmed by the nanoGPT results presented in Appendix E. A simplification of the architecture in terms of (i) reduction of dimensionality; (ii) reduction of number of heads; (iii) removal of fully connected layers is well tolerated, while positional embedding and attention layers are mandatory for the LM in order to properly perform token to value transformation (and vice versa). Table 3 summarizes the results.

6. Discussion and conclusions

In this paper we introduced a simplified setup to allow a light LM to learn binary addition and multiplication. Both the LM architectures considered easily learn the two tasks and generalizes well on unseen data, proving that memorization of the training data is neither necessary nor efficient. The experiments on the interpolation/extrapolation

capabilities and correlation of input–output representations with internal embedding suggest that the model solves the computational task as a supervised regression problem in the value space after an initial encoding from token to values, and a final decoding from output value to tokens. Under this hypothesis: (i) any task that can be solved by a neural network regressor can be solved by an LM as well, with the extra burden of end-to-end learning decoding/encoding steps; (ii) when looking at interpolation/extrapolation capabilities of an LM applied to a mathematical task, we should not concentrate on the input token representation but on the internal representation after encoding, keeping in mind the difficulties of a numerical regressor to work on region spaces not covered by the training set; (iii) on a more speculative side, we could guess that modern LLMs learn the number encoding/decoding once and reuse it across different numerical tasks whereas a specific regressor is learned for each task.

Our ERD hypothesis could be questioned considering some recent findings from Lee et al. (2023) where providing in the prompt intermediate information (scratchpad) about the decomposition of arithmetic tasks improves the training efficiency and requires fewer examples. This could suggest that a symbolic manipulation approach is adopted to learn imitating step by step the proposed decomposition. However, in most of the cases their model was able to learn the same task (even if slowly) without scratchpad and/or with wrong scratchpads. As argued by the authors the higher efficiency is actually in terms of examples and not in terms of tokens since each scratchpad requires a large number of extra tokens, and we guess these could be used as extra features by the underlying regressor. Furthermore, scratchpad contribution is negligible for more complex operations such as sine and square root, but, unexpectedly, learning such complex operations was simpler than multiplication. This is not strange under the ERD hypothesis where a unary smooth operator like the sine can be learned by a supervised regressor independently of the mathematical method used for its computation.

The algorithmic interpretation that (Nanda et al., 2023) provided for modular addition, could also suggest that their LM discovered and efficient symbolic manipulation approach; however, as discussed in Section 2.3, it is more likely that a regressor was learned to numerically approximate an efficient sparse Fourier decomposition, under regularization constraints favoring sparsity. Finally, the information flow described in Stolfo et al. (2023), points out that MLPs in the last layers are responsible for the numerical computation of the solution, which is compatible with the hypothesis of a multi-layer regressor.

Of course we are not claiming that all the capabilities of modern LLMs can be explained by regression, but regression is likely to be one of the internal tools that LLMs uses to predict the next token when numbers come into play.

As to future research we plan to: (i) further investigate the generalization capabilities of LMs in arithmetic tasks with respect to the composition of the training and test sets (Feng et al., 2023; Keskar et al., 2017), (ii) design simplified experiments/setups for tasks that cannot be easily mapped to regression problems such as chain of reasoning and logic deductions.

CRediT authorship contribution statement

Davide Maltoni: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Matteo Ferrara:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization.

```
   1100 (this is 12 in decimal)
 × 1001 (this is 9 in decimal)
 ------
   1100 (this is 1100×1)
  0000 (this is 1100×0, shifted left one position)
 0000  (this is 1100×0, shifted left two positions)
+ 1100 (this is 1100×1, shifted left three positions)
---------
 1101100 (this is 108 in binary)
```

Fig. B.5. Example of 4-digit binary multiplication. The sum can be performed incrementally with a two-operand adder.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data are available on Github.

Appendix A. Addition input–output discontinuities

Given an input/output pair we consider the (2^{14}) variants obtained by perturbing (i.e., 0–1 swap) the input bits and counting the resulting changes in the output. These values, averaged over all possible input/output pairs (again 2^{14}) and normalized by row are inserted in the cells of Table A.4. So, for example the value in cell (row=2, column=3) means that in the 27.9% of the cases a perturbation of 2 (over 14) bits in the input leads to a change of 3 (over 8) bits in the output.

Input–output discontinuities, which are further amplified in case of multiplications, make it very unlikely to solve these tasks by smooth interpolation of the input representation.

Appendix B. Binary addition and multiplication

Binary addition can be executed by summing pairs of corresponding bits a_i and b_i, starting from the LSBs (a_0 and b_0) and propagating carries. Let c_{i-1} be the pending carry used to sum current bits,[5] then a two-output 3-bit truth table (Table B.5) can be used to generate the output bit o_i and carry c_i used when summing the next pair of bits:

A simple approach to execute binary multiplication is through iterative binary sums. Each bit b_i of the second operand is multiplied by the whole first operand, but this inner multiplication is straightforward since it results either in a sequence of 0 (if $b_i = 0$) or a copy of the first operand (if $b_i = 1$). This intermediate result is then shifted left and summed to the current output. An example is reported in Fig. B.5 below.

Appendix C. Learning a regressor under predict-next-token training

In Section 4.3 we argued that an arithmetic computation task can be decomposed into three steps whose central one is learning a regressor in the value space: $\mathbf{v}_R = regress(\mathbf{v}_A, \mathbf{v}_B)$. If we consider the autoregressive working mode of a Transformer and its predict-next-token training, the regressor must be able to work incrementally given the output produced so far. In particular, we can formulate the problem as: $\mathbf{v}_{r_i} = regress(\mathbf{v}_A, \mathbf{v}_B, i, \mathbf{c}_{R_{i-1}})$ where:

---

[5] When summing the LSBs ($i = 0$), there is no pending carry, so $c_{-1} = 0$.

Table A.4
Addition input–output discontinuities.

	0	1	2	3	4	5	6	7	8
0	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
1	0.0%	50.0%	28.6%	12.5%	5.4%	2.2%	0.9%	0.3%	0.1%
2	3.8%	8.8%	30.6%	27.9%	16.4%	7.9%	3.3%	1.1%	0.2%
3	0.8%	8.0%	16.5%	27.0%	24.3%	14.6%	6.5%	2.0%	0.3%
4	0.8%	4.3%	14.2%	22.7%	25.9%	19.4%	9.4%	2.9%	0.4%
5	0.5%	3.8%	11.3%	21.9%	26.1%	21.2%	11.2%	3.5%	0.5%
6	0.4%	3.2%	11.0%	20.9%	26.5%	22.0%	11.8%	3.7%	0.5%
7	0.4%	3.3%	10.7%	21.2%	26.4%	22.1%	11.8%	3.6%	0.5%
8	0.5%	3.4%	11.1%	21.2%	26.5%	21.8%	11.5%	3.6%	0.5%
9	0.5%	3.6%	11.3%	21.3%	26.0%	21.4%	11.7%	3.6%	0.5%
10	0.6%	3.8%	11.4%	20.6%	25.2%	21.5%	12.0%	4.3%	0.6%
11	0.6%	3.7%	10.5%	19.1%	23.8%	21.7%	14.4%	5.1%	1.1%
12	0.6%	3.3%	8.5%	15.2%	22.1%	23.4%	15.9%	9.9%	1.1%
13	0.7%	2.6%	4.8%	9.4%	16.5%	25.0%	26.8%	7.1%	7.1%
14	0.8%	0.8%	1.6%	3.1%	6.3%	12.5%	25.0%	50.0%	0.0%

Table B.5
Two-output 3-bit truth table for binary addition.

Inputs			Outputs	
a_i	b_i	c_{i-1}	o_i	c_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- $\mathbf{v}_A = [\mathbf{v}_{a_0}\mathbf{v}_{a_1}...\mathbf{v}_{a_6}]$ and $\mathbf{v}_B = [\mathbf{v}_{b_0}\mathbf{v}_{b_1}...\mathbf{v}_{b_6}]$ are the value vectors of the two input operands, obtained as the concatenation of the value vectors of single tokens. Both are always fully available to the decoder. Note that, $\mathbf{v}_{a_i}$ and $\mathbf{v}_{b_i}$ are not the bits of the inputs, but correspond to their value vectors including also positional information.
- i is the position of the token to be predicted (we can assume it is available through positional encoding).
- $\mathbf{c}_{R_{i-1}} = [\mathbf{c}_{r_0}\mathbf{c}_{r_1}...\mathbf{c}_{r_{i-1}}]$ is a value vector encoding the current context determined by the result produced so far (entering in the decoder from the bottom).
- $\mathbf{v}_{r_i}$ is the value vector of the ith token.

In principle, the regressor could predict each $\mathbf{v}_{r_i}$ based on $\mathbf{v}_A$ and $\mathbf{v}_B$ alone, but we argue that the exploitation of the result produced so far $\mathbf{c}_{R_{i-1}}$ can lead to higher training efficiency. To this purpose is interesting to evaluate the impact of the output ordering (plain or reverse). In both the addition and multiplication the ith token of the result only depends on the tokens of the inputs at positions $\leq i$ (see Appendix B). Therefore, if reverse order is adopted, as we assumed until now, $\mathbf{v}_{A_i} = [\mathbf{v}_{a_0}\mathbf{v}_{a_1}...\mathbf{v}_{a_i}]$, $\mathbf{v}_{B_i} = [\mathbf{v}_{b_0}\mathbf{v}_{b_1}...\mathbf{v}_{b_i}]$ and $\mathbf{c}_{R_{i-1}}$ are sufficient to predict $\mathbf{v}_{r_i}$. Vice versa, if the output computation starts with the MSBs the regressor cannot leverage the above iterative decomposition and needs to learn the task as a global operation using whole vectors $\mathbf{v}_A$ and $\mathbf{v}_B$ with almost no support from the result produced so far.

In Fig. C.6 we note that with plain order both addition and multiplication require a much longer number of epochs to converge and the learning curve is less stable. Further experiments proved that, as expected, the order of the inputs (also reverse by default in this study) is irrelevant, since the LM can always access the whole input representations $\mathbf{v}_A$ and $\mathbf{v}_B$. The advantages of using the reverse order are pointed out in other recent studies (Lee et al., 2023; Nogueira

et al., 2021). In particular, Lee et al. (2023) reported a significant improvement with respect to plain order.

Appendix D. Amnesic probing results

The outcome of correlation analyses performed in Section 4.3 suggests that the embeddings in the central layers of the decoder contain information related to the value representation of the output (see Fig. 4). However correlation does not mean causation, and here we investigate deeper. Amnesic probing was proposed in Elazar et al. (2021) building on the approach (Ravfogel et al., 2020) to check to what extent a model output is affected by the removal of specific features or attributes in an intermediate level embeddings. Here we focus on addition and we try to remove some features from the decoder layer 3 embeddings ($dec_3(X,Y)$). To this purpose a linear probe (a linear regressor in our case) was trained to predict the output value ($X+Y$) starting from the $dec_3(X,Y)$ embeddings and its nullspace is used to project the embeddings in a new space lacking output value information. According to Ravfogel et al. (2020), due to the simplicity of the linear regressor used, the procedure is repeated twice to remove more information. Our results show that:

1. A simple linear regressor trained on $dec_3(X,Y)$ embeddings can reach high accuracy in predicting $X+Y$ (rmse = 0.28).
2. If the projected embeddings are overwritten in the LM decoder at level 3, and a partial forward pass is performed thereafter, the addition sequence accuracy severely drops from 100% to 0.13%.
3. As indicated in Elazar et al. (2021) since any information removal could hamper the model accuracy, a control test was performed by removing the same amount of information (but on random directions instead of the nullspace directions) and in this case the LM final sequence accuracy remained 100%.

This experiment provides further support to the hypothesis that the value information is not only present in the inspected embeddings but is also crucial for the output computation.

On the computational side, we argue that amnesic probing complexity is low because it relies on simple steps as linear regression and null space computation, with the former being the most demanding step. Linear regression complexity is $O(nd^2 + d^3)$ where n is the number of training examples and d the dimensionality of the embeddings.

Appendix E. NanoGPT - a decoder-only LM

To demonstrate that our findings generalize beyond the encoder–decoder architecture of the original Transformer used in this work, the
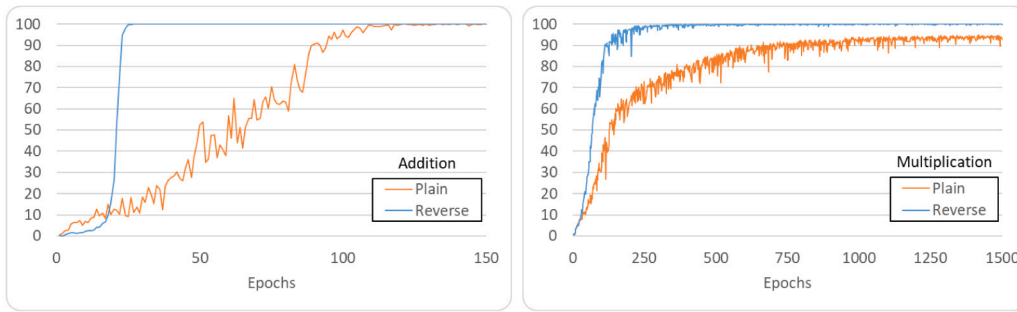
Fig. C.6. Sequence accuracy on validation set for reverse (default in this work) and plain order of the input and output representations. From left to right: addition and multiplication.
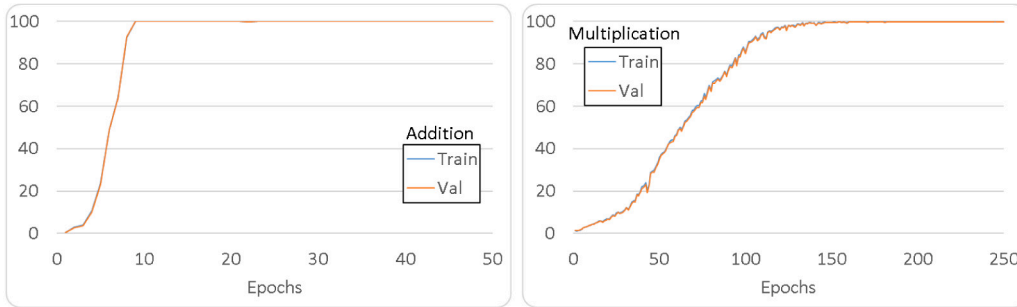

Fig. E.7. Sequence accuracy of the nanoGPT model (refer to Section 4.1 for more details). From the left: addition and multiplication. Results are averaged over five runs. Note that, training and validation curves are almost overlapped.
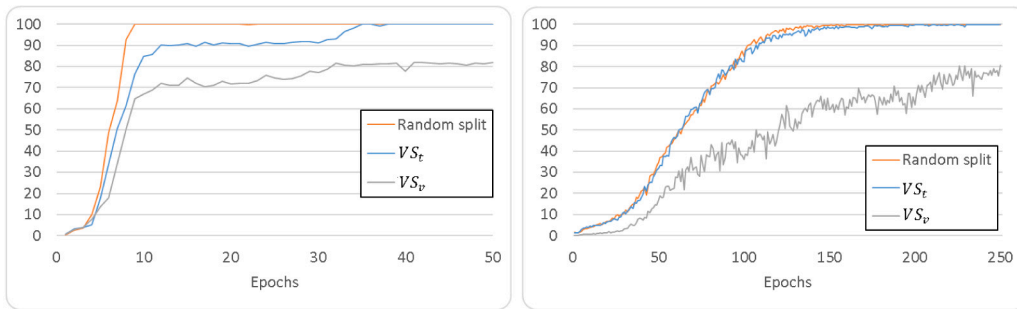

Fig. E.8. Sequence accuracy of the nanoGPT model on Random, VS_t, and VS_v validation subsets for addition (left) and multiplication (right). Results are averaged over five runs.

Table E.6
Details of the nanoGPT model..

token embedding	learned
positional encoding	learned
d_{model}	64
d_{ff}	$d_{model} \times 4$
num_heads h	8
decoder layers	6
dropout	0.1
learnable parameters	298K

main experiments have been repeated using a second LM, that is the nanoGPT (Karpathy, 2022) decoder-only model. Table E.6 reports the details of the nanoGPT model adopted.

Fig. E.7 shows that the nanoGPT model was able to learn addition and multiplication still more efficiently than the original Transformer (compare Fig. 1 with Fig. E.7). For the training, we used a minibatch size of 128, a standard CrossEntropy loss, the AdamW optimizer with a learning rate of 0.001 and betas = 0.9 and 0.98, and a gradient clipping to 1.0.

Fig. E.8 shows the sequence accuracy of the nanoGPT model on Random, VS_t, and VS_v validation subsets for addition and multiplication

(see Section 4.4 for more details). Using the VS_t subset, it reaches 100% and 99.9% accuracy on addition and multiplication, respectively (the same of Random split) while, using the VS_v subset, it reaches 82.0% on addition and 80.6% on multiplication (18.0% and 19.4% less than Random split, respectively). Results are inline with those obtained in Section 4.4 but here the difference between VS_t, and VS_v is still more significant.

References

Belinkov, Y. (2022). Probing classifiers: Promises, shortcomings, and advances. *Computational Linguistics, 48*, 207–219.

Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y., Li, Y., Lundberg, S., Nori, H., Palangi, H., Ribeiro, M., & Zhang, Y. (2023). Sparks of artificial general intelligence: Early experiments with GPT-4. arXiv:2303.12712.

Elazar, Y., Ravfogel, S., Jacovi, A., & Goldberg, Y. (2021). Amnesic probing: Behavioral explanation with amnesic counterfactuals. *Transactions of the Association for Computational Linguisticss, 9*, 160–175.

Elhage, N., Nanda, N., Olsson, C., Henighan, T., Joseph, N., Mann, B., Askell, A., Bai, Y., Chen, A., Conerly, T., DasSarma, N., Drain, D., Ganguli, D., Hatfield-Dodds, Z., Hernandez, D., Jones, A., Kernion, J., Lovitt, L., Ndousse, K., ... Olah, C. (2021). A mathematical framework for transformer circuits. Transformer Circuits Thread - https://transformer-circuits.pub/2021/framework/index.html.

Feng, Y., Zhang, W., & Tu, Y. (2023). Activity–weight duality in feed-forward neural networks reveals two co-determinants for generalization. *Nature Machine Intelligence, 5*, 908–918.

Geiger, A., Lu, H., Icard, T. F., & Potts, C. (2021). Causal abstractions of neural networks. In *35th conference on neural information processing systems*.

Hewitt, J., & Manning, C. (2019). A structural probe for finding syntax in word representations. In *Conference of the North American chapter of the association for computational linguistics*.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). Scaling laws for neural language models. arXiv:2001.08361.

Karpathy, A. (2022). nanoGPT: A lightweight implementation of medium-sized GPTs. https://github.com/karpathy/nanoGPT.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P. (2017). On large-batch training for deep learning: Generalization gap and sharp minima. In *5th international conference on learning representations*.

Lasri, K., Pimentel, T., Lenci, A., Poibeau, T., & Cotterell, R. (2022). Probing for the usage of grammatical number. In *60th annual meeting of the association for computational linguistics*.

Lee, N., Sreenivasan, K., Lee, J., Lee, K., & Papailiopoulos, D. (2023). Teaching arithmetic to small transformers. arXiv:2307.03381.

Liu, T., & Low, B. K. H. (2023). Goat: Fine-tuned LLaMA outperforms GPT-4 on arithmetic tasks. arXiv:2305.14201.

Liu, P., Saleh, M., Pot, E., Goodrich, B., Sepassi, R., Kaiser, L., & Shazeer, N. (2018). Generating Wikipedia by summarizing long sequences. In *6th international conference on learning representations*.

Muffo, M., Cocco, A., & Bertino, E. (2023). Evaluating transformer language models on arithmetic operations using number decomposition. In *13th conference on language resources and evaluation* (pp. 291–297).

Naik, A., Ravichander, A., Rose, C., & Hovy, E. (2019). Exploring numeracy in word embeddings. In *57th annual meeting of the association for computational linguistics*.

Nanda, N., Chan, L., Lieberum, T., Smith, J., & Steinhardt, J. (2023). Progress measures for grokking via mechanistic interpretability. arXiv:2301.05217.

Nogueira, R., Jiang, Z., & Lin, J. (2021). Investigating the limitations of transformers with simple arithmetic tasks. In *1st mathematical reasoning in general artificial intelligence workshop*.

Räuker, T., Ho, A., Casper, S., & Hadfield-Menell, D. (2023). Toward transparent AI: A survey on interpreting the inner structures of deep neural networks. arXiv:2207.13243.

Ravfogel, S., Elazar, Y., Gonen, H., Twiton, M., & Goldberg, Y. (2020). Null it out: Guarding protected attributes by iterative nullspace projection. In *58th annual meeting of the association for computational linguistics* (pp. 7237–7256).

Schober, P., Boer, C., & Schwarte, L. (2018). Correlation coefficients: Appropriate use and interpretation. *Anesthesia and Analgesia, 126*, 1763–1768.

Stolfo, A., Belinkov, Y., & Sachan, M. (2023). A mechanistic interpretation of arithmetic reasoning in language models using causal mediation analysis. arXiv:2305.15054.

Sundararaman, D., Si, S., Subramanian, V., Wang, G., Hazarika, D., & Carin, L. (2020). Methods for numeracy-preserving word embeddings. In *Conference on empirical methods in natural language processing*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*: *vol. 30*.

Wallace, E., Wang, Y., Li, S., Singh, S., & Gardner, M. (2019). Do NLP models know numbers? Probing numeracy in embeddings. arXiv:1909.07940.

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., Chi, E., Hashimoto, T., Vinyals, O., Liang, P., Dean, J., & Fedus, W. (2022). Emergent abilities of large language models. *Transactions on Machine Learning Research (TMLR)*.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. arXiv:2201.11903.

White, J., Pimentel, T., Saphra, N., & Cotterell, R. (2021). A non-linear structural probe. In *Conference of the North American chapter of the association for computational linguistics*.

Yuan, Z., Yuan, H., Tan, C., Wang, W., & Huang, S. (2023). How well do large language models perform in arithmetic tasks? arXiv:2304.02015.