INVESTIGATING THE LIMITATIONS OF TRANSFORMERS WITH SIMPLE ARITHMETIC TASKS

Rodrigo Nogueira, Zhiying Jiang & Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo

ABSTRACT

The ability to perform arithmetic tasks is a remarkable trait of human intelligence and might form a critical component of more complex reasoning tasks. In this work, we investigate if the surface form of a number has any influence on how sequence-to-sequence language models learn simple arithmetic tasks such as addition and subtraction across a wide range of values. We find that how a number is represented in its surface form has a strong influence on the model's accuracy. In particular, the model fails to learn addition of five-digit numbers when using subwords (e.g., "32"), and it struggles to learn with character-level representations (e.g., "3 2"). By introducing position tokens (e.g., "3 10e1 2"), the model learns to accurately add and subtract numbers up to 60 digits. We conclude that modern pretrained language models can easily learn arithmetic from very few examples, as long as we use the proper surface representation. This result bolsters evidence that subword tokenizers and positional encodings are components in current transformer designs that might need improvement. Moreover, we show that regardless of the number of parameters and training examples, models cannot seem to learn addition rules that are independent of the length of the numbers seen during training. Code to reproduce our experiments is available at `https://github.com/castorini/transformers-arithmetic`

1 INTRODUCTION

Abstraction and composition are two important themes in the study of human languages, made possible by different linguistic representations. Although treatments in different linguistic traditions vary, representations at the lexical, syntactic, and semantic levels are a common feature in nearly all theoretical studies of human language, and until relatively recently, these representations are explicitly "materialized" in language processing pipelines (for example, semantic role labeling takes as input a syntactic parse).

However, with the advent of pretrained transformer models, these intermediate representations no longer have any explicit "reality": while various studies have found evidence of syntactic and semantic knowledge in these models (Tenney et al., 2019), it is no longer possible to isolate, for example, a subject–verb relation in a specific part of the model. With transformers, the *only* input to the model is the surface form of text combined with supplemental embeddings (e.g., positional embeddings, and in the case of BERT, segment embeddings).

What are the consequences of this exclusive focus on the surface form of text? Some might say, nothing, as bigger models, better pretraining objectives, etc. will lead us to models that are capable of reasoning (Brown et al., 2020). We believe this to be an untenable position and present a case study in simple arithmetic tasks where having the right representation is the difference between a nearly-impossible-to-learn task and an easy-to-learn task. Our work shows that it is possible to "inject" representations into transformer models by simple manipulations of the input sequence (in our case, explicitly enumerating the semantics of digit positions), and that doing so makes it possible for off-the-shelf models to easily perform simple arithmetic, whereas it is nearly impossible otherwise.

While we present only a case study, our findings have broader implications for various language analysis tasks: First, although end-to-end training enabled by neural networks is a powerful tool,

having the right representation is crucial also. Second, we demonstrate a simple way in which representations can be "injected" into transformer models in a completely transparent manner, without any need to re-pretrain. This work points out a path that might allow us to combine the best of both worlds: leveraging the power of pretraining, with additional guidance from our understanding of the problem domain.

However, we find that even explicit semantic representations have their limits. Despite our best efforts, we find that models cannot extrapolate, i.e., they fail to perform simple arithmetic when evaluated on inputs whose length distribution differs from the one seen during training. This appears to be a problem that neither larger models, more compute, nor more data can solve.

There are, of course, many previous papers that investigate the representation of numbers and various numeric reasoning tasks in the literature. We present related work in Appendix A.

2 METHODOLOGY

Our tasks are the addition and subtraction of two numbers. We cast them as sequence-to-sequence tasks in which both inputs to the models and target outputs are treated as sequences of tokens. For the addition task, an example input is "What is 52 plus 148?" and the target output is "200". For the subtraction task, an example input is "What is 20 minus 185?" and the target output is "-165".

We programmatically generate training, development, and test sets of different sizes depending on the experiment. The input template is always "What is [number1] [operation] [number2]?", where [number1] and [number2] are numbers randomly sampled and [operation] is either "plus" or "minus". Below, we discuss different ways of representing [number1] and [number2] and their corresponding answer. We use two different methods to sample numbers for training, development, and test sets, which are described below.

Balanced sampling: To generate training and development sets, we first set the maximum number of digits D and then create each example as follows: We first sample d from $[2, D]$ and then independently sample [number1] and [number2] from $[10^{d-1}, 10^d - 1]$. We then compute the answer according to the operation (i.e., either addition or subtraction). This method ensures that the set will have a roughly equal proportion of d-digit numbers, where $d \in [2, D]$.

Random sampling: To generate test sets, we sample [number1] and [number2] independently from $[0, 10^D - 1]$. This results in approximately 90% of the numbers having D-digits, 9% having $(D-1)$-digits, and so on. This unbalanced set aims at evaluating models on the largest numbers it was trained on. We study how different sampling methods influence model effectiveness in Appendix G.

Metric: Our metric is accuracy. That is, the model receives a score of one if its output matches the target output exactly. Otherwise, it receives a score of zero.

Our experiments use T5 (Raffel et al., 2020), a pretrained sequence-to-sequence model where every natural language processing task—for example, machine translation, question answering, and classification—is formulated as feeding the model some input sequence and training it to generate some output sequence. We follow this same approach and feed the addition or subtraction question (described above) as a sequence of tokens to the model and train it to generate the answer, token by token. We use greedy decoding as beam search showed similar effectiveness but is slower.

We train the models using the AdamW optimizer (Loshchilov & Hutter, 2018), batches of 128 examples, and a learning rate of 0.0003. We experimented with all T5 model sizes except for T5-11B due to its computational cost. We refer to T5-small, T5-base, and T5-large as T5-60M, T5-220M, and T5-770M, respectively, to easily distinguish models by their numbers of parameters. We also experiment with "vanilla" (i.e., non-pretrained) transformers (see Appendix B).

Previous studies have recognized that commonly used subword tokenization techniques today are not ideal to represent numbers (Wallace et al., 2019; Henighan et al., 2020; Saxton et al., 2018; Lample & Charton, 2019), although none of them studied the problem in depth. Here, we investigate how six different number representations, illustrated in Table 1, impact model accuracy on the arithmetic tasks. In our main results, we only experiment with the "standard" ordering of generating digits (i.e., most to least significant), but in Appendix C, we also experimented with inverting the order.

Orthography	Example	Notes
DECIMAL	832	default representation
CHARACTER	8 3 2	ensures consistent tokenization
FIXED-CHARACTER	0 8 3 2	ensures consistent positions (e.g., max. 4 digits)
UNDERSCORE	8_3_2	underscores provide hints on digit significance
WORDS	eight hundred thirty-two	leverages pretraining
10-BASED	8 100 3 10 2	easy to determine digit significance
10E-BASED	8 10e2 3 10e1 2 10e0	more compact encoding of above

Table 1: Different ways of representing numbers explored in this work.

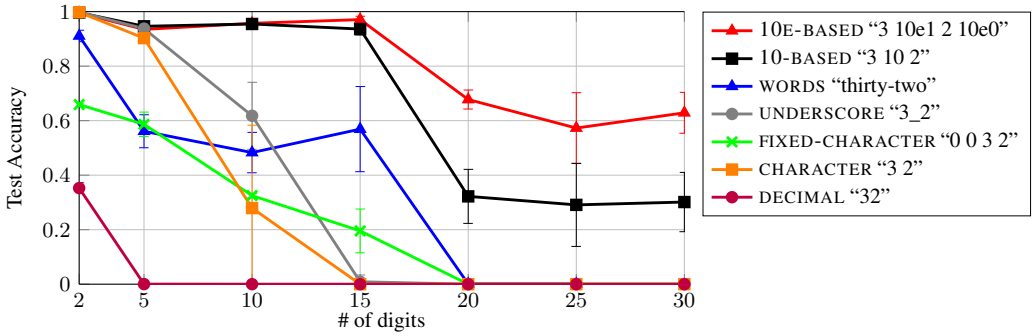

Figure 1: Accuracy of different number representations on the addition task.

DECIMAL: Digits are represented in the Hindu–Arabic numeral form (also called decimal form).

CHARACTER: Digits are separated by a white space, thus allowing the model to work on embeddings that always represent single digits.

FIXED-CHARACTER: In the character representation above, it is hard to determine the significance of a digit by relative position embeddings because relative positions change on a per example basis. To address this, we introduce the FIXED-CHARACTER representation in which numbers have the same maximum number of digits.

UNDERSCORE: Digits are separated by an underscore token. A possible advantage of this representation is that the model can learn to find the significance of a digit by counting the number of underscores to the right until the least significant digit.

WORDS: Numbers are converted to words using the *num2words* package.[1] We can anticipate two advantages in this representation: (1) the T5 model was pretrained on large amounts of textual data, so it likely knows that "hundred" is larger than "ten" (Zhang et al., 2020); (2) digits are surrounded by tokens that describe their significance ("hundred", "thousand", etc.), thus making it easier to find which two digits in the input sequence should be added (or subtracted).

10-BASED: Digits are separated by powers of 10, which we call position tokens. This representation allows the model to find the significance of a digit by simply inspecting its left or right tokens.

10E-BASED: Digits are separated by powers of 10 represented using scientific notation. This orthography has a more compact representation for the position tokens of large numbers than the 10-BASED orthography. For example, in the 10-BASED orthography, the position token of the most significant digit of a 60-digit number occupies 60 characters (i.e., "1" followed by 59 zeros). In the 10E-BASED orthography, this position token occupies only 5 characters (i.e., "10e59").

3 RESULTS

We present results in Figure 1. Each point in the graph represents the mean accuracy of a T5-220M model trained for 100 epochs with five different sets of 1,000 addition examples sampled using the balanced method. A separate development set of 1,000 examples is used to select the best checkpoint of each run. Error bars correspond to 95% confidence intervals. The values on the x-axis represent the maximum number of digits used for training and testing. We use a maximum of 30-digit numbers as some representations such as WORDS would result in input sequences that have too many tokens (e.g., more than 512), and hence prohibitively long training times.

In the DECIMAL representation, the model barely learns addition of 2-digit numbers, and it fails to learn addition of larger numbers, i.e., it has an accuracy of zero for 5 digits or more. One explanation for this failure is because numbers are not systematically tokenized into digits. For instance, "132" might be tokenized as "1" and "32", whereas "232" might be tokenized as "23" and "2". Hence, the model would have to learn that sometimes the embedding of a token refers to a single digit, other times to two digits, etc. It might be hard to learn (i.e., need more examples) to map an embedding to a number when the number of digits it represents changes irregularly (dependent on the training data of the tokenizer).

The CHARACTER and UNDERSCORE representations have much higher accuracy than DECIMAL, thus showing that it is easier to learn when embeddings represent single digits. Both representations exhibit decreasing accuracy as we increase the number of digits, until reaching an accuracy of zero with 15-digit addition. One explanation for this failure is that, since digits with the same significance have different positions in each example, the model has to count the number of digits on the right side in order to find its significance. With larger numbers, counting becomes harder.

The FIXED-CHARACTER representation achieves higher accuracy than CHARACTER and UNDERSCORE for numbers longer than 12 digits, thus showing that the model can learn to memorize digit positions to determine their significance. However, with an accuracy of approximately 20% for 15-digit numbers, the memorization strategy eventually breaks down. It appears to be hard to learn relative positional embeddings that precisely encode the distance between two tokens for our task.

The WORDS representation shows stable accuracy in the range of 40-60% from 5 to 15 digits. Our hypothesis for this stability is that the intrinsic position tokens present in this representation (e.g., "hundred", "thousand") make it easier for the model to find and sum two digits that are far apart in the input sequence. However, for 20 digits or more, the models fail at the task. Pretraining might have contributed to the high accuracy on 15 digits or less because the model might have already seen these numbers in this representation in the pretraining corpus. On the other hand, it is very unlikely that the corpus contains numbers of 20 digits or more expressed in plain English. We further investigate the impact of pretraining in Appendix E.

With up to 15 digits, the 10-BASED and 10E-BASED representations achieve accuracy close to 100%. Our explanation for their success is the explicit position tokens added between each digit, which allows the model to inspect the left or right tokens of a digit to determine its significance.

In the Appendices, we present a number of additional experimental results that build on our main findings here. In Appendix B, we study the impact of various position embeddings on the addition task. In Appendix C, we investigate how models of different sizes perform interpolation and extrapolation tasks. Although larger models perform better than smaller ones, we show that not even 3B-parameter models can learn simple arithmetic rules. In Appendix D, we show that all representations can reach accuracies of 97% or more when enough training data is provided. Results here, however, show that representations do matter when training data is scarce. In Appendices E and F, we study how pretraining can impact a model's ability to learn arithmetic. Finally, in Appendix G, we investigate how a mismatch between the length distribution of training and test sets can be problematic for the addition task.

[1] https://github.com/savoirfairelinux/num2words

4 CONCLUSION

Rumelhart et al. (1985) wrote in their germinal "backpropagation" paper that "unfortunately, this [addition] is the one problem we have found that reliably leads the system into local minima". Almost four decades later, despite remarkable progress in neural networks, the field is still exploring this task. Our small contribution is to show that simple manipulations of surface representations to render semantics explicit can help neural models to learn simple arithmetic tasks. It remains to be seen if this "trick" can be applied to other tasks, but our results provide evidence that improving tokenizers and positional encodings are promising directions for future exploration.

ACKNOWLEDGMENTS

This research was supported in part by the Canada First Research Excellence Fund and the Natural Sciences and Engineering Research Council (NSERC) of Canada. In addition, we would like to thank Google Cloud for credits to support this work.

REFERENCES

Daniel Andor, Luheng He, Kenton Lee, and Emily Pitler. Giving BERT a calculator: Finding operations and arguments with reading comprehension. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 5949–5954, 2019.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, pp. 1877–1901, 2020.

Jui Chu, Chung-Chi Chen, Hen-Hsen Huang, and Hsin-Hsi Chen. Learning to generate correct numeric values in news headlines. In *Companion Proceedings of the Web Conference 2020*, pp. 17–18, 2020.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In *International Conference on Learning Representations*, 2018.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, 2019.

David Ding, Felix Hill, Adam Santoro, and Matt Botvinick. Object-based attention for spatio-temporal reasoning: Outperforming neuro-symbolic models with flexible distributed architectures. *arXiv preprint arXiv:2012.08508*, 2020.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 2368–2378, 2019.

Mor Geva, Ankit Gupta, and Jonathan Berant. Injecting numerical reasoning skills into language models. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 946–958, July 2020.

Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. DeBERTa: Decoding-enhanced BERT with disentangled attention. *arXiv preprint arXiv:2006.03654*, 2020.

Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B. Brown, Prafulla Dhariwal, Scott Gray, Chris Hallacy, Benjamin Mann, Alec Radford, Aditya Ramesh, Nick Ryder, Daniel M. Ziegler, John Schulman, Dario Amodei, and Sam McCandlish. Scaling laws for autoregressive generative modeling. *arXiv preprint arXiv:2010.14701*, 2020.

Danny Hernandez, Jared Kaplan, Tom Henighan, and Sam McCandlish. Scaling laws for transfer. *arXiv preprint arXiv:2102.01293*, 2021.

Zhiheng Huang, Davis Liang, Peng Xu, and Bing Xiang. Improve transformer models with better relative position embeddings. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, pp. 3327–3335, 2020.

Chengyue Jiang, Zhonglin Nian, Kaihao Guo, Shanbo Chu, Yinggong Zhao, Libin Shen, Haofen Wang, and Kewei Tu. Learning numeral embeddings. *arXiv preprint arXiv:2001.00003*, 2019.

Devin Johnson, Denise Mak, Andrew Barker, and Lexi Loessberg-Zahl. Probing for multilingual numerical understanding in transformer-based language models. In *Proceedings of the Third BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, pp. 184–192, 2020.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. *Advances in Neural Information Processing Systems*, 28:190–198, 2015.

Łukasz Kaiser and Ilya Sutskever. Neural GPUs learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Nal Kalchbrenner, Ivo Danihelka, and Alex Graves. Grid long short-term memory. *arXiv preprint arXiv:1507.01526*, 2015.

Guolin Ke, Di He, and Tie-Yan Liu. Rethinking the positional encoding in language pre-training. *arXiv preprint arXiv:2006.15595*, 2020.

Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *International Conference on Learning Representations*, 2019.

Jierui Li, Lei Wang, Jipeng Zhang, Yan Wang, Bing Tian Dai, and Dongxiang Zhang. Modeling intra-relation in math word problems with different functional multi-head attentions. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 6162–6167, 2019.

Bill Yuchen Lin, Seyeon Lee, Rahul Khanna, and Xiang Ren. Birds have four legs?! NumerSense: Probing numerical commonsense knowledge of pre-trained language models. *arXiv preprint arXiv:2005.00683*, 2020.

Qianying Liu, Wenyv Guan, Sujian Li, and Daisuke Kawahara. Tree-structured decoding for solving math word problems. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 2370–2379, 2019.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2018.

Swaroop Mishra, Arindam Mitra, Neeraj Varshney, Bhavdeep Sachdeva, and Chitta Baral. Towards question format independent numerical reasoning: A set of prerequisite tasks. *arXiv preprint arXiv:2005.08516*, 2020.

Aakanksha Naik, Abhilasha Ravichander, Carolyn Rose, and Eduard Hovy. Exploring numeracy in word embeddings. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 3374–3380, 2019.

Benjamin Newman, John Hewitt, Percy Liang, and Christopher D. Manning. The EOS decision and length extrapolation. In *Proceedings of the Third BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, pp. 276–291, 2020.

Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 2227–2237, 2018.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Eric Price, Wojciech Zaremba, and Ilya Sutskever. Extensions and limitations of the neural GPU. *arXiv preprint arXiv:1611.00736*, 2016.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

Qiu Ran, Yankai Lin, Peng Li, Jie Zhou, and Zhiyuan Liu. NumNet: Machine reading comprehension with numerical reasoning. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 2474–2484, 2019.

Abhilasha Ravichander, Aakanksha Naik, Carolyn Rose, and Eduard Hovy. EQUATE: A benchmark evaluation framework for quantitative reasoning in natural language inference. In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, pp. 349–361, 2019.

Yuanhang Ren and Ye Du. Enhancing the numeracy of word embeddings: A linear algebraic perspective. In *CCF International Conference on Natural Language Processing and Chinese Computing*, pp. 170–178. Springer, 2020.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. Technical report, Institute for Cognitive Science, University of California, San Diego, 1985.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. In *International Conference on Learning Representations*, 2018.

Imanol Schlag, Paul Smolensky, Roland Fernandez, Nebojsa Jojic, Jürgen Schmidhuber, and Jianfeng Gao. Enhancing the transformer with explicit relational encoding for math problem solving. *arXiv preprint arXiv:1910.06611*, 2019.

Hongjie Shi. A sequence-to-sequence approach for numerical slot-filling dialog systems. In *Proceedings of the 21th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pp. 272–277, 2020.

Alon Talmor, Yanai Elazar, Yoav Goldberg, and Jonathan Berant. oLMpics—on what language model pre-training captures. *arXiv preprint arXiv:1912.13283*, 2019.

Ian Tenney, Dipanjan Das, and Ellie Pavlick. BERT rediscovers the classical NLP pipeline. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4593–4601, 2019.

Avijit Thawani, Jay Pujara, Pedro A. Szekely, and Filip Ilievski. Representing numbers in NLP: a survey and a vision. *arXiv preprint arXiv:2103.13136*, 2021.

Andrew Trask, Felix Hill, Scott E. Reed, Jack Rae, Chris Dyer, and Phil Blunsom. Neural arithmetic logic units. In *Advances in Neural Information Processing Systems*, pp. 8035–8044, 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pp. 5998–6008, 2017.

Eric Wallace, Yizhong Wang, Sujian Li, Sameer Singh, and Matt Gardner. Do NLP models know numbers? Probing numeracy in embeddings. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 5310–5318, 2019.

Benyou Wang, Donghao Zhao, Christina Lioma, Qiuchi Li, Peng Zhang, and Jakob Grue Simonsen. Encoding word order in complex embeddings. In *International Conference on Learning Representations*, 2019.

Xikun Zhang, Deepak Ramachandran, Ian Tenney, Yanai Elazar, and Dan Roth. Do language embeddings capture scales? In *Proceedings of the Third BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, pp. 292–299, 2020.

Yanyan Zou and Wei Lu. Quantity tagger: A latent-variable sequence labeling approach to solving addition-subtraction word problems. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 5246–5251, 2019a.

Yanyan Zou and Wei Lu. Text2Math: End-to-end parsing text into math expressions. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 5330–5340, 2019b.

A Related Work

Recent studies have explored the numerical capabilities learned by neural networks trained on large amounts of texts (Talmor et al., 2019; Jiang et al., 2019; Naik et al., 2019; Wallace et al., 2019; Lin et al., 2020; Johnson et al., 2020; Mishra et al., 2020). See Thawani et al. (2021) for a detailed survey.

A common finding is that the learned embeddings capture *magnitude* (e.g., $2 < 3$), but many models fail to capture *numeracy* (e.g., two=2) (Naik et al., 2019; Wallace et al., 2019; Ren & Du, 2020; Zhang et al., 2020). Character-level models such as ELMO (Peters et al., 2018) have stronger numeracy than sub-word models such as BERT (Devlin et al., 2019), perhaps because two numbers that are similar in value can have very different sub-word tokenizations (Wallace et al., 2019). Our work shows that characters are adequate representations for small to medium numbers, but they are not sufficient when dealing with large numbers, which require precise position representations for each digit.

However, independently of the tokenization method, pretrained word embeddings have trouble extrapolating to numbers unseen during training (Wallace et al., 2019). Some alternatives to improve the extrapolation capabilities of neural models include augmenting pretraining corpora with numerical texts (Geva et al., 2020; Chu et al., 2020) or using scientific notation to represent numbers (Zhang et al., 2020). Similarly, better numerical skills can be achieved by augmenting input texts with pre-computed numerical computations (Andor et al., 2019) or by explicitly inferring mathematical equations from natural language text (Zou & Lu, 2019a;b; Li et al., 2019; Liu et al., 2019; Shi, 2020).

Special architectures have also been proposed for arithmetic tasks (Kaiser & Sutskever, 2015; Kalchbrenner et al., 2015; Price et al., 2016; Trask et al., 2018). Many of these models are capable of summing numbers larger than the ones seen during training. In contrast, more general-purpose architectures fail to extrapolate on numerical tasks (Joulin & Mikolov, 2015; Dehghani et al., 2018; Schlag et al., 2019).

Others have proposed neural–symbolic hybrids, which are typically composed of a neural model to convert inputs to contiguous vector representations and a symbolic component that applies rules over these vectors (Ran et al., 2019). However, a body of evidence has shown that neural networks can perform reasoning tasks. For instance, a modern pretrained model with self-attention that uses the right level of input representation can outperform neural–symbolic hybrids on artificial reasoning tasks that require answering questions from videos (Ding et al., 2020). Deep learning models were also successfully applied to symbolic integration, to solve differential equations (Lample & Charton, 2019), and automated theorem proving (Polu & Sutskever, 2020).

Furthermore, it is not clear how architectures specialized to some tasks can be adapted to simultaneously perform a range of tasks a human is capable of. Our work instead focuses on a general-purpose architecture that can be applied to almost all natural language processing tasks.

Novel ways of encoding positions of tokens in the transformer architecture have been proposed, but they were mostly evaluated on natural language processing tasks, showing small performance gains (Ke et al., 2020; He et al., 2020; Wang et al., 2019; Huang et al., 2020). We instead expose the limitations of subword tokenizers and positional encodings using simple arithmetic tasks.

Datasets such as DROP (Dua et al., 2019), EQUATE (Ravichander et al., 2019), or Mathematics Questions (Saxton et al., 2018) test numerical reasoning; they contain examples that require comparing, sorting, and performing other complex mathematical tasks. This work focuses on isolating the failure cases of the transformer architecture by studying how it performs simple arithmetic tasks. We argue that this is a necessary skill to solve more complex reasoning tasks.

B Position Embeddings

Here, we study the impact of various position embeddings on the addition task. Since pretraining from scratch is a costly process, we experiment with only small transformer models fine-tuned without pretraining.

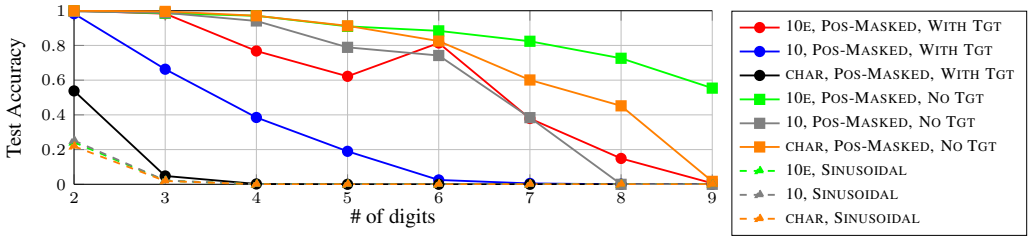

Figure 2: Addition accuracy of vanilla transformers with different position encoding methods.

The architecture of the transformer follows Vaswani et al. (2017) except we use 4 layers for the encoder and the decoder, respectively. We look into the effect of representation and positional encoding on addition from 2 digits to 9 digits. Due to the cost of these experiments, we choose a subset of the representations studied in Section 3: 10E-BASED, 10-BASED, and CHARACTER.

The dataset is split into training and test sets with a ratio of 9:1. For 3–9 digits addition, we randomly generate 10,000 samples for the whole dataset. For 2-digit addition, we use all of the combinations for every addend $a \in [10, 99]$, which results in less than 10,000 samples. The models are trained for 55 epochs with a learning rate of 10^{-5}.

We find that the original positional encoding in Vaswani et al. (2017) fails to learn addition effectively, as shown in Figure 2. This might be due to the correlation introduced by two heterogeneous signals—embedding and absolute positional encoding (Ke et al., 2020). Therefore, we designed a position-wise masked embedding for this task.

More specifically, for an n-digit number whose embedding is e with embedding size d, we will set $e[u : v] = 1$ for $i - th$ digit in the number, where $u = int(\frac{d}{n}) \cdot (n - i)$ and $v = int(\frac{d}{n}) \cdot (n - i + 1)$. We set other position embedding values to 0. Note that i follows the "Big-Endian" style (e.g., $i = 3$ for "2" in the number "271"). However, during inference, digit information is not provided for the target sequence as we don't know the exact digit of the decoded number in advance. So, we face a format discrepancy between training and inference. To investigate how this discrepancy will affect the result, we train the model in two different ways—training with target position provided and training without target position provided (position encoding for the target is the zero vector). Note that position encoding is provided for the source sequence in both cases for training and inference; position encoding is not provided for the target sequence during inference in both cases. The results are shown in Figure 2, labeled as "WITH TGT" and "NO TGT", respectively. We label our position-wise masked embedding as "Pos-Masked". The original representation is called "Sinusoidal".

Consistent with previous experiments, 10E-BASED performs best given the same position encoding and training strategies. Comparing "WITH TGT" and "NO TGT", we can see that training with target position encoding creates fluctuations among different digits. In general, it performs worse than training without target position encoding given the same encoding representation. Unsurprisingly, under our experiment setting, whether the target position is provided is not as important as having the same format between training and inference.

C EXPERIMENTS ON EXTRAPOLATION

One advantage of working with arithmetic tasks is that the rules to be learned are well defined and relatively simple. Thus, it is easy to verify if models learned such rules by evaluating them on numbers that are larger than the ones they were trained on. If successful, such a model would have no problem correctly adding or subtracting arbitrarily long numbers.

In this section, we investigate how models of different sizes perform interpolation and extrapolation tasks. We train T5-60M, T5-220M, T5-770M, and T5-3B models on numbers that are sampled using the "balanced" method. Models are trained 100K iterations using batches of 128 examples and a learning rate of 10^{-3}. We save checkpoints every 2,000 iterations, and the best checkpoint is chosen using a separate validation set of 10,000 examples. The models are evaluated on a test set of 10,000 examples with numbers sampled using the "random" method.

	Interpolation				Extrapolation			
Order:	Inverse		Regular		Inverse		Regular	
Operation:	Add	Sub	Add	Sub	Add	Sub	Add	Sub
T5-60M	**1.000**	0.934	**0.998**	0.830	0.000	0.000	0.004	0.000
T5-220M	**1.000**	**0.998**	**1.000**	**0.995**	0.000	0.000	0.862	0.641
T5-770M	**1.000**	0.947	**0.999**	**0.982**	0.003	0.000	0.442	0.373
T5-3B	**1.000**	**0.997**	**1.000**	**0.993**	**0.974**	0.865	**0.988**	**0.982**

Table 2: Interpolation and extrapolation accuracy. Interpolation refers to training and testing on up to 60-digit numbers. Extrapolation refers to training on up to 50-digit numbers and testing on 60-digit numbers. We highlight in **bold** accuracy above 97%.

For interpolation experiments, the models are trained and evaluated on up to 60-digit numbers. For extrapolation experiments, the models are trained on up to 50-digit numbers and evaluated on 60-digit numbers. We use that many digits for training because the models could not extrapolate with fewer; see more below.

Regular vs. inverse orders: Auto-regressive models such as the ones used in this work generate the output sequence token by token. Thus, to produce the first digit of the answer, which is the most significant one, the model has to perform all the carry operations. In the addition example "What is 52 plus 148?", to produce the first digit "2", the model has to perform the carry operation for the unit digits (2 and 8), and then the carry for the decimal digits (5 and 4). Hence, the model has to perform the digit-wise addition (or subtraction) of all the digits in the question before generating the first digit of the answer. We call this generation order "regular".

Another way to produce an answer is by generating the least significant digits first. This order is perhaps easier to learn than the "regular" order because to decode each digit, the model only needs to add (or subtract) single digits and check if the previous digit-wise operation had a carry. We call this generation order "inverse".

The results presented in Table 2 show that models of all sizes successfully perform interpolation tasks. Two exceptions are T5-60M on the subtraction tasks, which achieve 0.934 and 0.830 accuracy for inverse and regular orders, respectively. Nevertheless, compared to the extrapolation results, these numbers are high enough to consider them as successful runs.

On extrapolation tasks, T5-3B succeeds on almost all of them, whereas smaller models fail more often. Even on tasks where T5-220M achieves reasonable accuracy (0.862 and 0.641 on addition and subtraction using regular order, respectively), T5-3B outperforms T5-220M by large margins. This result provides evidence that larger models might perform better on data whose distribution is outside its training data distribution. However, it remains to be investigated if this trend holds for more complex tasks, especially those involving natural language.

The difference in accuracy is negligible between regular and inverse orders on interpolation tasks. However, models trained and evaluated on the regular order show higher extrapolation accuracy than those that use the inverse order. For example, T5-220M fails to extrapolate on both addition and subtraction tasks when using the inverse order (i.e., accuracy is zero), but it performs better when using the regular order, with accuracy between 60–90%. This result is perhaps surprising since one would expect that the inverse order would be easier to learn.

Supported by recent work, we suspect that the problem is related to the bias of selecting the termination (i.e., end-of-sequence) token when the generated sequence becomes longer than those seen during training (Newman et al., 2020). In the inverse order, the answer is generated from least to most significant digit, so the model might have a tendency to select the termination token right after it generates the most significant digit seen during training. In the regular order, however, the model has to predict the full length of the sequence before emitting the first and second tokens. For example, the first two tokens of the answer to the question $10^{60} + 10^{60}$ are "2" and "10e60". This explicit length prediction allows the model to better generalize to longer sequences, but it appears to be insufficient to induce models to learn addition rules that are independent of the length of numbers seen during training (more below).

We observe high variance in accuracy for the extrapolation experiments. For example, during the training of a T5-770M model on up to 30-digit numbers, the accuracy ranges from 20% to 50% when evaluated on 60-digit numbers. Extrapolation accuracy also oscillates between 20–40 percentage points when changing the seed for training data generation.

Extrapolation is hardly achieved when trained on fewer than 50 digits, regardless of the model size. For example, T5-220M, T5-770M, and T5-3B trained on 15 digits show an accuracy of zero when evaluated on 20 digits.

Beyond a critical amount, increasing the training data does not improve extrapolation accuracy. For example, when trained on up to 30-digit and evaluated on 60-digit numbers, a T5-770M showed a similar accuracy range (20%–50%) when trained with either 100K, 1M, or 10M examples. As training progresses, interpolation accuracy always reaches 100%, but extrapolation accuracy starts to decrease after some number of training steps. The number of training steps after which this drop occurs varies dramatically between runs that differ only in the seed used to generate the training data. We are unable to isolate the cause of this behavior.

Contrary to the hypothesis of Newman et al. (2020), we find that the end-of-sequence token does not seem to be the cause of extrapolation failures. For example, when a T5-770M model trained on 30-digit numbers is evaluated on 60-digit numbers, it correctly generates the first 23 position tokens (i.e., from "10e60" until "10e38") but it suddenly skips to position token "10e27", and continues generating the correct position tokens until the last one ("10e0"). Here we show one such sequence:

> 1 10e60 0 10e59 1 10e58 2 10e57 3 10e56 0 10e55 2 10e54 7 10e53 0 10e52 1 10e51 0 10e50 3 10e49 9 10e48 0 10e47 5 10e46 3 10e45 1 10e44 5 10e43 3 10e42 6 10e41 3 10e40 6 10e39 0 **10e38** 8 **10e27** 1 10e26 4 10e25 1 10e24 2 10e23 6 10e22 6 10e21 9 10e20 5 10e19 3 10e18 4 10e17 8 10e16 3 10e15 8 10e14 8 10e13 9 10e12 5 10e11 3 10e10 5 10e9 0 10e8 6 10e7 4 10e6 3 10e5 5 10e4 6 10e3 7 10e2 2 10e1 2 10e0

Hence, although the model correctly emits the end-of-sequence token after the "10e0" token, it decides to shorten the sequence in the middle of the generation, i.e., by skipping position tokens "10e37" until "10e28". This skipping behavior is consistent across model sizes, dataset sizes, and extrapolation ranges (e.g., training on 20 digits, evaluating on 30 digits, etc.). Investigating it further might help us understand why neural models often fail on extrapolation tasks.

D Impact of data size

In Section 3, we show that the choice of orthography has a large impact on the addition task when training data is scarce (i.e., 1,000 training examples). In this section, we investigate how these representations perform with varying amounts of training data. We train and evaluate T5-220M on the addition task of up to 30-digit numbers using the regular order. Due to the high computational cost of training this model on millions of examples, we reduce the number of epochs depending on the dataset size, which is detailed in Table 3. We select the best checkpoint using a validation set of 10,000 examples and evaluate the models on a test set of 10,000 examples.

Size	Epochs
10^3	200
10^4	100
10^5	20
10^6	10
10^7	1

Table 3: Number of training epochs for each dataset size presented in Figure 3.

Results are shown in Figure 3. The 10E-BASED representation presents the best results for training sizes of 1,000 and 10,000 examples, followed by 10-BASED, WORDS, UNDERSCORE, CHARACTER, and DECIMAL. For larger datasets such as 10M examples, almost all representations achieve more

than 99.9% accuracy. The exception is the DECIMAL representation, which still has a high error of 2.1% even when trained with 10M examples.

We conclude that with enough training data, models can learn the addition task regardless of the representation. The limitations of some representations are exposed only when training data is small.

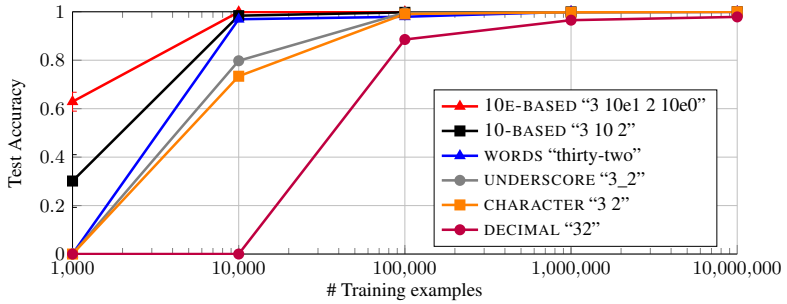

Figure 3: Accuracy of different number representations when varying the amount of training examples. The task is addition of 30-digit numbers.

E PRETRAINED VS. FROM SCRATCH MODELS

One hypothesis for the high interpolation accuracy reported in Section 3 despite using a small number of training examples is that the model has already seen addition and subtraction examples during pretraining. To test this hypothesis, we compare pretrained models with models trained from scratch (i.e., no pretraining on the masked language modeling task) on the addition task. In this experiment, the models never see the same training example more than once. That is, they are not limited by training data.

Figure 4 shows that both pretrained T5-220M and T5-3B need approximately ten times fewer training examples (and compute) than models trained from scratch to reach 100% accuracy on the addition of 60-digit numbers.

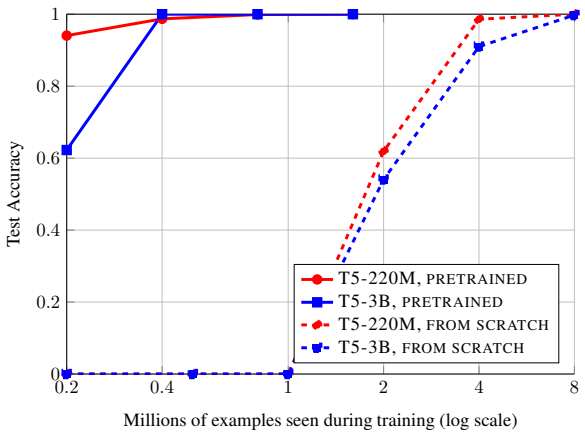

Figure 4: Accuracy of pretrained models vs. from scratch models with respect to the number of training examples. Models are trained and evaluated on numbers with up to 60 digits in length.

F ACCURACY ON DIFFERENT BASES

Here we propose another way to test how pretraining can impact a model's ability to learn arithmetic. We hypothesize that a model might have difficulty learning bases different than base 10 (i.e.,

	Test Accuracy	
Base	From Scratch	Pretrained
2	0.000 ± 0.000	$\mathbf{0.999} \pm 0.001$
3	0.000 ± 0.000	$\mathbf{0.999} \pm 0.002$
10	0.000 ± 0.000	$\mathbf{0.993} \pm 0.003$
19	0.000 ± 0.000	$\mathbf{0.976} \pm 0.007$

Table 4: Test set accuracy of 15-digit addition on various bases. Numbers are represented with 10E-BASED orthography.

decimal) because examples rarely occur in the pretraining corpus. To test this hypothesis, we train a T5-220M model on addition examples using binary, ternary, decimal, and base 19. While there might be examples of binary addition in the pretraining corpus, our expectation is that it contains few (if any?) examples of addition using base 19 numbers. We use the 10E-BASED orthography and inverse order due to its slightly better accuracy (see Table 2). We also evaluate models trained from scratch.

We report the mean accuracy and 95% confidence intervals of a model trained with five different sets of 1,000 addition examples for 100 epochs. A separate development set of 1,000 examples was used to select the best checkpoint of each run. We trained and evaluated on numbers equivalent to 15 decimal digits.

For these experiments, we use only 1,000 training examples since experiments in Appendix D show that models can successfully learn with enough training data, thus too much data defeats the purpose of measuring the impact of pretraining; see also Hernandez et al. (2021). Results are shown in Table 4. The pretrained model has no problem learning binary, ternary, and decimal bases, but its accuracy degrades slightly on base 19. Since it is unlikely that the pretrained model has encountered substantial numbers of examples of addition in rare bases (i.e., ternary and 19), it seems that pretraining helps on this task in other ways than simple memorization.

To show that the task is not easy, we also report in the table that models trained from scratch fail to learn the task regardless of the base. This result is expected since a large number of parameters (220M) need to be learned from scratch using just 1,000 examples.

G Impact of Different Length Distributions

Here we investigate to what extent a mismatch between the length distribution of training and test sets is problematic for the addition task. We train T5-220M models on 100,000 examples, select the best checkpoint using a development set of 10,000 examples, and evaluate on another 10,000 examples. Here we use the regular order. Training and test sets are generated using either the balanced or random sampling methods described in Section 2.

Results are shown in Table 5. When trained on the balanced distribution, the model succeeds on both random and balanced evaluation sets. When trained on the random distribution, it succeeds on the random evaluation set, but it fails on the balanced evaluation set. In other words, when trained on data where most numbers (i.e., 90%) have 60 digits, it does not learn to add numbers with fewer digits. This shows that models have problems performing addition of sequences shorter than the ones seen during training. This is complementary to the results presented in Appendix C, which shows that models cannot generate examples longer than the ones seen during training.

		Test	
		Balanced	Random
Train	Balanced	1.000	1.000
	Random	0.014	1.000

Table 5: Accuracy on 60-digit addition, with balanced and random sampling as described in Section 2.