

Needle in a Haystack: Probing Transformer Capabilities to Recognize Non-Star-Free Languages

Stanford CS224N Custom Project

Richard Gu
Stanford University
yrichard@stanford.edu

Sambhav Gupta
Stanford University
samgupta@stanford.edu

Andy Tang
Stanford University
andyt2@stanford.edu

Abstract

Transformer-based language models have become ubiquitous, but in applications with structured output model reliability remains questionable. Previous work on the toy problem of understanding regular languages show limitations on transformer recognition of non-star-free languages. We partially reproduce these results but show the promise of these models to classify certain non-star-free regular languages like a^*b^* that we term “haystack languages,” where specific “needle” discrepancies are enough to reject a candidate string, and interpretability probes show intuitive associations in intermediate activations though precise mechanisms remain elusive.

1 Introduction

Our mentors are Aditya Agrawal (TA) and Zhengxuan Wu. We are not sharing the project, and all members proposed and ran experiments, gathered and visualize data, and wrote and revised large sections of the report.

Large transformer-based language models have become a backbone for natural language processing, applied to tasks as diverse as translation, code generation, and genomic prediction. However, until recently these models have been taken as “black boxes”, with more attention paid to the quality of their output than their internal mechanisms. Interest in mechanistic interpretability has increased as architectures have grown in complexity, to provide assurances for model safety and reliability.

We focus specifically on regular languages to provide a toy model for interpretability results that may be extendable to larger models in the wild. Regular languages are especially interesting for interpretability work because the human-understandable rules that govern it coincide with weaknesses of general transformer architectures: for example, Bhattamishra et al. (2020) describes deficits in transformer understanding of non-star-free languages compared to recurrent networks. We apply interpretability techniques such as visualizing intermediate activations and attention outputs to better understand how transformers might learn (be able to correctly classify) certain regular and counter-based languages without hacks like feature engineering or external memory, especially the non-star-free languages that previous transformers had difficulty with.

We find that simple versions of what we call “haystack languages,” which are non-star-free languages which can be accepted or rejected based on a single local string, are in fact often recognizable by transformer classifiers, although accuracy decreases as complexity increases. Data augmentation via random padding often improves performance, but adding feed-forward networks and embedding engineering does not, and further that attention correlations emerge in an intuitive way between characters which relate to one another. Improving our understanding of these languages in this manner likely has downstream benefits, as many phenomena in the real world can be modeled as languages with certain constraints, such as code or genetic sequences.

2 Related Work

As mentioned previously, substantial work has been done in understanding the ability of transformer-based models to identify regular languages. Regular languages are a subset of all languages recognizable by a Turing machine, and are of interest as simple problems for traditional human-designed algorithms that nevertheless present challenges for large language models. Specifically, Bhattamishra et al. (2020) find that while single-layer transformers are able to learn certain families of languages like Shuffle-Dyck, they have trouble learning non-star-free languages like $(aa)^*$. They found that these languages can only be learned with certain engineering of the positional encoding, such as for the above example setting the positional encoding to a period of 2, which can not be generalized as a technique. Additionally, they note that RNNs such as LSTMs are able to learn these same languages well, showing a deficiency in the ability of transformers.

More recent work has focused on the ability to engineer transformers to behave better with regular languages. Chi et al. (2023) present an architecture called RegularGPT which employs several representational tricks such as sliding-dilated-attention, weight sharing, and adaptive-depth to solve languages such as PARITY which vanilla transformers do not do well on. The combination of these features serves to introduce something akin to working memory in humans or the hidden state carried through computations in an RNN. However, such engineering to fix the network for a few specified regular languages does not promise to generalize across all languages.

Our work hopes to better characterize the specific classes of languages that transformers work well on, refining Bhattamishra et al. (2020) and giving insight on what modifications might be necessary to allow transformers to perform predictably on all regular languages. This work has significant precursors in other areas of machine learning: for example, Szegedy et al. showed that neural architectures trained on ImageNet could be fooled by imperceptible adversarial perturbations, directly leading to works on architectures and methods to resist these attacks such as Madry et al. (2019). Similarly, Yosinski et al. (2015) has helped guide and interpret later architectural improvements such as Ledig et al. (2017). As such, it is a natural next step to apply these same techniques to transformers which classify regular languages to achieve better understanding of transformer-based language model architectures similar to existing understanding of image classification models.

3 Approach

We propose the concept of the *needle* of a regex, indicating the specific substring that directly leads to the decision of match or non-match for a given string against the regex, and call all languages with such needles “haystack languages”. For instance, the regex “ a^*b^* ” has the needle “ ba ”, since any string consisting of “ a ” and “ b ” that includes the substring “ ba ” does not match with the pattern “ a^*b^* ”, whereas all strings lacking this substring do match. Another example is “ aaa ” for the regex “ $(?!.*aaa)[ab]^+$ ”. Since we hope to interpret our models rather than simply train models with high performance on regex strings, our main focus will be on regexes that have a corresponding needle, as they provide clearer patterns or structures for interpretation. We will trace through the network to determine which parts of the architecture are responsible for identifying specific features.

The formal problem that our transformer-based classifier aims to solve is: *given an example of a string, output 1 if it matches the regex and 0 otherwise*. To identify optimal pairings of model architectures and regex patterns for interpretability research, we conducted a series of preliminary experiments examining various factors, including regex patterns, type of positional embeddings, number of attention heads, and the presence or absence of initial padding and feed-forward networks (FFN) within the Transformer blocks. After analyzing the cases from experiments above, we focus one case of success and two cases of failure to account for the factors tested above.

3.1 Single Headed Transformer Classifier With One-Hot Positional Embeddings

We define the following model architecture with the goal of producing maximally interpretable downstream effects from absolute positional embeddings, though we acknowledge that this positional embedding scheme does not generally scale. The model is illustrated in Figure 1.

The model is as follows:

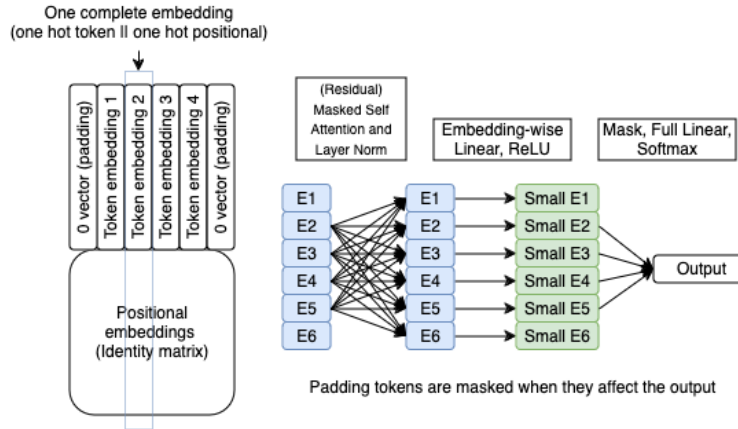


Figure 1: The most refined transformer-based classifier model we train on various regex languages.

1. Let there be m non-padding tokens and let the sequence length be L . The input indices is transformed to an $m \times L$ embedding matrix, where each column is a one-hot vector for its corresponding index and the padding token receives the 0 vector embedding.
2. We concatenate the $L \times L$ identity matrix under the embedding matrix, which adds a position index to each column.
3. We pass these (column) embeddings through single-headed self attention. Self attention outputs are added to embeddings, then passed through layernorm. This is followed by a linear layer which squeezes the new (long) embeddings to a very small hidden dimension d , activated by ReLU.
4. We mask all activations corresponding with padding tokens to 0, then flatten all $L \cdot d$ activations and pass them to a final fully connected layer. We scale the output of this layer by the number of non-padding tokens to normalize for the masking operation, then take a sigmoid activation of the final result.

In some experiments, we use a version of the model with no intermediate layer between attention and the output, which is no longer of the same form as a transformer encoder. We also try sinusoidal embeddings in place of one-hot positional embeddings, and in some cases enhance our model by masking out padding tokens.

3.2 Languages

We use several languages to attempt to interpret our model outputs. In general, p is a padding token and s and e are start and end tokens, respectively. We generally wrap all strings in a start token and end token, then fill the remainder of the context with padding, where the padding can occur before the string as well.

We use the regex $\sim [xyz]^+ \$$ (i.e., strings that only contain the characters x, y, z) to analyze single-character needles, which are needles consisting of exactly one character. We test for multiple-character needles with the regex a^*b^* .

We use the regex $(aa)^*$ as a beginner-level representative of counter languages, and a direct comparison with Bhattamishra et al. (2020). For this language, the model must find a way to compute the parity of the number of a characters. The cases with and without initial paddings are both tested.

We use the regex $\sim (?!.*abc)[abc]^+ \$$ (strings consisting of character a, b, c but without substring "abc") and $\sim (?!.*bca)[abc]^+ \$$ (similar to above, but without substring "bca") to test the case of multiple-character key in a random string of a, b , and c .

We use $\sim (?!.*aaa)[ab]^+ \$$ (strings consisting of "a" and "b" with no 3 consecutive "a") and $\sim (?!.*aaaa)[ab]^+ \$$ (similar to the former, but with no more than 4 consecutive "a") to test how the length of the needle influences the model's accuracy.

3.3 Interpretability

To interpret these models, we evaluate their effectiveness against our base datasets to verify their ability to learn the regex. We run experiments to better understand each component of the transformer in improving (or not improving) the model’s understanding of the regular expression, including various means of visualizing intermediate weights and outputs. We wrote all the code for training models, ablations, and interpreting model attention, intermediate activations, and outputs, as given the small size of our models, more sophisticated methods like interventions on the model proved unnecessary.

4 Experiments

4.1 Data

We generate our own datasets for each language, as stated above. Positive examples are generated via the regex, while negative examples are generated as follows. \sim [xyz]+\$: include distractor characters ‘a’ and ‘b’. $a*b*$: generate strings and insert “ba” into them. $(aa)^*$: generate odd-length strings. All other regexes: same as $a*b*$, ensure that the “needle” is in a random string with appropriate tokens.

4.2 Evaluation method

We generate a 50/50 balanced dataset of regex positive and negative examples up to MAX_LENGTH for each respective dataset, which is then split 80/20 into a train and test dataset. Additionally, we generate another *out-of-distribution* dataset that contains 50/50 positive and negative examples of length at least MAX_LENGTH + 1 (in practice, up to twice the in-distribution maximum length). If our model performs well on both test and out-of-distribution datasets, we can have confidence that our model is learning the underlying regular language.

4.3 Experimental details

We train our model on each of the languages previously defined. A basic summary of results follows, from each of our languages trained with learned positional encoding, an embedding space of dimension 6, only one attention head and one layer, no padding masking, and generally MAX_LENGTH = 200 and OOD_MAX_LENGTH = 400 for 10 epochs, with random initial padding. Enhancements and ablations are reported with changes from this default procedure.

4.4 Results

For functionality of our transformer models, we aggregate data about each model’s train, test, and validation accuracy.

Contrary to Bhattamishra et al. (2020), we do find that our transformers are able to learn some non-star-free languages, suggesting that the relationship between star-free languages and learnability by transformers is more complex than previously stated, although similar to their paper we find that counting-based languages like $(aa)^*$ are difficult for transformers to learn. We note that this is an expected result, as on the surface, transformer-based networks do not appear to have an explicit way to encode a counter (as some RNN and LSTM-based networks do).

Language	Train Accuracy	Test Accuracy	OOD Accuracy
$a*b*$	0.950	0.947	0.934
$\sim(?!.*a)[xyz]^+$	1	1	1
$\sim(?!.*[ab])[abxyz]^+$	1	1	0.918
$(aa)^*$	0.502	0.5	0.502
$\sim(?!.*abc)[abc]^+$	0.656	0.658	0.711
$\sim(?!.*aaa)[ab]^+$	0.935	0.930	0.940
$\sim(?!.*aaaa)[ab]^+$	0.872	0.873	0.777

Table 1: Functionality

4.4.1 Enhancements

One interesting phenomenon we found was that for some needle-style star-free languages, adding randomized initial padding helped with model accuracy. Notably, this randomization helped even though other forms of data augmentation (such as including more diverse negative examples and weighting each positive example multiple times in the dataset) did not help or even hurt performance. Some results are included below:

	Original	Initial Padding Ablated
a*b*	0.950/0.947/0.934	0.877 / 0.881 / 0.525
^(?!.*a)[xyz]+	1/1/1	1/1/1
^(?!.*[ab])[abxyz]+	1/1/0.918	1/0.954/0.957
(aa)*	0.502/0.5/0.502	Not Trainable ¹
^(?!.*abc)[abc]+\$	0.656/0.658/0.711	0.650/0.650/0.691
^(?!.*aaa)[ab]+\$	0.935/0.930/0.940	0.928/0.921 /0.914
^(?!.*aaaa)[ab]+\$	0.872/0.873/0.777	0.865 / 0.866 / 0.751

Table 2: Results for various languages with and without initial padding

We found that learnable or sinusoidal embeddings generally did not have a significant effect on the ability of our model to learn the language. Despite differences in methodology (with their formulation of the problem as next-token prediction versus ours as classification), we reproduced the result of Bhattamishra et al. (2020) that the specific language (aa)* is not easily learnable by general positional encoding schemes.

To test the functionality of positional embedding, we use the exmaple of ^(?!.*abc)[abc]+\$, which requires the model to distinguish the c after different suffixes, and is the other language our model with initial padding performs poorly on. Ideally, positional encoding should differentiate “c” with “ab” or “a” in the front. The results are presented below:

	Learnable	Sinusoidal
6/1/1	0.656/0.658/0.711	0.658 / 0.664 / 0.682

Table 3: Results for learnable or sinusoidal positional embedding

Interestingly again, we find that increasing the number of layers and attention heads for a model does not necessarily increase its accuracy, despite the expectation that more layers might better help match longer “needles” as in the trial below.

	^(?!.*aaa)[ab]+\$	^(?!.*aaaa)[ab]+\$
6/1/1	0.928/0.921 /0.914	0.745 / 0.724 / 0.716
12/2/2	0.910 / 0.903 / 0.861	0.732 / 0.721 / 0.653
18/3/3	0.910 / 0.901 / 0.827	0.745 / 0.726 / 0.640

Table 4: Results for ^(?!.*aaa)[ab]+\$ and ^(?!.*aaaa)[ab]+\$

The results presented above indicate that the depth of the model does not necessarily correlate positively with accuracy. A possible reason is that large models start to overfit on the relatively small dataset rather than learning its pattern, therefore performing worse on the OOD dataset.

5 Analysis

This section is in various subsections corresponding to different qualitative interpretability experiments to explain previous quantitative results.

5.1 Initial Padding

We analyzed the attention weights of the regex ^(?!.*[ab])[abxyz]+ to understand why initial padding is beneficial. For ^(?!.*[ab])[abxyz]+, negative strings contain at least one

¹Only 100 positive examples and 100 negative examples could be generated without initial padding tokens.

occurrence of the characters ‘a’ or ‘b’. Thus, we hypothesize that the model should use the attention operation in some capacity to identify ‘a’ and ‘b’ characters. Indeed, in Figure 2, each row represents the attention scores for a token and the query vector it represents (so the heatmap is the matrix $\text{softmax}(QK^T/\sqrt{d_k})$). The highlighted vertical lines correspond directly with the positions of ‘a’ and ‘b’, confirming that the attention operation learns to pick out these tokens. The non-padding tokens are located from position 26 to 90, evident from two clear horizontal cuts in the graph. When we do not mask padding, we find that some non-padding characters, such as ‘x’, diffuse the attention by assigning similar scores across all tokens. In contrast, padding tokens consistently identify the key negative characters. This observation also suggests that the effectiveness of padding is not necessarily due to its initial placement but rather its general presence in the sequence.

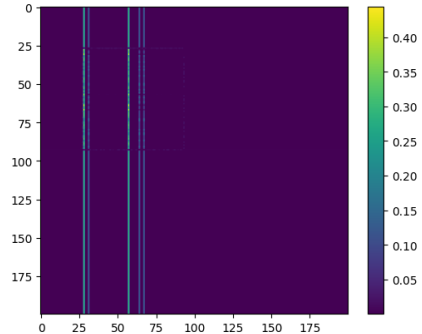


Figure 2: Attention weights for a negative sample of $\text{^(?!.*[ab])[abxyz]+}$

5.2 Positional Encoding and FFN

We conducted tests on positional encoding and feed-forward network with the regex $\text{^(?!.*abc)[abc]+\$}$, which identifies “abc” as its critical needle, hoping to improve accuracy beyond the 60% to 65% from the vanilla method on all datasets. The graph below illustrates that the vanilla model struggles to learn meaningful positional encodings. Switching to sinusoidal positional encoding slightly improved accuracy, yet the model still failed to assign a significantly higher value to ‘c’ following ‘ab’, which is the needle in this scenario.

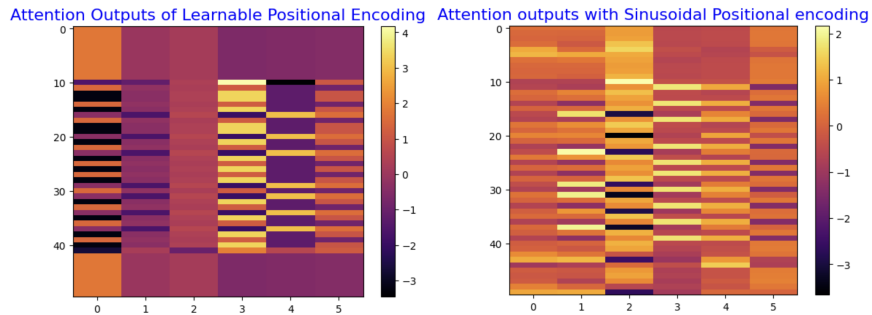


Figure 3: Attention Outputs with Learnable and Sinusoidal Positional Encoding of $\text{^(?!.*abc)[abc]+\$}$

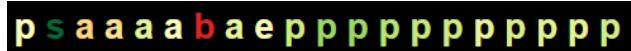
As illustrated on the left of Figure 3, despite using learned embeddings, often result in similar attention output across the string for a specific character, regardless of contextual differences. This demonstrates that learned embeddings did not effectively capture positional nuances. In contrast, sinusoidal encoding introduced some variability in the embeddings as shown on the right. However, this variation was still insufficient to focus on the critical needle ‘abc’.

We further probed the ability of multi-head attention and feed-forward network (FFN) to improve results on these regular languages. In the cases of these regular languages, different attention heads indeed tend to different characters. FFN was a near-identity of the attention outputs: for associated graphs, see the appendix. Multi-head attention also was ineffective: for graphs, see appendix.

5.3 Case Study: a*b* Mechanisms

To dive deeper into the mechanics that these models learn, we look at failure cases of a*b*, especially with shorter-length strings. We first attempted to follow Szegedy et al. Szegedy et al. with adversarial gradient ascent, but found this difficult due to character choices being discrete, even after implementing the reparameterization trick Jang et al. (2017) on categorical distributions (also known as Gumbel softmax). Our model is quite robust at medium-length strings (30 to 200 characters), especially with initial padding, so even a policy-gradient reinforcement learning approach (REINFORCE as in Sutton Sutton et al. (1999)) fails to generate substantial negative examples at this length.

However, a closer analysis of a short false positive example reveals interesting traits of our basic algorithm. Here, each position is colored by its total contribution towards the classification of the string: the red 'b' is highly negatively correlated, as expected, but interestingly the initial padding token is correlated quite positively. Further, padding tokens in general are biased slightly positive.



Looking at one of the few false negatives also seems to show additional artifacts not conducive towards learning the language well (despite impressive results both in and out-of-distribution). Strangely, many valid 'b' characters have a negative contribution to the overall result, especially at short lengths. One potential explanation is that this is due to the low amount of overall positive examples of this particular language, leading to fewer examples to learn from for short strings.



Interestingly, averaging last-layer activations over our dataset appears to show that overall the average bias of our entire set of activations is near-zero. However, average individual activations, such as the contribution of the second token in the string, are variable (in this case, average second token contribution is +0.9606 towards a positive classification).

Size	5	10	20	40
Average Bias	0.0513	0.0214	-0.0419	0.0045

Table 5: Average Bias in Logit Sum

One way to explain this intriguing behavior is that our transformer model might put details of the entire distribution of the language encoded into both early tokens and useless padding tokens. Indeed, attempts to induce errors in classifying 100-character valid examples seem to only succeed if the "a" section of the string is remarkably short, overlapping with this candidate "memory region."

To solve this issue of the memory region, we try explicitly masking padding. We achieve slightly higher in and out-of-distribution results here, with around 97% accuracy on both, but the reliance on padding tokens has been edited out, resulting in fewer errors on short strings. We plot the attention scores for five highly positive and negative scoring examples from the test set, which can be seen in 4. In this case, we see a very strong attention pattern emerge in both cases; for positive examples, 'b' characters attend very strongly to 'a' characters, especially those at the beginning and end of the run of 'a's, while 'a' characters attend more weakly to 'b' characters, biased towards the third quartile. In negative results, there is visible banding from the random strings, and the 'b' characters still attend to 'a' characters where they occur.

Notably, in all variations of our methods and experiments, we were unable to find evidence for the most human-rational way of understanding a*b*: that is, looking for "ba" and rejecting any string that includes it. In the future, we hope to probe deeper for more subtle approaches these transformers may use to identify these non-star-free languages.

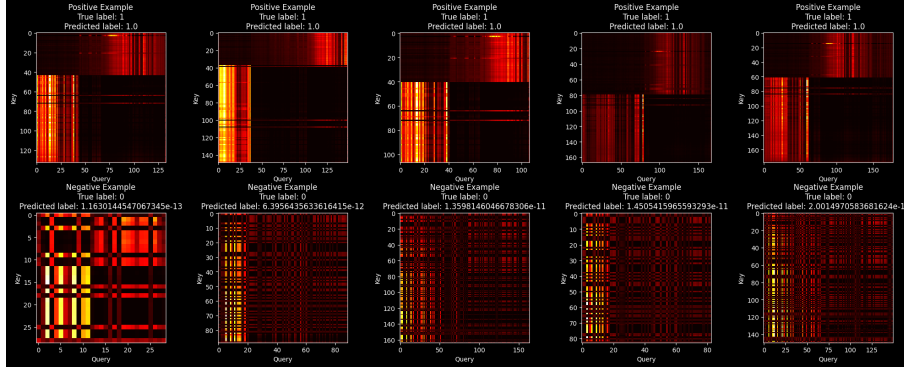


Figure 4: Attention scores for five strong positive and negative examples for a^*b^*

6 Conclusion

Upon running a large variety of regex datasets through a large variety of models, we are left with a few key results:

1. Transformers are efficient at finding single-character needles. Augmenting padding often helps transformers, and they can use padding for additional computation.
2. Transformers generally have difficulty finding multi-character needles and counting, including when provided with learnable or sinusoidal positional embeddings. This is in line with results from Bhattamishra et al. (2020). Furthermore, learned embeddings cause uniformity in attention scores, while sinusoidal embeddings do not.
3. Multi-head and multi-layer attention have limited effect on improving transformer performance on multi-character needles. This provides evidence that, in general, multi-character needles is a hard problem for the transformer architecture, although with greater scale these challenges may be surmounted.
4. Transformers do well on the a^*b^* language in general. The attention scores of the attention layer for this language hint at the model parsing the text as a human might, by attending strongly to the corners of runs of 'a's and 'b's. We also find that b characters seem to always push the final output negative.

Overall, we noted several key results which all provide great motivation for further study; in particular, results on finding multi-character keys warrant further study to discover either (1) multi-character needle languages which are learnable by this class of transformer classifier, or (2) augmented transformer classifier architectures which are able to learn multi-character needles.

We also note that we were unable to interpret our models strongly enough to obtain a complete understanding of their function; even at small scales and for simple languages, we found that the transformer models learn processing methods with no clear interpretation. In future work we hope to explore this more deeply, especially on how values are learned by the attention mechanism.

7 Ethics Statement

In conducting this research, we adhered strictly to ethical standards in computational research for problem formulation, experimentation, data handling, and report writing processes. Aditya Agrawal (Teaching Assistant) provided mentorship over our proposal and milestone, ensuring that our methodologies and interpretations remained scientifically rigorous and transparent. By investigating 'black box' models, we aim to contribute positively to the fields of AI safety and reliability, providing insights that may help in crafting more accountable and comprehensible AI systems.

References

- Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. 2020. On the ability and limitations of transformers to recognize formal languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*.
- Ta-Chung Chi, Ting-Han Fan, Alexander Rudnicky, and Peter Ramadge. 2023. Transformer working memory enables regular language reasoning and natural language length extrapolation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5972–5984, Singapore. Association for Computational Linguistics.
- Eric Jang, Shixiang Gu, and Ben Poole. 2017. Categorical reparameterization with gumbel-softmax.
- Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, and Wenzhe Shi. 2017. Photo-realistic single image super-resolution using a generative adversarial network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2019. Towards deep learning models resistant to adversarial attacks.
- Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, volume 12. MIT Press.
- Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations*.
- 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*, volume 30. Curran Associates, Inc.
- Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. 2015. Understanding neural networks through deep visualization.

A Appendix

A.1 FFN Graphs

We probed the role of the Feed-Forward Network (FFN) as implemented by Vaswani et al. Vaswani et al. (2017) in understanding regular languages. We tried ablating the FFN by connecting the output of the attention mechanism directly to the output projection layer. The graphs in Figure 5 provide insights into the impact of these architectural changes:

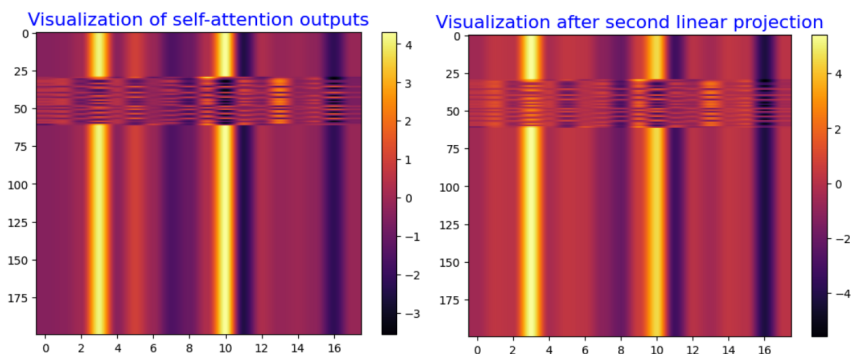


Figure 5: Attention outputs for a negative sample of $^{(?!.*abc)[abc]+\$$ with and without a FFN layer

This is a qualitative analysis of a negative match for $\sim(?!.*abc)[abc]+\$$. Both graphs represent outputs of dimension 200×18 , the second of which has gone through a fully-connected network. Our analysis reveals that the attention distribution is remarkably similar across both graphs, with only minor variations in value. This similarity suggests that the FFN does not significantly alter outputs, indicating its limited utility in this specific context.

A.2 Multi-head Attention

We hypothesized that with a diverse vocabulary containing several characters, different attention heads might specialize (divide the work) and focus on different characters. To test that, we compared the output of models equipped with single-head attention and three-head attention.

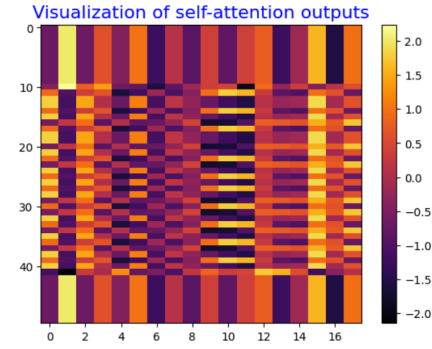


Figure 6: Three-head Attention Outputs of $\sim(?!.*abc)[abc]+\$$

The attention patterns reveal distinct preferences in weight assignment by different heads. Specifically, in the first attention head (columns 0-5), the weights prioritize character ‘c’ over ‘b’ and ‘a’, respectively. Conversely, the second head (columns 6-11) shows a preference for ‘a’ over ‘b’ and ‘c’. The third head (columns 12-17) assigns the highest weights to ‘c’, followed by ‘a’ and ‘b’. While the distribution of vocabulary among different attention heads leads to high accuracy of the detection of ‘a’, ‘b’ in $\sim(?!.*[ab])[abxyz]+\$$, it has limited influence on a multi-character needle such as “abc”, as it does not interpret the substring ‘abc’ in a consecutive manner.