Controlled tasks for model analysis: Retrieving discrete information from sequences

Ionut-Teodor Sorodoc[*] **Gemma Boleda**[*] [†] **Marco Baroni**[*] [†]

[*]Universitat Pompeu Fabra
[†]ICREA
Barcelona, Spain
`{firstname.lastname}@upf.edu`

Abstract

In recent years, the NLP community has shown increasing interest in analysing how deep learning models work. Given that large models trained on complex tasks are difficult to inspect, some of this work has focused on controlled tasks that emulate specific aspects of language. We propose a new set of such controlled tasks to explore a crucial aspect of natural language processing that has not received enough attention: the need to retrieve discrete information from sequences.

We also study model behavior on the tasks with simple instantiations of Transformers and LSTMs. Our results highlight the beneficial role of decoder attention and its sometimes unexpected interaction with other components. Moreover, we show that, for most of the tasks, these simple models still show significant difficulties. We hope that the community will take up the analysis possibilities that our tasks afford, and that a clearer understanding of model behavior on the tasks will lead to better and more transparent models.

1 Introduction

There has been a continuous increase in performance in computational linguistics in recent years. This development correlates with larger and more complex models which are trained on ever bigger datasets. These new characteristics of modelling made it harder to understand which model components are relevant and how they work. This has motivated the NLP community to develop new methods for model analysis.

Given the difficulties analyzing large models, some of this work (Hupkes et al., 2018; Lake and Baroni, 2018; Hewitt and Liang, 2019; Chrupała and Alishahi, 2019; White and Cotterell, 2021) has focused on controlled tasks that emulate specific aspects of language. We propose a new set of such controlled tasks to explore a crucial aspect of natural language processing that has not received enough attention: the need to retrieve discrete information from sequences. [1] Retrieving discrete information from sequences is necessary for natural language processing for instance to track agreement between different constituents of a sentence and to establish coreference relationships (among many other examples). The controlled tasks are built using binary vectors as input in order to exclude the possible inferences of linguistic patterns that are not the focus of our analysis. In particular, we design the tasks such that the models need to emulate four abilities that are crucial for natural language: incremental processing, indirect mappings, contextualization, and order tracking.

We study model behavior on the proposed tasks with simple instantiations of Transformers (Vaswani et al., 2017) and LSTMs (Hochreiter and Schmidhuber, 1997), and show that, for most of the tasks, these models show significant difficulties. Transformers are the state of the art in deep learning for NLP, and LSTMs are a classical architecture that was designed for sequence processing. In the analysis, we aim at understanding the role of the different components of the models, focusing on self-attention and positional encoding for Transformers and decoder attention for LSTMs.

In the Transformer, self-attention turns out to only have a clear beneficial role when the task requires taking token order into account, with decoder attention performing most of the work in other cases. Positional encoding plays a counterintuitive role, helping in tasks that do not involve order by providing beneficial noise, and harming performance when it is redundant with self-attention. The LSTM only compares favorably to the Transformer in short sequence lengths for simple tasks, and decoder attention is highly beneficial for this model.

[1]The associated dataset and code can be downloaded at: https://github.com/sorodoc/DiscreteSeq

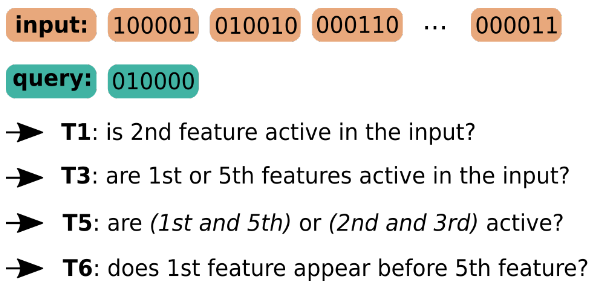
input: 100001 010010 000110 ··· 000011
query: 010000
T1: is 2nd feature active in the input?
T3: are 1st or 5th features active in the input?
T5: are (1st and 5th) or (2nd and 3rd) active?
T6: does 1st feature appear before 5th feature?

Figure 1: Schematic description of our tasks. Different tasks require different interpretations of the query. The answer is positive for all these instances.

2 Related Work

The current study belongs in the newly developed area of interpretability and explainability of computational linguistics, which seeks to understand how models capture natural language phenomena (Alishahi et al., 2019; Belinkov and Glass, 2019; Rogers et al., 2021).

Much of this literature has focused on model behaviour in complex NLP tasks (Linzen et al., 2016; Conneau et al., 2018; Voita et al., 2019; Abnar and Zuidema, 2020; Sinha et al., 2021; Lakretz et al., 2021). Some other work has used controlled tasks to analyze neural networks regarding specific aspects, such as reasoning skills (Weston et al., 2015), compositionality (Lake and Baroni, 2018; Hupkes et al., 2020), PoS tagging (Hewitt and Liang, 2019), inductive biases (White and Cotterell, 2021) or hierarchical structure (Hupkes et al., 2018; Chrupała and Alishahi, 2019). We follow this latter methodology, and design tasks that highlight one of the core abilities that a model needs to possess in order to process natural language, namely detecting one or more features in a sequence of tokens, possibly in a context- and/or order-sensitive way.

3 Description of the tasks

3.1 Task description

In all tasks, the goal is to perform *feature detection*, that is, to provide a binary response about whether a feature (or feature combination) is present in the input sequence. The tasks are schematically illustrated in Figure 1. The model is always presented with an *input* consisting in a sequence of *tokens*. Each token is a 36-dimensional binary vector with a variable number of dimensions (*features*) activated (set to 1). The model is also given a *query*, which is a single 36-dimensional binary vector with 1 feature activated, except for Task 2 below, where it has 2 activated features.

The tasks are meant to be *incremental*, in the sense that, when processing the input sequence, the model does not know what information it will need to retrieve from it. Some uses of Transformers in the NLP literature instead give access to all the information from the start, such as in BERT (Devlin et al., 2019) and its variants. Arguably, incrementality is necessary in most instances of language understanding "in the wild": For some applications, like Machine Translation for documents, it is realistic to give access to all the input at once, but as we move towards real-time applications such as virtual assistants, models will need to act incrementally.

T1: one-feature detection. T1 asks whether the active feature of the query is present in some token in the sequence. A linguistic example, relevant for syntactic processing, would be: did the plural feature occur in a span of tokens?

In the example in Figure 1, the second input token has feature 2, which is the one the query asks about, so the answer should be positive.

T2: two-feature detection. T2 asks whether two features occur in the sequence, be it in the same or in different tokens. This ability is required, for example, to check agreement between two syntactic units, or to answer a conjoint question.

In T1 and T2, query features and input features coincide: if the query asks about feature 2, then feature 2 needs to be active in some token for the answer to be positive. In the rest of the tasks, the relationship between query and input features is instead indirect. For instance, as illustrated in Figure 1 for T3, a query with feature 2 may ask about features 1 and 5 in the input. Models need to learn this implicit mapping when they are trained. Translation is an example (among many) of an indirect linguistic task, where words in the source language map to different target-language words.

T3: set member detection. In T3, models need to detect whether at least one out of two features is present in the sequence, that is, the task checks for feature disjunction (see Figure 1). This is akin to set member detection because we ask for a positive answer when at least one of the two features (set members) is present.

For example, many question-answering setups require retrieving an *instance* (*Fido, Snoopy, . . .*) when prompted with the name of the *class* (*dog*).

T4: contextualized 2-feature detection. T4 asks about a conjunction of two features, like T2 (but with the indirect mapping). A linguistic example could involve a question requiring more than one piece of information to answer (*Who played the 1982 World Cup Final? Italy and Germany*). In addition, we design the mapping such that each feature is part of 2 queries; hence, it is associated only with two other features. This dataset structure should encourage **contextualization**, because a given feature is only ever relevant in the context of one of the other two features it is associated with.

Contextualization is crucial for natural language, where the same feature (e.g., a word) requires different responses depending on other features occurring in the context. A natural hypothesis is that the Transformer is naturally good for contextualization (this should be the strength of self-attention), but as we will see our results are mixed.

T5: contextualized set member detection. Like T3, the task concerns disjunction with indirect mapping, but it consists in checking whether at least one out of two *pairs* of features is present in the input (e.g., $(1 \wedge 5) \vee (2 \wedge 7)$; see Figure 1). Again, in each pair, one feature is only a hit in the context of the other, so this task also requires contextualization.

A linguistic example would be a coreference task where *toy* might refer to *rubber lion* or *plastic truck*, but not to *lion* or *truck* in other contexts.

T6: ordered feature detection. T6 adds order to T4: it asks about a conjunction of two features with the indirect mapping, where the features must be in a specific order. This has many linguistic counterparts, e.g., parsing words in the correct order to determine their syntactic relation.

T7: contextualized ordered set member detection. Finally, T7 adds order to T5: It asks models to check whether at least one out of two pairs of features is present in the input, but now order of context matters (e.g., *(1 before 5)* $\vee$ *(2 before 7)*). An example would be semantic role identification, where *cow* is typically an *agent* if it precedes *eats*, but not if it follows it.

3.2 Dataset construction

All datasets contain 100k training, 10k validation, and 10k test examples. For each datapoint, we sample uniformly at random whether the answer is positive or negative. For each task, we create datasets with sequence lengths 5, 10, 15, 20, 25, and 30. Also, for each datapoint, we exclude between 1 and 6 randomly chosen attributes. We apply this restriction because otherwise, for longer sequences, nearly all positive datapoints would contain all features, whereas negative datapoints must, by construction, miss at least one feature. This would allow models to develop a degenerate guessing strategy ("datapoint is positive if it contains all features").

	Nr. of distinct queries	Nr. of queries containing each feature
T1	36	1
T2	630	36
T3, T4, T6	36	2
T5, T7	36	4

Table 1: Description of the query space.

T1. For each datapoint, we choose a random feature as a query. If the answer is positive, the feature will appear in the sequence. All other features are randomly set to active with $p = 0.2$. During training, the queried feature can appear multiple times (following the 0.2 probability), but it appears only once at test time for 2 main reasons: to facilitate analysis and to avoid having the number of appearances of the queried feature in the sequence as an informative variable.

T2. The 2 queried features are randomly chosen. This is the only task where we have 2 query activations, instead of 1. As is highlighted in Table 1, the query space is much larger for this task, in comparison with all the other tasks (630 possible queries, vs. 36 possible queries for the others). For positive datapoints, we randomly choose whether the features are in the same token. For negative datapoints, we randomly choose whether one of the features appears in the sequence.

For T3-7, we use predefined mappings between query and input features (these mappings will not be accessible to the models, but will need to be learned as part of carrying out the task). The input feature pairs in these tasks consist of a feature between 1 and 18 and one between 19 and 36. Also, we aim for an equal representation of features in the query space, so each feature appears twice in T3, T4 and T6 and four times in T5 and T7 in the input-query mapping (see Table 1).

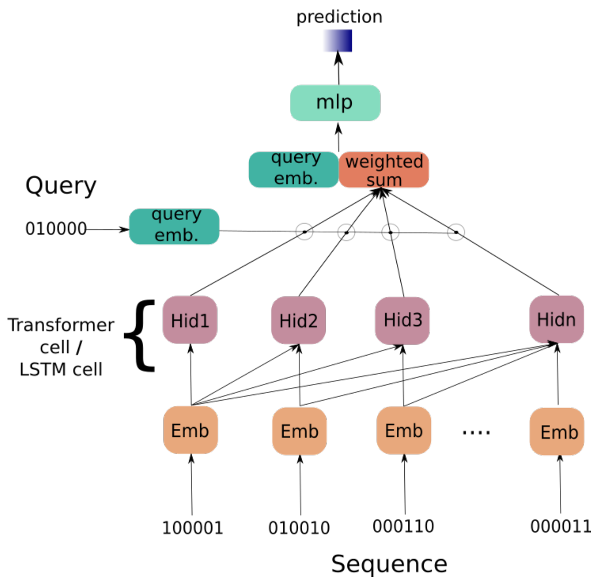

Figure 2: Diagram of the models.

T3. First, we choose a random feature to activate in the query. We then check which are the 2 features associated with the chosen query feature in the predefined mapping. Then, for positive datapoints, we choose randomly whether one or 2 of these features appear in the sequence. If 2 features appear, we randomly choose whether they appear in the same token. For negative datapoints, we ensure that none of these 2 features appear in the input sequence.

T4. The pipeline is similar to T2, the difference being that we randomly choose 1 query and we use its associated features.

T5. We choose one of the 2 pairs of features that are associated with one query. We then apply the same pipeline as in T2 while ensuring that at least 1 feature from the other pair doesn't appear in the sequence.

T6. For positive cases, we randomly choose 2 positions in the sequence and put the 2 tokens on these positions in the correct order. For negative cases, we randomly choose whether the datapoint contains both features in the wrong order or contains at most one of the features.

T7. We choose one of the 2 pairs of features that are associated with one query. We then apply the same procedure as in T6, while ensuring that at least 1 feature from the other pair doesn't appear in the sequence.

4 Models

The general structure of the models is presented in Figure 2. Model equations and training details are provided in the appendix.[2]

Transformer. The Transformer architecture consists of a sequence encoder and a decoder adapted to our tasks. We choose the most basic architecture for interpretability: a single-head, single-layer architecture. To enable incremental processing, the encoder self-attention only looks into the past, as in standard "causal" architectures such as the one of Dai et al. (2019).

The input vectors are mapped to a 100-dimensional space and combined with sine-based positional embeddings, as in Vaswani et al. (2017). Each input token vector then goes through a transformer encoder cell with one attention head and one layer. The query vector is also mapped onto a 100-dimensional space, through a separate embedding matrix. Decoding works as a dot-product-based attention between the query embedding and each token representation (Bahdanau et al., 2015).

The concatenation of the query embedding and the decoded sequence representation is then passed through a multi-layer perceptron (MLP) to generate the answer, a number in the range of [0,1] obtained via the Sigmoid function. The model is optimized with a binary cross-entropy loss. At test time, a result larger than 0.5 is considered a positive answer, as is standard.

We ablate the full Transformer architecture by removing self-attention and skipping positional encoding when embedding the input. The variants solely based on decoder attention are akin to Memory Networks (Sukhbaatar et al., 2015).

LSTM. In order to have a fair comparison with the Transformer model, we do minimal changes to the architecture in the LSTM model, only substituting the transformer self attention block with the LSTM cell.

We experiment with two variants of LSTM:

- (basic) LSTM: the input goes through the LSTM and the output of the last time step is considered the representation of the input and is concatenated with the query embedding to be sent to the MLP.

[2]The models were implemented using the framework Pytorch (Paszke et al., 2019).

Tasks↓ Models→	T-Pos-Att	T+Pos-Att	T-Pos+Att	T+Pos+Att
T1 1 feature	91.3±2.8	96.1±2.9	96.5±1.7	**98.7**±1.1
T2 2 features	83.4±0.2	**84.7**±0.5	83.2±0.2	**84.9**±1.5
T3 set member	98.7±1.2	**100**	99.4±0.9	**100**
T4 context. 2 features	86.2±0.6	**87.7**±0.3	86.4±0.6	**87.7**±0.2
T5 context. set member	77.4±0.4	77.7±1.2	**78.8**±2.5	77.8±1.1
T6 ordered feature	56.6±1	80.2±0.4	**92.1**±0.3	90.7±1.9
T7 ord. context. set member	57.5±0.4	65.8±0.8	**78.6**±0.6	67.6±0.6

Table 2: Results for sequence length 10. Results are averaged over 5 random seeds, with s.d. Best results boldfaced. Models are coded as follows: **+/-Pos** marks absence (-) vs. presence (+) of positional encoding, **+/-Att** marks absence (-) vs. presence (+) of self-attention. The full Transformer is model T+Pos+Att.

- attention LSTM: similarly to the Transformer model, there is an attention mechanism that compares the query embedding to the output from each time step (again using dot product). The input representation that will be concatenated to the query representation is then the weighted sum of the outputs.

5 Results

5.1 Transformer

Since the Transformer surpasses the LSTM in most cases, and the Transformer patterns are quite similar across sequence lengths, we first take a detailed look at the behavior of the Transformer on the tasks for sequence length 10. Table 2 summarizes the results for this sequence length. Recall that datapoints are always uniformly distributed between negative and positive answers, such that a random baseline always averages 50% accuracy.

Task 1 All the models succeed at T1 (single feature detection) with accuracy over 90%. As could be expected, there is a high correlation between the accuracy and the degree of decoder attention on the correct token: This attention is at an average of 0.82 when the answer is correct, vs. 0.17 when it is wrong.

Surprisingly, while this task doesn't involve contextualization nor order, both self-attention and positional encoding bring a boost in performance, from 91.3% accuracy for the base model (T-Pos-Att) to over 96% for the models T+Pos-Att and T-Pos+Att (which add positional encoding and self-attention, respectively). We conjecture that positional embeddings act as a beneficial noising mechanism, akin to regularization: altering a token's embedding depending on the position helps the model not to overfit.

We also find that self-attention triggers an inverse recency effect on accuracy: performance is better when the target feature is towards the beginning of the sequence. Indeed, there is a highly significant Pearson correlation of -0.51 between position and accuracy for models with self-attention vs. no significant correlation for models without it. This recency effect is probably due to the fact that self-attention can copy a feature through the following hidden states, so an earlier feature will tend be more prominent in the weighted sum, and thus easier to detect. We find that this copying mechanism improves results for earlier tokens without significantly harming performance in later tokens.

Thus, the first take-home message from our experiments is that the presence of a component in a Transformer architecture does not imply that the model will learn to use it as expected in a task, even if it puts it to a use that improves performance. In T1, self-attention preserves information and positional encoding possibly adds noise, and both have a serendipitous positive effect. This underscores the need for model analysis.

Task 2 When moving to two-feature detection (T2), there is a predictable drop in accuracy. The models have problems when the queried features are in different tokens (false negatives), with a 25% accuracy drop, and when only one target feature occurs in the sequence (false positives), with a 40% drop. The main reason why features in different tokens are missed is that decoder attention fails to operate distributively, i.e., to focus on two different input tokens at once: On average, the difference between the decoder attention weight of the most attended token and the second most attended token is of 0.6-0.7 across the models, showing that decoder attention indeed focuses on a single token.

Given that the feature combinations are random, it is not clear how self-attention could help, and indeed using self-attention does not improve results.

Task 3 The T3 results show that all model variants can easily learn to detect a class instance, or feature disjunction, even when using an indirect mapping. Decoder attention suffices, as the models learn to associate a query with both features of its class, and trigger high attention values when either of them is present in a token (average attention on target token between 0.6-0.7 for all variants). Here, model behavior is as expected.

Contextualization Self-attention should help with contextualized feature detection (T4 and T5), by highlighting a specific feature only when it is preceded by one of the other features it is associated with. However, models settle for a degenerate strategy instead, and thus we detect no competitive advantage for models with self-attention (compare models T-Pos-Att and T-Pos+Att in Table 2). In T4, they answer 'yes' if one specific feature is present in the sequence (always the same feature for each query): Average attention for the more attended feature in each query is around 0.9, while the average attention on the other relevant feature is around 0.05. The maximum accuracy for this strategy is 87.5%, which is about what the best models reach in T4.

Thus, the need to contextualize by itself is not enough for models to profit from self-attention; instead, as we will see next, the need to track order does trigger a productive use of self-attention.

Ordered feature detection. In T6 and T7, using self-attention brings accuracy from near chance to 92.1% and 78.6%, respectively (compare models T-Pos-Att and T-Pos+Att in Table 2). We find strong evidence that the models with self-attention use it to record the presence of the first feature in the hidden state of the token containing the second, as we predicted it should do already for tasks requiring contextualization: In both T6 and T7, model T-Pos+Att puts most of the decoder attention on the second feature-carrying token in the sequence, as we show next.

We report in Table 3 the difference between the average decoder attention on the second vs. first feature-carrying token for T6, T7, and T5, which is the unordered version of T7. In models that use self-attention as predicted, this difference will be large and positive. We see in the first column of the table that for T5, where order does not matter, the difference is very low, while it is much larger for T6 and T7. This confirms that there has been a change in strategy, with self-attention being productively used in the order-sensitive tasks.

	T-Pos+Att	T+Pos+Att
T5	0.03±0.02	0.01±0.01
T6	0.79±0.005	-0.14±0.55
T7	0.46±0.03	0.16±0.18

Table 3: Average decoder attention difference (and s.d.) between second and first feature-carrying tokens for the two models with self-attention in the contextualized tasks (T6 and T7), together with T5 for comparison.

Positional encoding is an alternative mechanism to track order, and for T6 and T7 it also improves results substantially with respect to the base models (compare models T-Pos-Att and T+Pos-Att), though much less than self-attention.

However, using both mechanisms actually harms performance (see results for T+Pos+Att). Self-attention and positional encoding seem to get in the way of each other, making it harder, when combined, for the model to converge on a single strategy to track order. We see evidence for this in Table 3, where the full Transformer (T+Pos+Att) has a much smaller difference for T7 compared to model T-Pos+Att, and a negative difference for T6 (meaning that it puts more attention on the first feature-carrying token). Moreover, the full Transformers exhibit a large standard deviation in both order-sensitive tasks, which means that different runs converge on different strategies. In contrast, the models with self-attention show a really low standard deviation; they always converge on the expected strategy.

Impact of sequence length Figure 3 presents average accuracies for sequence lengths from 5 to 30 (in steps of 5; error bars represent standard deviations across 5 random seed initializations). The patterns discussed for length 10 are generally confirmed at other sequence lengths. We further notice that longer sequences are beneficial for "easy" tasks, such as T1 and T3 (perhaps they help models avoiding trivial guessing strategies). On the other hand, when the complexity of the task is higher (Tasks 5, 6 and 7), longer sequences are detrimental to model performance. Accuracies for tasks for which degenerate solutions were found for sequence length 10 (T2 and T4) do not change across sequence length, indicating that the strategy does

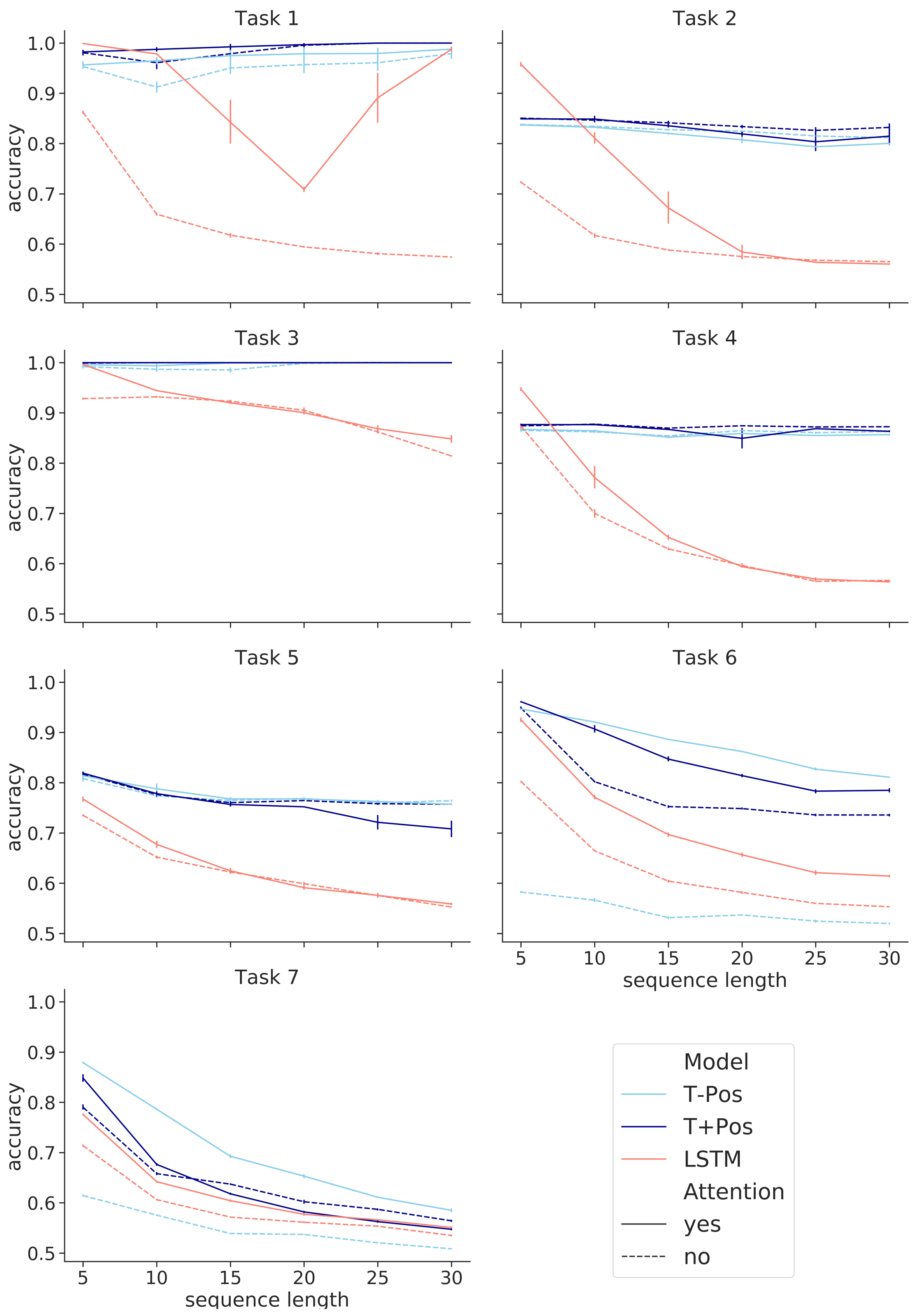

Figure 3: Model performance on multiple sequence lengths for all tasks

not change either. Also note that for the most difficult tasks (T5-T7), model T-Pos+Att is consistently better in longer sequences. Thus, the "getting in the way" effect observed above holds for the three tasks.

5.2 LSTM

In general, the performance of LSTM models on our tasks is markedly worse than that of the Transformer, and they are impacted by sequence length to a much larger extent (see Figure 3). This could be expected given general results on the two architectures (Transformers outperform LSTMs in most computational linguistics tasks, and LSTMs have been shown to have issues with long-distance dependencies). There are two notable exceptions to these general trends.

First, the accuracy of the LSTM with attention in Task 1 shows a characteristic V-shape, dropping at first and recovering for longer sequences (cf. orange solid line in Figure 3).[3] The result is that its performance is on par with that of the Transformer models in sequences of length 5 and 30.

We find that the LSTM changes its behaviour, from a simple recurrent LSTM to an actual attention-based LSTM. Indeed, for short sequences, it does not use decoder attention to identify the target feature (and still performs optimally): for instance, for sequence length 5, the attention is distributed uniformly, with attention values of around 0.2 on all the input tokens. Instead, for long sequences it does use the attention mechanism: the attention values spike on the position of the input that contains the queried feature (e.g., with an average of 0.85 for sequence length 30). This ability to switch behaviours could theoretically help also for other tasks, but it seems that the complexity of the tasks prevents the model from doing so.

Second, the LSTM with attention surpasses all Transformer variants for short sequences in Tasks 2 and 4. Recall that the Transformer models reached a degenerate solution for these tasks (with a maximum accuracy of 87.5%), in which only one of the two relevant features was attended to; instead, the LSTM solves the task correctly, because the information about the previous tokens flows through the recurrent steps in the token representations.

As for the differences between the two model variants of the LSTM, the model enhanced with the decoder attention is consistently better than the classic, basic model. This suggests that the decoder attention mechanism (also present in all the Transformer models) is beneficial independently of the base architecture.

[3]Instead, the basic LSTM model degrades quickly and monotonically with sequence length; see dashed orange line.

6 Conclusion

Our tasks shed light on how the main model components act and interact regarding the retrieval of discrete information from sequences, uncovering behaviours that would be difficult to detect when the architectures are applied to complex NLP tasks.

A take-home message from our experiments is that the presence of a component in an architecture does not imply that the model will learn to use it as expected in a task. In particular, we found that only the need to track ordered information led the architecture to use self-attention in the predicted way, and that decoder attention can be difficult for LSTMs to use correctly.

On the other hand, the components can assume unexpected functions. Self-attention in transformers can simply serve to blindly propagate information across time, leading to more robust representations of features contained in earlier tokens, that are copied over and over. Also, surprisingly, positional embeddings provide a small but consistent benefit in tasks that do not require order tracking. This suggests that they might have a serendipitous function, possibly adding helpful noise to the representations. Moreover, as both self-attention and positional embeddings can learn to keep track of order, the two mechanisms get in the way of each other, making it harder, when combined, for the Transformer models to converge on a single strategy to track order.

Regarding LSTM, the sequence length is a very big factor. The model has a steep loss in performance when the sequence gets longer. On the simplest task, the model enhanced with decoder attention develops the ability to switch between strategies, but this doesn't generalize to longer sequences.

To conclude, we hope to have shown that the proposed tasks constitute a useful probing mechanism for the ability of models to detect discrete information in sequences, testing in particular four key abilities: incremental processing (in the sense that the query is not known at input processing time), indirect mappings, context-dependence and order tracking. We have shown that most of the tasks are difficult for models even for short sequences; and they can be easily extended to more dimensions and even longer sequences.

Future work should go in two different directions: 1) examining how behavior changes when the probed models are scaled by increasing the number of layers and attention heads, and 2) modify the models to solve some of the problems that we encounter in the current experiments. We hope that the community will take up the analysis possibilities that our tasks afford, and that a clearer understanding of model behavior on the tasks will lead to better and more transparent models.

Acknowledgments

We gratefully acknowledge Laura Aina, Thomas Brochhagen and Sebastian Riedel for the feedback, advice and support. We are also grateful to the anonymous reviewers for their valuable comments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 715154). We are also grateful to the NVIDIA Corporation for the donation of GPUs used for this research. This paper reflects the authors' view only, and the EU is not responsible for any use that may be made of the information it contains.

References

Samira Abnar and Willem Zuidema. 2020. Quantifying attention flow in transformers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4190–4197, Online. Association for Computational Linguistics.

Afra Alishahi, Grzegorz Chrupała, and Tal Linzen. 2019. Analyzing and interpreting neural networks for nlp: A report on the first blackboxnlp workshop. *Natural Language Engineering*, 25(4):543–557.

Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015*.

Yonatan Belinkov and James Glass. 2019. Analysis methods in neural language processing: A survey. *Transactions of the Association for Computational Linguistics*, 7:49–72.

Grzegorz Chrupała and Afra Alishahi. 2019. Correlating neural and symbolic representations of language. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2952–2962, Florence, Italy. Association for Computational Linguistics.

Alexis Conneau, German Kruszewski, Guillaume Lample, Loïc Barrault, and Marco Baroni. 2018. What you can cram into a single $&!#* vector: Probing sentence embeddings for linguistic properties. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2126–2136, Melbourne, Australia. Association for Computational Linguistics.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, Florence, Italy. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. 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)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

John Hewitt and Percy Liang. 2019. Designing and interpreting probes with control tasks. 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)*, pages 2733–2743, Hong Kong, China. Association for Computational Linguistics.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Dieuwke Hupkes, Verna Dankers, Mathijs Mul, and Elia Bruni. 2020. Compositionality decomposed: how do neural networks generalise? *Journal of Artificial Intelligence Research*, 67:757–795.

Dieuwke Hupkes, Sara Veldhoen, and Willem Zuidema. 2018. Visualisation and 'diagnostic classifiers' reveal how recurrent and recursive neural networks process hierarchical structure. *Journal of Artificial Intelligence Research*, 61:907–926.

Brenden Lake and Marco Baroni. 2018. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International Conference on Machine Learning*, pages 2873–2882. PMLR.

Yair Lakretz, Dieuwke Hupkes, Alessandra Vergallito, Marco Marelli, Marco Baroni, and Stanislas Dehaene. 2021. Mechanisms for handling nested dependencies in neural-network language models and humans. *Cognition*, page 104699.

Tal Linzen, Emmanuel Dupoux, and Yoav Goldberg. 2016. Assessing the ability of LSTMs to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics*, 4:521–535.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8026–8037.

Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2021. A primer in bertology: What we know about how bert works. *Transactions of the Association for Computational Linguistics*, 8:842–866.

Koustuv Sinha, Robin Jia, Dieuwke Hupkes, Joelle Pineau, Adina Williams, and Douwe Kiela. 2021. Masked language modeling and the distributional hypothesis: Order word matters pre-training for little. *arXiv preprint arXiv:2104.06644*.

Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. 2015. End-to-end memory networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems-Volume 2*, pages 2440–2448.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. 2019. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5797–5808, Florence, Italy. Association for Computational Linguistics.

Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. 2015. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*.

Jennifer C White and Ryan Cotterell. 2021. Examining the inductive bias of neural language models with artificial languages. *arXiv preprint arXiv:2106.01044*.

Appendix

Transformer implementation

Each datapoint has **n** 36-dimension binary vectors as input **in** and a 36-dimension binary vector as query **q**. All these vectors are embedded to 100 dimensions using the matrices W_{in} and W_q. Then we apply a sinusoidal positional encoding on the input embeddings and the ReLU function on top of the query embedding:

$$emb_{in_i} = PosEnc(in_i * W_{in})$$
$$emb_q = ReLU(q * W_q)$$

with PosEnc defined for each input position *pos* and for each token dimension *i* similarly to the method used in the main Transformer architecture (Vaswani et al., 2017):

$$PosEnc(pos, 2i) = sin(pos/10000^{2i/d_{model}})$$
$$PosEnc(pos, 2i+1) = cos(pos/10000^{2i/d_{model}})$$

Then, each input embedding goes through the TransformerEncoder cell which has 1 attention head and 1 layer:

$$h_i = TransfEnc(emb_{in_i})$$

as defined in Vaswani et al. (2017), TransfEnc is a scaled dot product attention where *d* is the dimensionality of the input vectors. The dot product is scaled in order to not have regions with very slow gradients.

$$TransfEnc(X) = softmax(\frac{X * X^T}{\sqrt{d}}) * X$$

In order to decode relevant information based on the query we use a dot product attention. α_i represents the attention value that we put on token embedding h_i relative to the query embedding. We then calculate the vector c which is the sum of the token embeddings weighted by α.

$$\alpha_i = \frac{\exp(h_i * emb_q)}{\sum_{k=1}^{n} \exp(h_k * emb_q)}$$
$$c = \sum_{i=1}^{n} \alpha_i h_i$$

We then use a multi-layer perceptron to generate the answer. Firstly, we apply a dimensionality reduction using W_{o1} from 200 to 100 dimensions with ReLU as nonlinearity on top of it. Then the hidden state hid_o is mapped from 100 to 1 dimension. Using the Sigmoid function, we get our result as a number in the range of [0,1]. At test time, if $o \geq 0.5$, then we consider the answer to be positive.

$$hid_o = ReLU((c||emb_q) * W_{o1})$$
$$o = Sigmoid(hid_o * W_{o2})$$

The loss is calculated using binary cross entropy.

LSTM implementation

In order to have a fair comparison with the Transformer model, we do minimal changes to the architecture, only substituting the transformer self attention block and the positional encoding with an LSTM cell.

Learning procedure

All the experiments are optimized with Adam and learning rate 0.0001. We apply 0.2 dropout and gradient clipping at 0.5. We run 100 epochs with batch size 10 and save the model at the epoch with the highest validation accuracy. We experimented with different hyperparameters, but we found the ones we just reported to give the most stable results.