Evaluating PaLM-FLAN-T5 and previous models on syntax

Kalle Hilsenbek, Nila Ravindran, Naveen Malla, Priyabanta Sandulu

***Abstract*—For the first time a language model is fully evaluated on all target word predictions of the cross-lingual syntax assessment. We analyze the T5-version of PaLM and previous models on the English, German and French syntax test suit. We publish the code to screen T5-models at github[1]. Also, we provide an annotated PaLM code in the appendix.**

CONTENTS

I. INTRODUCTION

Using few-shot learning, large language models (LMs) demonstrated outstanding performance across a wide range of natural language tasks [1]. Pathways Language Model (PaLM) [2] was trained using Pathways, a machine-learning system that enables very efficient training across a wide collection of instruction tasks on several paralleled hardware accelerators. PaLM outperformed the fine-tuned state-of-the-art on a series of multi-step reasoning tasks and outperformed average human performance on the recently published BIG-bench test, achieving breakthrough performance on hundreds of language comprehension and generating benchmarks. PaLM is also good at multilingual tasks and source code creation (see figure 1 for the variety of tasks).

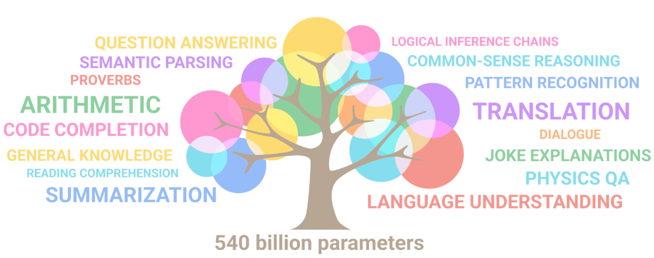

Figure 1: As the scale of the model increases, the performance improves across tasks while also unlocking new capabilities.

[1]github.com/Bachstelze/clams

Fine-tuning the model on a bigger set of instruction tasks in "Scaling Instruction-Finetuned Language Models" (FLAN) [3] improved the model to learn and follow explicit instructions and perform a wide range of tasks that require reasoning and understanding of natural language. There are smaller model versions of PaLM-FLAN[2], which are released as FLAN-T5 on huggingface, whereas the five initial letters of T5 are standing for "Text-to-Text Transfer Transformer" [4], a bigger standard encoder-decoder transformer with the commonly used masked language modeling. By improving the instruction-finetuning process, the open-sourced model PaLM-FLAN-T5 can learn from a large and diverse set of instruction tasks, enabling it to improve its performance on a variety of language tasks. This approach helps to improve the model's ability to reason, understand and generate natural language by fine-tuning the model on a bigger and more diverse range of language tasks.

Besides the impressive results on downstream tasks, none of the fundamental language modeling of PaLM was analyzed. So far we could only assume that large parts of Bertology also apply to PaLM due to the fact that PaLM is based on the transformer architecture. We use the Cross-Linguistic Assessment of Models on Syntax (CLAMS) [5] to make a well-founded analysis. CLAMS utilizes the cloze pretraining objective to measure the syntax modeling ability. Because most sentences are grammatically straightforward and most words can be anticipated from their local context, ambiguity rewards LMs largely for semantic predictions, making the quality of the LMs syntactic predictions appear to be difficult to quantify. The evaluation in CLAMS is based on "Assessing the Ability of LSTMs to Learn Syntax-Sensitive Dependencies" [6], which identified syntactically difficult sentences in corpora.

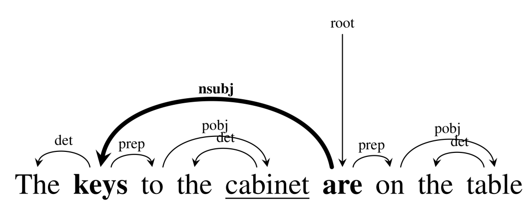

Figure 2: Syntax example from "Assessing the Ability of LSTMs to Learn Syntax-Sensitive Dependencies" [6].

The copied example in figure 2 from their research describes the syntactic prediction: The form of the verb is determined by the head of the subject, which is directly connected to it via a nominal subject edge. Other nouns that intervene between the head of the subject and the verb (here cabinet is such a noun) are irrelevant in determining the form of the verb and can be ignored.

[2]in comparison to the biggest and unpractical 540 billion PaLM

II. RELATED WORK TO PALM

Working with many pathways yields a big variety of related work, which we summarize in this section.

A. Prompting

Large language models use a prompting mechanism that allows them to learn from a few examples and to generalize to a wide range of tasks [7]. The prompting mechanism works by providing cues to the model such as hints, keywords, and questions. These cues help the model identify the correct answer or action and give it the ability to generalize beyond the examples it was trained on. The prompting mechanism can be used to efficiently learn from a few examples, making it ideal for few-shot learning [8] [9]. The model relies on correctly interpreting the syntax of the input to make accurate predictions. PaLM can quickly adapt to new tasks and conditions with its good language modeling abilities, making it well-suited for a wide variety of natural language tasks.

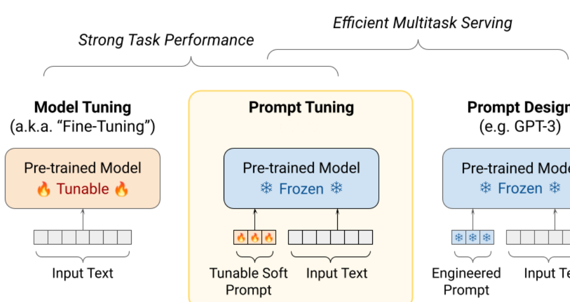

Figure 3: Comparing Model Tuning, Prompt Design, and Prompt Tuning: Prompt tuning retains the strong task performance of model tuning, while keeping the pre-trained model frozen, enabling efficient multitask serving [10].

However, it can also be challenging to design effective prompts that flow logically and account for a wide range of possible inputs and scenarios. Therefore, there are many approaches and research of prompting[3] for e.g. linguistic structures [27]. One particular prominent approach is the chain of thoughts prompting [28]. It emerged with the recent large language models and it involves generating a sequence of prompts or questions that build on each other to guide the model toward the desired output. This mechanism is often used in logical reasoning and conversational AI, where the goal is to generate a coherent and contextually relevant response to a user's input. In the chain of thoughts prompting mechanism, the prompts or questions are designed to flow logically from one step to the next, with each step building on the information provided by the previous one. The chain-of-thoughts prompting mechanism is useful because it allows the model to gradually build up a complex logic and a more complete understanding of the user's intent and context, which can help it generate more accurate and relevant responses (see figure 5 for a comparison with the standard prompting).

[3]Examples for prompting research: [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26](see figure 8 for an overview)

B. PaLM aspects and results

PaLM model has demonstrated its remarkable abilities on various tasks including Language Understanding and Generation, Reasoning and Code-Related tasks

Language Understanding and Generation:PALM underwent comparison with state-of-the-art (SOTA) models based on its performance in 29 English-based NLP tasks. Results showed that PALM outperformed the other SOTA models on 28 of the 29 tasks, particularly in the area of few-shot performance [2]. Figure 4 depicts the performance improvement of PaLM 540B over SOTA approaches

Performance of PaLM on Using BIG-bench tasks: PaLM was evaluated using BIG-bench [29], a collaborative benchmark for probing large language models. The tasks included in BIG-bench are highly challenging, with an average human success rate of only 50 percent. However, by scaling PaLM from 8 billion parameters to 540 billion parameters, it was discovered that PaLM 5-shot performance exceeded that of the average human performance [2].

Reasoning tasks: PaLM was evaluated on reasoning tasks that requires multi-step reasoning. PaLM demonstrated significant performance improvement compared to previous language models through the use of chain of thought prompting. The generation of intermediate reasoning steps by the large language models contributed significantly to the improvement in accuracy.

Performance of PaLM on reasoning tasks: PaLM, utilizing 8-shot prompting, outperforms previous state-of-the-art (SOTA) approaches by achieving a success rate of 58 percent on GSM8K benchmark [30], a benchmark consisting of several complex math questions commonly asked in grade-school. The enhancement in performance is thought to be due to PaLM's seperate encoding of digits within its vocabulary. This result surpasses the prior highest score of 55 percent by the SOTA models [2].

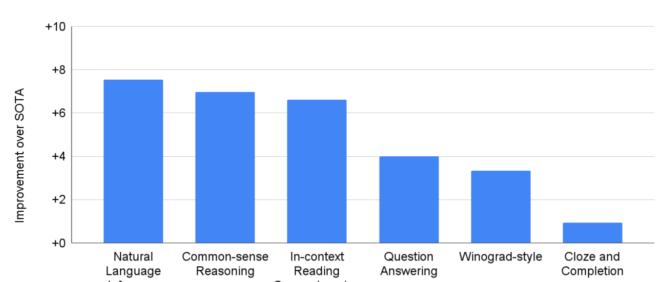

Figure 4: Performance improvement of PaLM 540B over State Of The Art results [31].

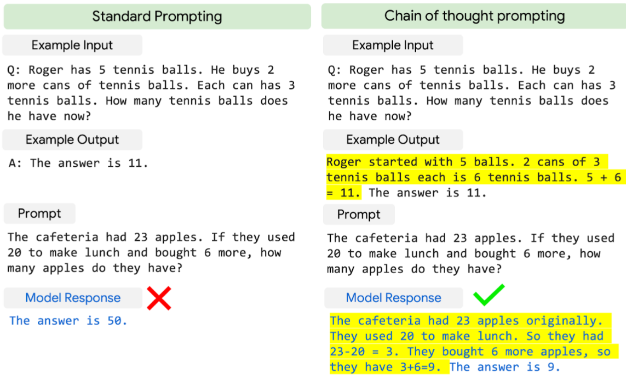

Figure 5: Standard prompting vs Chain of thought prompting [31].

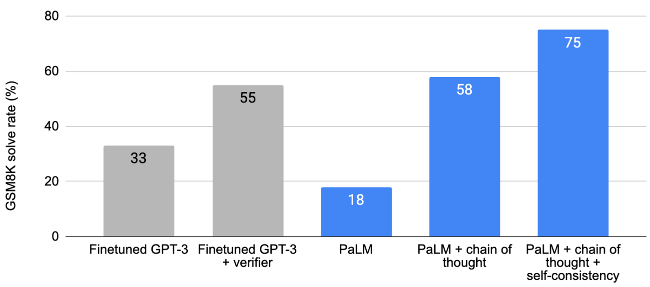

Figure 6: PaLM vs State-of-the-art results on GSM8K benchmark [32].

Code generation tasks: PaLM has demonstrated proficiency in performing coding tasks such as generating code from natural language descriptions, translating code from one programming language to another, and resolving errors during code compilation [31].

Performance of PaLM on Code-Generation tasks: PaLM 540B demonstrates comparable performance to the fine-tuned Codex 12B model, while requiring 50 times lesser Python code for training the model. As a large model, PaLM is also more sample efficient. To achieve a better performance, PaLM was fine-tuned on a dataset consisting only the Python code, named PaLM-Coder [2] [31].

		Pretraining only		Code Finetuning			
		LaMDA 137B	PaLM 540B	Codex 12B[a]	Davinci Codex*	PaLM Coder 540B	Other Work
HumanEval (0)	pass@100	47.3	76.2	72.3	81.7	**88.4**	–
MBPP (3)	pass@80	62.4[b]	75.0	–	**84.4**	80.8	–
TransCoder (3)	pass@25	–	79.8	–	71.7	**82.5**	67.2[c]
HumanEval (0)	pass@1	14.0	26.2	28.8	**36.0**	**36.0**	–
MBPP (3)	pass@1	14.8[b]	36.8	–	**50.4**	47.0	–
GSM8K-Python (4)	pass@1	7.6	**51.3**	–	32.1	50.9	–
TransCoder (3)	pass@1	30.2	51.8	–	54.4	**55.1**	44.5[c]
DeepFix (2)	pass@1	4.3	73.7	–	81.1	**82.1**	71.7[d]

Figure 7: Results obtained by the PaLM 540B and PaLM-Coder 540B models across code synthesis and software engineering tasks. [2].

C. Memorization

Memorization of a language model refers to the phenomenon where the model generates text that is identical or nearly identical to text it has seen before, rather than producing novel responses.

The degree to which the model memorizes text depends on factors like size of the model, amount of training data, and the nature of the data. To test this, the first 50 tokens from the span were used to prompt the model with 100 token sequences chosen at random from the training samples. Greedy decoding was performed, and the frequency with which the model generated a 50-token continuation that precisely matched the training example was measured.

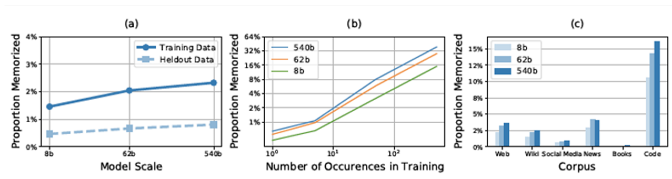

Figure 8: The graphs show the relationship between model size, training data frequency, corpus size, and the proportion of memorized text by the Google Pathway Language Model, highlighting the trade-off between memorization and generalization ability of the model [10].

- When the size of the model grows, the rate of memorizing increases.
- The analysis reveals that examples seen once have a low memorization rate, while those seen more than 500 times have a high memorization rate.
- The amount of exact duplication, near duplication, or templating in the training data is found to be the biggest differentiator in memorization rates.

D. Data contamination

Contamination refers to the presence of unintended biases or noise in the dataset that can affect the accuracy of the language model. High data overlap rates between benchmark assessment sets and training data have been reported by prior studies, however these investigations solely focused on the occurrence of high-order n-grams. In order to discover which of 29 major English NLP benchmark tasks had a high proportion of contaminated examples, this new study generated statistics and carefully reviewed examples from each work. Based on the way the dataset was created, the 29 tasks were separated into four groups, such as Wholesale contamination, Constructed from web, Context on web and No significant overlap.

The PaLM study examined and categorized data sets as "clean" or "contaminated" based on whether the data had been seen during training. Results showed that data contamination did not significantly affect accuracy. The study also compared the performance of two models(8B and 540B) and found that neither had memorized the evaluation sets.

E. Interview with ChatGPT

There are similar models to PaLM from other companies and researchers. ChatGPT [33] made it into the article headlines with its chat interface. It is a specific instance of instructGPT [34] that

has been fine-tuned on conversational data to generate human-like responses to user input. It's important to note that the following interview is responded by ChatGPT, an AI language model.
Interviewer: Can you help us with our questions about ML4NLU?
ChatGPT: Sure, I'd be happy to help.
Interviewer: Are you proficient in English?
ChatGPT: Yes, English is one of the languages I'm proficient in.
Interviewer: Can you describe yourself in detail?
ChatGPT: I'm a large language model based on the Transformer architecture. I was trained on a massive dataset of text from the internet using unsupervised learning techniques, and I've been fine-tuned on various NLP tasks to improve my performance.
Interviewer: Why is syntax modeling important in natural language understanding, and why is it challenging for language models to achieve high accuracy in this area?
ChatGPT: Syntax modeling is critical for natural language understanding because it helps to disambiguate sentences and understand their intended meaning. However, it is also challenging for language models to achieve high accuracy in this area because natural language is often ambiguous, and there are many different ways to structure a sentence while still conveying the same meaning.
Interviewer: Are there any emerging techniques or approaches that could improve syntax modeling in language models?
ChatGPT: Yes, there are various emerging techniques or approaches that could improve syntax modeling in language models, such as using graph-based representations, incorporating explicit syntactic structures into the model architecture, or leveraging external knowledge sources like ontologies or databases.
Interviewer: How do language models like ChatGPT and PaLM process and analyze language data differently from traditional rule-based approaches?
ChatGPT: Language models like ChatGPT and PaLM process and analyze language data differently from traditional rule-based approaches by using deep learning techniques. Rather than relying on hand-crafted rules and heuristics, these models learn to represent the statistical patterns in language data and use these representations to generate text or perform other NLP tasks.
Interviewer: How do you deal with syntactic errors or ambiguities in the input or output?
ChatGPT: To deal with syntactic errors or ambiguities in the input or output, language models can use various techniques, such as using context clues or leveraging external knowledge sources like ontologies or databases.
Interviewer: Why do language models based on T5 score not even half the accuracy of a perfect syntax language model?
ChatGPT: There could be multiple reasons why language models based on "Text-to-Text Transfer Transformer" (T5) may score half the accuracy of a perfect syntax language model. Here are a few possible explanations:

Training Data: The accuracy of a language model depends heavily on the quality and quantity of training data. If the training data used to train the T5 model is insufficient, noisy, or biased, the model's performance may suffer.

Model Complexity: The complexity of the model architecture may also affect its performance. While T5 is a powerful transformer-based model, it may not be designed specifically for syntax modeling. It is possible that a more specialized architecture tailored for syntax modeling would perform better.

Evaluation Metrics: It's important to note that the evaluation metrics used to measure the performance of a language model can also influence the reported accuracy. Depending on the task and dataset, some metrics may be more appropriate than others. Therefore, it's important to use appropriate evaluation metrics that accurately reflect the model's performance on the given task.

Natural Language Variation: Natural language is incredibly diverse, and it can be challenging to build a model that can handle all the variations that exist. Even a perfect syntax language model may struggle to handle the full range of natural language syntax.

Overall, the accuracy of a language model depends on many factors, and it can be challenging to pinpoint a single reason for a model's performance. It is essential to carefully analyze the model's performance and use appropriate evaluation metrics to gain a deeper understanding of its strengths and weaknesses.
Interviewer: Why is there only a reimplementation of RLHF on top of the PaLM architecture[4], but no open-source model like BLOOM-GPT from the open-science community?
ChatGPT: There may be many reasons why certain models are not available as open-source projects, including proprietary technology or intellectual property issues. However, there are many other language models and frameworks available as open-source projects, including GPT-3, BERT, and others.
Interviewer: Can we publish this dialogue as an interview in our research paper?
ChatGPT: Yes, you're welcome to use our dialogue as an interview in your research paper.

III. Syntax results

We evaluate a promising subset of the over 5000 T5-models from huggingface on a subset of the syntax test suit.)) and use the syntax test suit from "Cross-Linguistic Syntactic Evaluation of Word Prediction Models" (CLAMS [5]), which is using non-recursive context-free grammar to construct the test data. The different test cases are well-defined in CLAMS.

It could be the first time that a language model is completely evaluated on English, German and French with this type of test suite. The previous models like RNN, BERT and XLM-R used only those target words that fully are in their vocabulary. It is possible to predict those missing tokens with a combination of the subtokens, but as far as we know, it wasn't done till today. The limitation of target words can distort the resulting score as it is accented in "Reproducing targeted syntactic evaluations of language models" [5]. So only the human prediction is our comparative value (see table 1). We took the map-reduce-algorithm

[4] https://github.com/lucidrains/PaLM-rlhf-pytorch
[5] This paper [35] is uploaded to that repository: github.com/Bachstelze/clams

of the previous paper, which partitions the dataset into similar target words, predicts the scores within small batches of one target word and in the end reduces the two scores for each sentence. We only changed the huggingface config, tokenizer, modelclass and pipeline [6].

Subject-verb-agreement cases	Human scores
Simple agreement	0.96
Verb-phrase coordination (short)	0.94
Verb-phrase coordination (long)	0.82
Across subject relative clause	0.88
Within object relative clause	0.78
Across object relative clause	0.85
Across prepositional phrase	0.85
Average accuracy	0.85

Table 1: Subject-verb-agreement accuracy of humans on all target words provided by Marvin and Linzen [36].

Accuracy is used as measurement. E.g. an accuracy score of 0.85 means that 85 out of 100 tests are correctly predicted. Or in technical terms: The probability of the correct word is higher than the wrong word in 85 out of 100 tests. A score of 0.5 is a random result as you could statistical throw a coin as prediction - in half of the cases this would be correct.

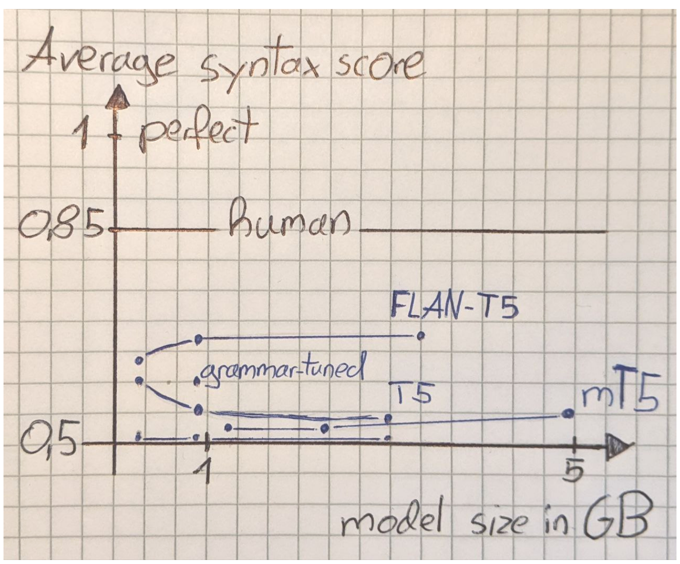

Figure 9: Results scored by the different T5-models in comparison to the human reference.

A. Results and interpretation of T5

Based on the results obtained from the T5 models[7], we can see that the t5-small achieves the highest overall average score (see table 2), followed by the t5-base model with an overall average score of 0.55 (see table 3), and then the T5-large model with an overall average score of 0.53.

This indicates that smaller T5-small model performs better on the Natural language understanding tasks in CLAMS when compared to the larger T5-base and T5-large models. One possible reason for this could be smaller model may have fewer parameters and thus may be better at generalizing new tasks, whereas the large models may be overfitting to the specific task.

Additionally, we can also observe that the performance of each model differs across the different languages.For instance, the T5-small model performs better in English with an accuracy score of 0.65, compared to French and German with scores of 0.61 and 0.54. Similarly, the T5-base model performs better in French with an accuracy score of 0.63, compared to English and German with random scores of 0.50 and 0.51. This implies that performance of each model can also be influenced by the language evaluated.

Models based on T5-base that are fine-tuned on English grammar and error correction improve over the pre-trained model, but not over T5-small. Suprisingly the model T5-base-grammar-correction[8] has its highest syntax score in French and not in the trained English language. Another model called "error corrector"[9] has overall the same score of 0.59 (see tables 5 and 6) by adapting to English and German, but then by reducing its French score.

B. Results and interpretation of efficient-T5

Based on the results obtained from efficient-T5 models [10], we can see that T5-efficient-small, T5-efficient-base, T5-efficient-large all achieved the average accuracy score of 0.51 in CLAMS (see the bottom line in figure 9). We can infer from this that the size of the T5-efficient model doesn't significantly affect its performance unlike the T5 models.

Additionally, we can also observe that the performance of T5-efficient model is consistent across different languages. All the T5-efficient models score 0.50 in English while they score between 0.52 and 0.55 in French and German.

C. Results and interpretation of mT5

Based on the scores obtained from mT5 models, we can see that the mT5-small and mT5-base[11] achieved similar overall average of 0.52, while the larger mt5-large model achieves the overall average score of 0.55. mT5 models are multilingual models which are specifically designed to perform well across multiple languages. Therefore, we can observe that the performance of each mT5 model is consistent across different languages, with the scores ranging from 0.50 to 0.58

One possible reason for the higher performance of mT5-large

[6]The Colab-notebook to run the evaluation: colab.research.google.com/drive/1hlpXsCf-K4WW1sYBtRtNasc0k5-g46vs

[7]The model card of T5 [4] on huggingface with an interactive interface: https://huggingface.co/t5-small

[8]The model vennify/t5-base-grammar-correction was fine-tuned on a fluency corpus for grammatical error correction (JFLEG) [37]: huggingface.co/vennify/t5-base-grammar-correction

[9]The model card of the error corrector: huggingface.co/prithivida/grammar_error_correcter_v1

[10]The model card of efficient-T5 [38] on huggingface with an interactive interface: https://huggingface.co/google/t5-efficient-base

[11]The model card of mT5 [39] on huggingface with an interactive interface: https://huggingface.co/google/mt5-base

models may due to its model size. The large models may be better at learning complex language patterns, resulting in better accuracy scores.

D. Results and interpretation of PaLM-FLAN-T5

Based on the accuracy scores obtained from FLAN-T5 models, we can see that the FLAN-T5-small achieved the overall accuracy score of 0.63, while the larger models flan-t5-base[12] and FLAN-T5-large achieved the overall accuracy score of 0.67

Flan-t5 model is the fine-tuned version of the T5 model. It is specifically designed to show good performance for natural language processing tasks. Therfore, we can see that the performance of FLAN-T5 models are relatively consistent across different languages, with accuracy scores ranging between 0.59 and 0.73. Comparing the other models evaluated, Flan-T5 models achieved higher average accuracy score in CLAMS.

IV. Conclusion

The results show that the syntax evaluation of all target tokens is considerably challenging for neural language models, even with sub-token vocabulary. Large language models are still under-performing human scores, contrary to previous results on a subset of target words, which are clearly higher. To possibly model all words or even made-up words is an excellent feature of recent neural nets. This gain of a quantitative vocabulary comes with a decrease in model quality. We can conclude that less common words with sub-tokens are badly represented in large language models and that there is a big room for improved syntax modeling.

The results show that models with improved syntax modeling are trained with an ambitious dataset. One approach to get such a dataset is to build a complex input and output relation, like in the instructions of FLAN-T5. Another approach is to use syntax-relevant data like in the models fine-tuned for error and grammar correction. Our type of evaluation that depends on the masked pre-training objective shows that such test data could in principle also be used for pre-training.

Future analysis could include xl-models like PaLM-FLAN-UL2-T5 [13] to test the current paradigm of ongoing size scaling and how the generative training objective alters the model. Furthermore, the grammatical framework [40] could be tested on CLAMS for one explicit grammar model, which can be qualitative perfect but lack the vocabulary quantity.

[12] The model card of PaLM-FLAN-T5 on huggingface with an interactive interface: https://huggingface.co/google/flan-t5-base

[13] The model card of PaLM-FLAN-UL2-T5 on huggingface with an interactive interface: https://huggingface.co/google/flan-ul2

References

[1] Y. Song, T. Wang, S. K. Mondal, and J. P. Sahoo, "A comprehensive survey of few-shot learning: Evolution, applications, challenges, and opportunities," 2022.

[2] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," 2022.

[3] H. W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, Y. Li, X. Wang, M. Dehghani, S. Brahma, A. Webson, S. S. Gu, Z. Dai, M. Suzgun, X. Chen, A. Chowdhery, A. Castro-Ros, M. Pellat, K. Robinson, D. Valter, S. Narang, G. Mishra, A. Yu, V. Zhao, Y. Huang, A. Dai, H. Yu, S. Petrov, E. H. Chi, J. Dean, J. Devlin, A. Roberts, D. Zhou, Q. V. Le, and J. Wei, "Scaling instruction-finetuned language models," 2022.

[4] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," 2019.

[5] A. Mueller, G. Nicolai, P. Petrou-Zeniou, N. Talmina, and T. Linzen, "Cross-linguistic syntactic evaluation of word prediction models," *CoRR*, vol. abs/2005.00187, 2020.

[6] T. Linzen, E. Dupoux, and Y. Goldberg, "Assessing the ability of LSTMs to learn syntax-sensitive dependencies," *Transactions of the Association for Computational Linguistics*, vol. 4, pp. 521–535, 2016.

[7] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," 2021.

[8] A. Parnami and M. Lee, "Learning from few examples: A summary of approaches to few-shot learning," 2022.

[9] X. Cao, W. Bu, S. Huang, Y. Tang, Y. Guo, Y. Chang, and I. W. Tsang, "A survey of learning on small data," 2022.

[10] A. R. Brian Lester and G. R. Noah Constant, Senior Staff Software Engineer, "Guiding frozen language models with learned soft prompts," 2022.

[11] S. Arora, A. Narayan, M. F. Chen, L. Orr, N. Guha, K. Bhatia, I. Chami, F. Sala, and C. Ré, "Ask me anything: A simple strategy for prompting language models," 2022.

[12] J. Jang, S. Ye, and M. Seo, "Can large language models truly understand prompts? a case study with negated prompts," 2022.

[13] F. Shi, M. Suzgun, M. Freitag, X. Wang, S. Srivats, S. Vosoughi, H. W. Chung, Y. Tay, S. Ruder, D. Zhou, D. Das, and J. Wei, "Language models are multilingual chain-of-thought reasoners," 2022.

[14] N. Ding, S. Hu, W. Zhao, Y. Chen, Z. Liu, H.-T. Zheng, and M. Sun, "Openprompt: An open-source framework for prompt-learning," *arXiv preprint arXiv:2111.01998*, 2021.

[15] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp, "Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Dublin, Ireland), pp. 8086–8098, Association for Computational Linguistics, May 2022.

[16] S. Narayan, Y. Zhao, J. Maynez, G. Simões, V. Nikolaev, and R. McDonald, "Planning with learned entity prompts for abstractive summarization," *Transactions of the Association for Computational Linguistics*, vol. 9, pp. 1475–1492, 2021.

[17] E. Ben-David, N. Oved, and R. Reichart, "PADA: Example-based Prompt Learning for on-the-fly Adaptation to Unseen Domains," *Transactions of the Association for Computational Linguistics*, vol. 10, pp. 414–433, 04 2022.

[18] S. Hu, N. Ding, H. Wang, Z. Liu, J. Wang, J. Li, W. Wu, and M. Sun, "Knowledgeable prompt-tuning: Incorporating knowledge into prompt verbalizer for text classification," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Dublin, Ireland), pp. 2225–2240, Association for Computational Linguistics, May 2022.

[19] J. Liu, A. Liu, X. Lu, S. Welleck, P. West, R. Le Bras, Y. Choi, and H. Hajishirzi, "Generated knowledge prompting for commonsense reasoning," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Dublin, Ireland), pp. 3154–3169, Association for Computational Linguistics, May 2022.

[20] Z. Wu, S. Wang, J. Gu, R. Hou, Y. Dong, V. G. V. Vydiswaran, and H. Ma, "Idpg: An instance-dependent prompt generation method," 2022.

[21] Y. Yu, R. Zhang, R. Xu, J. Zhang, J. Shen, and C. Zhang, "Cold-start data selection for few-shot language model fine-tuning: A prompt-based uncertainty propagation approach," 2022.

[22] C. Liao, Y. Zheng, and Z. Yang, "Zero-label prompt selection," 2022.

[23] T. Zhang, X. Wang, D. Zhou, D. Schuurmans, and J. E. Gonzalez, "Tempera: Test-time prompting via reinforcement learning," 2022.

[24] T. Khot, H. Trivedi, M. Finlayson, Y. Fu, K. Richardson, P. Clark, and A. Sabharwal, "Decomposed prompting: A modular approach for solving complex tasks," 2022.

[25] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. Le, and E. Chi, "Least-to-most prompting enables complex reasoning in large language models," 2022.

[26] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," 2022.

[27] T. Blevins, H. Gonen, and L. Zettlemoyer, "Prompting language models for linguistic structure," 2022.

[28] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2022.

[29] A. Srivastava, A. Rastogi, A. Rao, A. A. M. Shoeb, A. Abid, A. Fisch, A. R. Brown, A. Santoro, A. Gupta, A. Garriga-Alonso, *et al.*, "Beyond the imitation game: Quantifying and extrapolating the capabilities of language models," *arXiv preprint arXiv:2206.04615*, 2022.

[30] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, *et al.*, "Training verifiers to solve math word problems," *arXiv preprint arXiv:2110.14168*, 2021.

[31] S. Narang and A. Chowdhery, "Pathways language model (palm): Scaling to 540 billion parameters for breakthrough performance," 2022.

[32] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," *arXiv preprint arXiv:2201.11903*, 2022.

[33] OpenAI, "Introducing chatgpt." https://openai.com/blog/chatgpt 2021.

[34] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," 2022.

[35] K. Hilsenbek, "Reproducing targeted syntactic evaluations of language models," 2022.

[36] R. Marvin and T. Linzen, "Targeted syntactic evaluation of language models," 2018.

[37] C. Napoles, K. Sakaguchi, and J. Tetreault, "Jfleg: A fluency corpus and benchmark for grammatical error correction," 2017.

[38] Y. Tay, M. Dehghani, J. Rao, W. Fedus, S. Abnar, H. W. Chung, S. Narang, D. Yogatama, A. Vaswani, and D. Metzler, "Scale efficiently: Insights from pre-training and fine-tuning transformers," 2021.

[39] L. Xue, N. Constant, A. Roberts, M. Kale, R. Al-Rfou, A. Siddhant, A. Barua, and C. Raffel, "mt5: A massively multilingual pre-trained text-to-text transformer," 2020.

[40] A. Ranta, "Grammatical framework," *Journal of Functional Programming*, vol. 14, pp. 145 – 189, 2004.

	English	French	German
Across object rel. clause	0.65	0.54	0.52
Across subject rel. clause	0.67	0.63	0.52
Within object rel. clause	0.50	0.56	0.50
Short VP coordination	0.76	0.68	0.63
Long VP coordination	0.59	0.64	0.52
Simple agreement	0.67	0.59	0.55
Across prep. phrase	0.69	0.61	0.52
Average	0.65	0.61	0.54

Table 2: Evaluation of t5-small with an overall average of 0.60

	English	French	German
Across object rel. clause	0.50	0.52	0.50
Across subject rel. clause	0.50	0.64	0.50
Within object rel. clause	0.50	0.64	0.54
Short VP coordination	0.50	0.69	0.51
Long VP coordination	0.50	0.60	0.50
Simple agreement	0.50	0.72	0.50
Across prep. phrase	0.50	0.59	0.50
Average	0.50	0.63	0.51

Table 3: Evaluation of t5-base with an overall average of 0.55

	English	French	German
Across object rel. clause	0.50	0.52	0.51
Across subject rel. clause	0.50	0.59	0.54
Within object rel. clause	0.50	0.55	0.54
Short VP coordination	0.50	0.66	0.54
Long VP coordination	0.50	0.61	0.50
Simple agreement	0.50	0.53	0.49
Across prep. phrase	0.50	0.57	0.50
Average	0.50	0.58	0.52

Table 4: Evaluation of t5-large with an overall average of 0.53

	English	French	German
Across object rel. clause	0.60	0.50	0.54
Across subject rel. clause	0.70	0.54	0.58
Within object rel. clause	0.52	0.50	0.51
Short VP coordination	0.72	0.62	0.79
Long VP coordination	0.66	0.57	0.61
Simple agreement	0.62	0.59	0.59
Across prep. phrase	0.61	0.50	0.61
Average	0.63	0.55	0.60

Table 5: Evaluation of prithivida//grammar_error_correcter_v1 with an overall average of 0.59

	English	French	German
Across object rel. clause	0.53	0.56	0.50
Across subject rel. clause	0.55	0.76	0.51
Within object rel. clause	0.50	0.59	0.51
Short VP coordination	0.67	0.79	0.65
Long VP coordination	0.56	0.60	0.52
Simple agreement	0.56	0.82	0.48
Across prep. phrase	0.55	0.68	0.50
Average	0.56	0.69	0.52

Table 6: Evaluation of vennify/t5-base-grammar-correction with an overall average of 0.59

	English	French	German
Across object rel. clause	0.50	0.51	0.52
Across subject rel. clause	0.50	0.53	0.52
Within object rel. clause	0.50	0.52	0.51
Short VP coordination	0.50	0.56	0.51
Long VP coordination	0.50	0.55	0.55
Simple agreement	0.50	0.50	0.50
Across prep. phrase	0.50	0.51	0.51
Average	0.50	0.53	0.52

Table 7: Evaluating mt5-small with an overall average of 0.52

	English	French	German
Across object rel. clause	0.50	0.51	0.54
Across subject rel. clause	0.50	0.50	0.57
Within object rel. clause	0.51	0.58	0.52
Short VP coordination	0.50	0.51	0.51
Long VP coordination	0.50	0.58	0.56
Simple agreement	0.50	0.51	0.51
Across prep. phrase	0.50	0.50	0.49
Average	0.50	0.53	0.53

Table 8: Evaluation of mt5-base with an overall average of 0.52

	English	French	German
Across object rel. clause	0.50	0.51	0.58
Across subject rel. clause	0.50	0.54	0.64
Within object rel. clause	0.54	0.61	0.60
Short VP coordination	0.50	0.55	0.59
Long VP coordination	0.50	0.77	0.53
Simple agreement	0.50	0.51	0.55
Across prep. phrase	0.50	0.51	0.59
Average	0.51	0.57	0.58

Table 9: Evaluation of mt5-large with an overall average of 0.55

	English	French	German
Across object rel. clause	0.50	0.50	0.50
Across subject rel. clause	0.50	0.50	0.51
Within object rel. clause	0.50	0.50	0.50
Short VP coordination	0.50	0.54	0.54
Long VP coordination	0.50	0.58	0.50
Simple agreement	0.50	0.50	0.51
Across prep. phrase	0.50	0.50	0.51
Average	0.50	0.52	0.51

Table 10: Evaluation of t5-efficient-small with an overall average of 0.51

	English	French	German
Across object rel. clause	0.50	0.50	0.50
Across subject rel. clause	0.50	0.54	0.50
Within object rel. clause	0.50	0.58	0.50
Short VP coordination	0.50	0.58	0.52
Long VP coordination	0.50	0.56	0.50
Simple agreement	0.50	0.51	0.51
Across prep. phrase	0.50	0.51	0.50
Average	0.50	0.54	0.50

Table 11: Evaluation of t5-efficient-base with an overall average of 0.51

	English	French	German
Across object rel. clause	0.50	0.50	0.48
Across subject rel. clause	0.50	0.54	0.49
Within object rel. clause	0.50	0.59	0.50
Short VP coordination	0.50	0.59	0.52
Long VP coordination	0.50	0.61	0.50
Simple agreement	0.50	0.50	0.49
Across prep. phrase	0.50	0.50	0.50
Average	0.50	0.55	0.50

Table 12: Evaluation of t5-efficient-large with an overall average of 0.51

	English	French	German
Across object rel. clause	0.71	0.60	0.60
Across subject rel. clause	0.78	0.61	0.65
Within object rel. clause	0.50	0.51	0.49
Short VP coordination	0.75	0.66	0.67
Long VP coordination	0.61	0.55	0.55
Simple agreement	0.79	0.61	0.64
Across prep. phrase	0.70	0.61	0.63
Average	0.69	0.59	0.60

Table 13: Evaluation of flan-t5-small with an overall average of 0.63

	English	French	German
Across object rel. clause	0.50	0.50	0.50
Across subject rel. clause	0.50	0.50	0.50
Within object rel. clause	0.50	0.50	0.50
Short VP coordination	0.50	0.53	0.52
Long VP coordination	0.50	0.50	0.54
Simple agreement	0.50	0.51	0.51
Across prep. phrase	0.50	0.50	0.50
Average	0.50	0.51	0.51

Table 16: Evaluation of google/byt5-small with an overall average of 0.51

	English	French	German
Across subject rel. clause	0.50	0.48	0.48
Across subject rel. clause	0.50	0.52	0.48
Within object rel. clause	0.50	0.55	0.49
Short VP coordination	0.50	0.50	0.51
Long VP coordination	0.50	0.59	0.53
Simple agreement	0.50	0.44	0.47
Across prep. phrase	0.50	0.46	0.50
Average	0.50	0.51	0.49

Table 17: Evaluation of google/t5-v1_1-base with an overall average of 0.50

	English	French	German
Across object rel. clause	0.72	0.63	0.71
Across subject rel. clause	0.78	0.63	0.73
Within object rel. clause	0.51	0.54	0.50
Short VP coordination	0.79	0.68	0.77
Long VP coordination	0.74	0.63	0.67
Simple agreement	0.78	0.63	0.66
Across prep. phrase	0.76	0.62	0.68
Average	0.72	0.62	0.67

Table 14: Evaluation of flan-t5-base with an overall average of 0.67

	English	French	German
Across object rel. clause	0.76	0.56	0.73
Across subject rel. clause	0.80	0.67	0.77
Within object rel. clause	0.53	0.52	0.50
Short VP coordination	0.79	0.69	0.75
Long VP coordination	0.65	0.76	0.64
Simple agreement	0.79	0.59	0.58
Across prep. phrase	0.76	0.57	0.65
Average	0.73	0.62	0.66

Table 15: Evaluation of flan-t5-large with an overall average of 0.67

Appendix 2: The annotated PaLM

Kalle Hilsenbek

***Abstract*— The annotated PaLM is based on PaLM-pytorch from Phil Wan [1] and the report style is inspired by the annotated transformer[2]. A compact implementation of the closed-sourced PaLM is provided with documentation after a brief summary of the elements.**

***Index Terms*—Research, PaLM, Scaling Language Modeling with Pathways, Annotation, Transformer.**

ORIGINAL PALM ABSTRACT

Large language models have been shown to achieve remarkable performance across a variety of natural language tasks using few-shot learning, which drastically reduces the number of task-specific training examples needed to adapt the model to a particular application. Up to 540-billion parameter models are trained to study the impact of scaling a on few-shot learning with a densely activated, Transformer language model, which is called Pathways Language Model (PaLM). The original PaLM is pretrained on 6144 TPU v4 chips using Pathways, a new ML system which enables highly efficient training across multiple TPU Pods. Whereas the reimplementation is based on pytorch and deepspeed. The continued benefits of scaling are achieved by state-of- the-art few-shot learning results on hundreds of language understanding and generation benchmarks. On a number of these tasks, PaLM 540B achieves breakthrough performance, outperforming the finetuned state- of-the-art on a suite of multi-step reasoning tasks, and outperforming average human performance on the recently released BIG-bench benchmark. A significant number of BIG-bench tasks showed discontinuous improvements from model scale, meaning that performance steeply increased by scaling to the largest model. PaLM also has strong capabilities in multilingual tasks and source code generation, which are demonstrated on a wide array of benchmarks. There are comprehensive analysis and studies provied on bias, toxicity, the extent of training data memorization with respect to the model scale. The original PalM paper also discusses the ethical considerations related to large language models and potential mitigation strategies.

I. INTRODUCTION

This annotated PaLM [1] implementation focuses on the model architecture and its python code under 300 lines. While there is a short summary of every component in this continuous text, the main documentation is in the code itself. The key takeaways of the PaLM paper are as follows:

- Efficient scaling: Pathways can efficiently train a language model across thousands of accelerator chips.
- Breakthrough capabilities with chain-of-thought prompting
- (Dis)continuous improvements from scaling with emerging capabilities
- Multilingual understanding: The English centric model (with 78 percent English trainings data) has few-shot capabilities in multilingual settings.
- Bias and toxicity: Firstly, for gender and occupation bias, the PaLM authors found that accuracy on the Winogender coreference task improves with model scale, and PaLM 540B sets a new state-of-the-art result in 1-shot and few-shot settings. Secondly, co-occurence analysis performed on race/religion/-gender prompt continuation demonstrates the potential for the model to falsely affirm stereotypes, for instance, associating Muslims with terrorism, extremism, and violence. However, the toxicity of the model-generated continuation correlates highly with the toxicity of the prompting text, whereas human-generation continuations do not have a strong toxicity correlation.

II. EINSTEIN ANNOTATION

The Einstein notation, also known as the Einstein summation convention or Einstein summation notation, is a convention used in mathematics and physics to simplify expressions that involve summation over indexed terms. It was introduced by Albert Einstein in his work "The Foundation of the General Theory of Relativity" and has since become a standard notation in many areas of physics and mathematics.

There is a python function called "einsum" to multiply, sum and transpose with this annotation in numpy and pytorch. The key is to choose the correct labeling for the axes of the input arrays and the output array. For example, the string 'ij,jk->ik' specifies a matrix multiplication operation between two 2D arrays A and B. The letters 'ij' and 'jk' label the axes of the input arrays A and B, respectively. The letter 'i' corresponds to the rows of A, and the letter 'j' corresponds to the columns of A and the rows of B. The letter 'k' corresponds to the columns of B. The arrow '->' separates the input and output labels. The output labels 'ik' correspond to the rows of A and the columns of B, respectively (see figure 1 on the next page).

[1]The repo is located at https://github.com/lucidrains/PaLM-pytorch with a personal page of Phil Wang at https://lucidrains.github.io/

[2]The web-version of the annotated transformer is located at http://nlp.seas.harvard.edu/annotated-transformer/

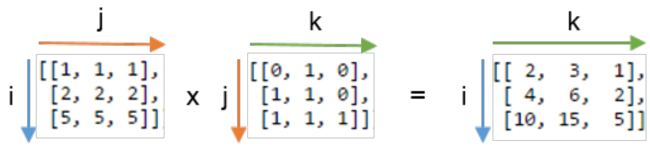

Figure 1: from the article "A basic introduction to NumPy's einsum" by Alex Riley [2]. newline Drawing on the labels, our matrix multiplication with np.einsum('ij,jk->ik', A, B)

So, the einsum function is a very useful tool for performing complex tensor operations with ease, flexibility and efficiency. It allows for a wide range of tensor manipulations to be expressed in a concise manner, which can greatly simplify the coding of complex numerical algorithms like PaLM.

III. The Annotated Model Architecture

PaLM uses a standard Transformer-decoder with a few modifications. This is surprising given the amount of different efficient transformer architectures.

A. Requirements

The requirements are einops>=0.4 torch>=1.6 triton>=2.0dev PaLM-pytorch triton-transformer and deepspeed [3].

B. Preprocessing

SentencePiece is used as vocabulary with 256k tokens, which was chosen to support a large number of languages in the training corpus without excess tokenization. The vocabulary was generated from the training data and is completely lossless and reversible, which means that whitespace is completely preserved in the vocabulary (especially important for code) and out-of-vocabulary Unicode characters are split into UTF-8 bytes, with a vocabulary token for each byte.

C. Import and defintion

After the import of torch and einops with its matrix rearrange function, the complete transformer-decoder is defined as a function (see code figure 8).

D. Parallel attention

Parallel layers (see code figures 9 and 10) like the GPT-J-6B [4] – rather than the standard "serialized" formulation. Specifically, the standard formulation can be written as:

y = x + MLP(LayerNorm(x + Attention(LayerNorm(x)))

Whereas the parallel formulation can be written as:

y = x + MLP(LayerNorm(x)) + Attention(LayerNorm(x))

The parallel formulation results in roughly 15 percent faster training speed at large scales, since the MLP and Attention input matrix multiplications can be fused. Ablation experiments showed a small quality degradation at 8B scale but no quality degradation at 62B scale. The effect of parallel layers should be quality neutral at the 540B scale by extrapolation of the results to the biggest model.

The standard Transformer formulation uses k attention heads, where the input vector for each timestep is linearly projected into "query", "key", and "value" tensors of shape [k, h], where h is the attention head size. Here the Multi-Query Attention [5] is used and the key/value projections are shared for each head, i.e. "key" and "value" are projected to [1, h], but "query" is still projected to shape [k, h]. This has a neutral effect on model quality and training speed, but results in significant cost savings at autoregressive decoding time. This is because standard multi-headed attention has low efficiency on accelerator hardware during auto-regressive decoding, because the key/value tensors are not shared between examples, and only a single token is decoded at a time.

E. Swish activation

The SwiGLU activation [6] (see code figure 11) is used for the multi-layer-perceptron (MLP) with the formula of a modified sigmoid-function displayed in figure 2. Swish can be displayed as a smooth interpolation function between the linear function and the ReLU function (see figure 3).

$$\mathrm{SwiGLU}(x, W, V, b, c, \beta) = \mathrm{Swish}_\beta(xW + b) \otimes (xV + c)$$

Figure 2: The Swish activation formular, where Swish is Swish(x) = x · sigmoid(ßx)

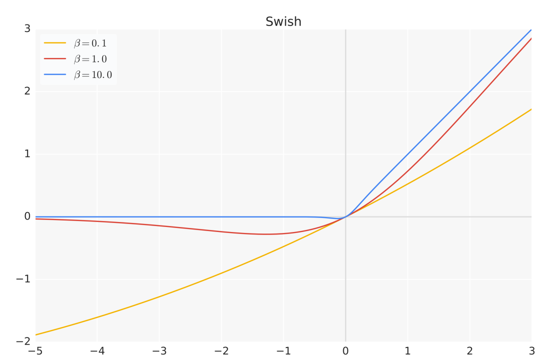

Figure 3: The Swish activation function with different beta values.

F. Rotary Embeddings

Rotary Position Embedding(RoPE) from the RoFormer [7], rather than absolute or relative position embeddings, since RoPE embeddings have been shown to have better performance with efficient attention as well as long sequence lengths.

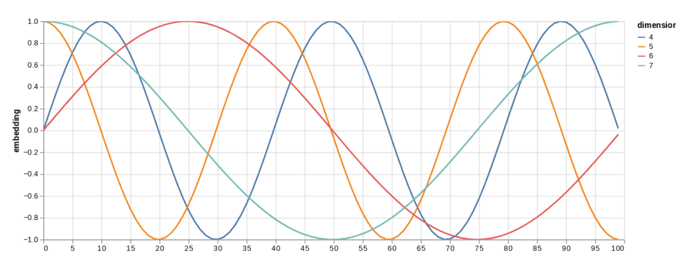

Figure 4: The original sinusoidal embeddings displayed in the lower dimensions [8].

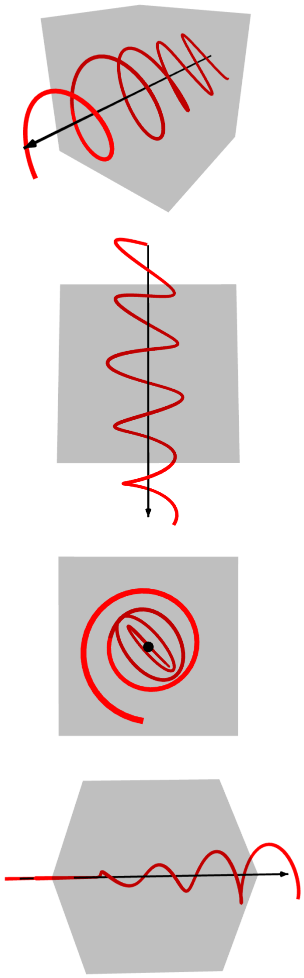

Figure 5: The rotary embedding adds a rotation to this wave oscillation, which is visualized from different angles in 3D with a quarter-waveplate that changes the polarization [9].

The original sinusoidal embeddings calculate the relative position and the ends of the encoder and decoder-block by calculating the sinus function of the dimension with the absolute position (see figure 4). This allows to preserve of the relative position informations, because the dot products of the attention do not preserve absolute positional information.

RoPE encodes the absolute position with a rotation matrix (see figure 5 and code figure 12) and incorporates the explicit relative position dependency in self-attention formulation. It enables valuable properties, such as flexibility of sequence length and decaying inter-token dependency with increasing relative distances.

G. Residual connections

Residual connections are a simple and effective technique to train deep neural networks by adding the input tensor of a layer to the calculated output tensor. It is widely adopted by different models and was introduced by ResNet in computer vision [10]. Residual connections like in the original transformer decoder are used in PaLM (see code figure 13).

H. Layer normalization

Batch normalization is a technique used to reduce the training time and stabilize the hidden state dynamics of deep neural networks. Batch normalization can be transposed into layer normalization [11] by computing the mean and variance used for normalization from all of the summed inputs to the neurons in a layer on a single training case (see figure 6 and code figure 14).

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Figure 6: Layer Normalization over a mini-batch of inputs as described in the paper Layer Normalization [11].

I. Deepspeed training

After the definition of all functions and classes, the PaLM can be initialised and can be trained with deepspeed (see figure 15). The training setup in this implementation with pytorch differs from the setup with original tensorflow and is not covered. Deepspeed is a common open-source tool to train very large language models in the past (see figure 7).

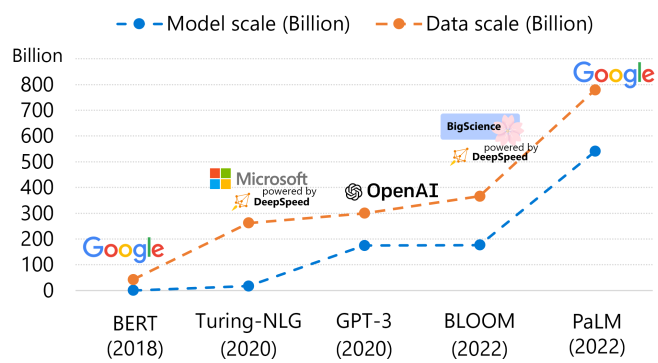

Figure 7: Model scale (number of parameters) and data scale (number of tokens consumed during training) of representative language models in the last years [3].

IV. Discussion

There is an ongoing discussion in the machine-learning community, whether it is preferable to use a decoder-only architecture like GPT or a full encoder-decoder like the vanilla transformer and T5. For more details have a look at: https://github.com/lucidrains/PaLM-rlhf-pytorch/issues/6

References

[1] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fe-

dus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," 2022.

[2] A. Riley, "A basic introduction to numpy's einsum." https://ajcr.net/Basic-guide-to-einsum/

[3] C. Li, Z. Yao, X. Wu, M. Zhang, C. Holmes, C. Li, and Y. He, "Deepspeed data efficiency: Improving deep learning model quality and training efficiency via efficient data sampling and routing," 2022.

[4] B. Wang and A. Komatsuzaki, "GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model." https://github.com/kingoflolz/mesh-transformer-jax May 2021.

[5] N. Shazeer, "Fast transformer decoding: One write-head is all you need," 2019.

[6] N. Shazeer, "Glu variants improve transformer," 2020.

[7] J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu, "Roformer: Enhanced transformer with rotary position embedding," 2021.

[8] A. Rush, "The annotated transformer," in *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, (Melbourne, Australia), pp. 52–60, Association for Computational Linguistics, July 2018.

[9] S. Biderman, S. Black, C. Foster, L. Gao, E. Hallahan, H. He, B. Wang, and P. Wang, "Rotary embeddings: A relative revolution."

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[11] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," 2016.

Figure 8: Import of external functions and the definition of the PaLM creation.

```python
import torch
import torch.nn.functional as F
from einops import rearrange
from torch import einsum, nn
# einsum is a compact function to efficiently operate with matrix operations

# definition of the complete transformer-decoder as a function
def PaLM(*,dimensions=512, sentencepiece_tokens=16000, depth=4, head_dimensions=64, heads=8, ff_mult=4):
    """
    It is a small model for testing.
    The model and head dimensions, as well as the number of tokens and heads are similar to the base transformer translation model.

    The input and output embedding matrices are shared, which is done frequently (but not universally) in past work.
    https://arxiv.org/pdf/1608.05859.pdf

    Parameters:
          dimensions(int): The dimensions of the transformer model
          sentencepiece_tokens(int): The size of the vocabulary
          depth(int): The number of stacked layers
          head_dimensions(int): The dimensions of the attention heads
          heads(int): The number of the attention heads
          ff_mult(int): The multiplication factor to calculate the hidden dimensions
    Return:
          net(torch.nn.Sequential): The initialized transformer-decoder
    """
    # Stacking sequentially the transformer-decoder:
    # First the embedding layer
    # Multiple times a combined residual ParallelTransformerBlock
    # In the end a layer normalization and a linear layer without bias
    net = nn.Sequential(
        nn.Embedding(sentencepiece_tokens, dimensions),
        *[
            Residual(ParallelTransformerBlock(dim=dimensions, dim_head=head_dimensions, heads=heads, ff_mult=ff_mult))
            for _ in range(depth)
        ],
        LayerNorm(dimensions),
        nn.Linear(dimensions, sentencepiece_tokens, bias=False)
    )

    # they used embedding weight tied projection out to logits, not common, but works
    net[-1].weight = net[0].weight

    # initialize the weights with the normal distribution and a standard deviation of 0.02
    nn.init.normal_(net[0].weight, std=0.02)
    return net
```

Figure 9: The ParallelTransformerBlock class

```
class ParallelTransformerBlock(nn.Module):
    """
    This is the class for the ParallelTransformerBlock.
    It can parallelize the attention and MLP with a decentralized training algorithm.

    The distributed tensor computation and model parallelism was introduced in Megatron-LM:
    https://arxiv.org/pdf/1909.08053.pdf
    Megatron-LM uses parallel attention layers and parallel multi-layer perceptron
    and fuses the general matrix-matrix multiplication (GEMM) with the cross-entropy loss,
    whereas PaLM fuses the MLP and attention input matrix.
    """
    def __init__(self, dimensions, dim_head=64, heads=8, feedforward_mult=4):
        """
        Initialization of the ParallelTransformerBlock
        Parameters:
            dimensions(int): The dimensions of the transformer model
            dim_head(int): The dimensions of the attention heads
            heads(int): The number of the attention heads
            feedforward_mult(int): The multiplication factor to calculate the hidden dimensions
        """
        super().__init__()
        self.norm = LayerNorm(dimensions)

        # calcute the inner dimensions and fuse them with the attention dimensions
        attn_inner_dim = dim_head * heads
        ff_inner_dim = dimensions * feedforward_mult
        self.fused_dims = (attn_inner_dim, dim_head, dim_head, (ff_inner_dim * 2))

        self.heads = heads
        # save the scaling for the query tensor with 1/sqrt(dim_head)
        self.scale = dim_head**-0.5
        self.rotary_emb = RotaryEmbedding(dim_head)

        # initialize the feedforward projection of the fused attention and MLP tensors
        self.fused_attn_ff_proj = nn.Linear(dimensions, sum(self.fused_dims), bias=False)
        # initialize the aggregation of the attention output
        self.attn_out = nn.Linear(attn_inner_dim, dim, bias=False)

        # SwiGLU-activation chained with a linear transformation from the inner to the output dimensions
        self.feedforward_out = nn.Sequential(
            SwiGLU(),
            nn.Linear(ff_inner_dim, dimensions, bias=False)
        )

        # for caching causal mask and rotary embeddings
        self.register_buffer("mask", None, persistent=False)
        self.register_buffer("pos_emb", None, persistent=False)

    def get_mask(self, sequence_length, device):
        """
        Get the mask matrix or create it, if there is not a proper one.
        Parameters:
            sequence_length(int): The length of the input tensor
            device (torch.device): The device on which the tensor is allocated
        Returns:
            mask_output_tensor(torch.Tensor): The mask tensor
        """
        # if there exists a large enough mask matrix,
        # then return this mask matrix as big as the sequence length
        if self.mask is not None and self.mask.shape[-1] >= sequence_length:
            return self.mask[:sequence_length, :sequence_length]

        # create a mask matrix with ones in the upper triangular part of a matrix
        mask = torch.ones((sequence_length, sequence_length), device=device, dtype=torch.bool).triu(1)
        self.register_buffer("mask", mask, persistent=False)
        return mask
```

Figure 10: The second half of the ParallelTransformerBlock class

```python

    def get_rotary_embedding(self, sequence_length, device):
        """
        Get an object of the RotaryEmbedding class or create it, if there is not a proper one.
        Parameters:
            sequence_length(int): The length of the input tensor
            device (torch.device): The device on which the tensor is allocated
        Returns:
            RotaryEmbedding(nn.Module): An object of the RotaryEmbedding class
        """
        # if there exists a large enough embedding matrix,
        # then return this matrix as big as the sequence length
        if self.pos_emb is not None and self.pos_emb.shape[-2] >= sequence_length:
            return self.pos_emb[:sequence_length]

        # create a new embedding object
        pos_emb = self.rotary_emb(sequence_length, device=device)
        self.register_buffer("pos_emb", pos_emb, persistent=False)
        return pos_emb

    def forward(self, input_tensor):
        """
        The calculation of the parallel attention mechanism.
        This einstein summation simplifies the notation:
            b - batch
            h - heads
            n, i, j - sequence length (base sequence length, source, target)
            d - feature dimension

        Parameters:
            input_tensor(torch.Tensor):
                An input tensor consiting of queries, keys, values and inner_feedforward tensors.
        Returns:
            out(torch.Tensor): The aggregated output tensor
        """
        sequence_length, device = input_tensor.shape[1], input_tensor.device

        # pre layernorm
        input_tensor = self.norm(input_tensor)

        # attention queries, keys, values, and feedforward inner
        queries, keys, values, inner_feedforward = self.fused_attn_ff_proj(input_tensor).split(self.fused_dims, dim=-1)

        # split the queries by the heads
        # they use multi-query single-key-value attention, yet another Noam Shazeer paper
        # they found no performance loss past a certain scale, and more efficient decoding obviously
        # https://arxiv.org/abs/1911.02150
        queries = rearrange(queries, "b n (h d) -> b h n d", h=self.heads)

        # rotary embeddings
        positions = self.get_rotary_embedding(sequence_length, device)
        queries, keys = map(lambda tensor: apply_rotary_pos_emb(positions, tensor), (queries, keys))

        # scale
        queries = queries * self.scale

        # calcute the similarity between the query and key matrix
        similarity = einsum("b h i d, b j d -> b h i j", queries, keys)

        # causal masking
        causal_mask = self.get_mask(sequence_length, device)
        similarity = similarity.masked_fill(causal_mask, -torch.finfo(sim.dtype).max)

        attention = similarity.softmax(dim=-1)

        # aggregate values
        out = einsum("b h i j, b j d -> b h i d", attention, values)

        # merge heads
        out = rearrange(out, "b h n d -> b n (h d)")
        return self.attn_out(out) + self.feedforward_out(inner_feedforward)
```

Figure 11: The SwiGLU class

```
class SwiGLU(nn.Module):
    """
    They use SwiGLU instead of the more popular GEGLU for gating the feedforward.
    """
    def forward(self, input_tensor):
        """
        The input tensor is split into two halves at the last dimension.
        A product is returned, consisting of the first half as a factor
        and the Sigmoid Linear Unit (SiLU) function with the second half.
        Parameters:
            input_tensor(torch.Tensor):
                The tensor is split in half and passed into the SiLU function.
        Returns:
            SwiGLU_output_tensor(torch.Tensor): The output tensor of the SwiGLU function.
        """
        tensor_chunk, gate_tensor = input_tensor.chunk(2, dim=-1)
        return F.silu(gate_tensor) * tensor_chunk
```

Figure 12: The RotaryEmbedding class with helper functions

```python
class RotaryEmbedding(nn.Module):
    """
    The rotary embedding takes the dimension of the embedding as input and calculates the predefined parameters    as the inverse frequency.
    This inverse frequency is used to calculate the position embeddings, which are then rotated by adding a rotation matrix to the sinusoidal wave oscillation.
    """
    def __init__(self, dimemsions):
        """
        Initializes the rotary embedding with the inverse frequency.
        Parameters:
            dimensions (int): The dimensions of the transformer model
        """
        super().__init__()
        inverse_frequency = 1.0 / (10000 ** (torch.arange(0, dimemsions, 2).float() / dimemsions))
        self.register_buffer("inverse_frequency", inverse_frequency)

    def forward(self, max_seq_len, *, device):
        """
        This forward function creates a flat sequence tensor with the maximum sequence length.
        And return the product of the maximum sequence and the inverse frequency.
        Parameters:
            max_seq_len (int): The maximum sequence length
            device (torch.device): The device on which the tensor is allocated
        Returns:
            frequency_cat(torch.Tensor): Concatenation of two frequency tensors on the last dimension
        """
        sequence = torch.arange(max_seq_len, device=device, dtype=self.inverse_frequency.dtype)
        frequency = einsum("i , j -> i j", sequence, self.inverse_frequency)
        return torch.cat((frequency, frequency), dim=-1)

def rotate_half(input_tensor):
    """
    This function rearranges the input tensor to cut it in half and then to concatenate the negative second half and the first half.
    Parameters:
        input_tensor(torch.Tensor): The tensor that is cut in half and rotated
    Returns:
        tensor_concatenation(torch.Tensor): A concatenation of the negative second half and the first half
    """
    rearranged_input_tensor = rearrange(input_tensor, "... (j d) -> ... j d", j=2)
    # cut the tensor in half on the nested, second last dimemsion
    first_half, second_half = rearranged_input_tensor.unbind(dim=-2)
    # concatenate the two halves
    return torch.cat((-second_half, first_half), dim=-1)
```

Figure 13: The residual class

```python
class Residual(nn.Module):
    """
    Residual connections are a type of shortcut connection that allow signal to bypass certain layers of a model.
    This helps to reduce the vanishing gradient problem and helps to optimize the model.
    In a transformer, a residual connection is added between the multi-head attention block and the feed-forward neural network.
    This allows for the signal to pass through the decoder layers without being significantly altered, improving the flow of information through the model and the accuracy of the output.
    By bypassing certain layers, the model can more accurately capture and reproduce the input signal.
    """
    def __init__(self, parallel_transformer):
        """
        Initializes the residual connection with a ParallelTransformerBlock
        Parameters:
            parallel_transformer (nn.Module): A ParallelTransformerBlock object
        """
        super().__init__()
        self.parallel_transformer = parallel_transformer

    def forward(self, input_tensor):
        """
        This forward function processes the input tensor with one ParallelTransformerBlock and adds the unchanged input tensor.
        Parameters:
            input_tensor (torch.Tensor): The input tensor for the ParallelTransformerBlock
        Returns:
            residual_addition (torch.Tensor): Addition of the ParallelTransformerBlock and the unchanged input tensor.
        """
        return self.parallel_transformer(input_tensor) + input_tensor
```

Figure 14: The layer normalization class

```python
class LayerNorm(nn.Module):
    """
    This class implements Layer Normalization as described in the layer norm paper by Jimmy Lei Ba, Jamie Ryan Kiros and Geoffrey E. Hinton.
    It applies a normalization layer to the input tensor without using a bias to increase training stability.
    The LayerNorm object is initialized with no bias and ones as multiplication factors.
    The forward method takes in an unnormed tensor and returns a normalized tensor using the initialized gamma and beta attributes.
    """
    def __init__(self, dimensions):
        """
        Initializes the LayerNorm object with no bias and ones as multiplication factors.
        Parameters:
            dimensions (int): The dimensions of the transformer model
        """
        super().__init__()
        self.gamma = nn.Parameter(torch.ones(dimensions))
        self.register_buffer("beta", torch.zeros(dimensions))

    def forward(self, unnormed_tensor):
        """
        Calculate the normalization without a bias.
        Parameters:
            self (nn.Module): A ParallelTransformerBlock object as context
            unnormed_tensor (torch.Tensor): The input tensor that is going to be normalized
        Returns:
            normalized_tensor (torch.Tensor): The normalized tensor processed by the layer norm with the initialized gamma and beta attributes.
        """
        return F.layer_norm(unnormed_tensor, unnormed_tensor.shape[-1:], self.gamma, self.beta)
```

Figure 15: The definition of PaLM and deepspeed training

```
# The PaLM 540B from the paper
# with a batch sizes from 512     1024     2048
palm = PaLM(
    number_sentencepiece_tokens = 256000,
    dimensions = 18432,
    depth = 118,
    heads = 48,
    head_dimensions = 256
)

!deepspeed train.py --deepspeed --deepspeed_config ds_config.json
```