Transformers and Makemore

Adam Zsolt Wagner

Hausdorff School: "Machine Learning and Theorem Proving"

Day 3

September 22, 2023

Overview

In this lecture:

1. What are transformers and how do they work? A very rough overview.
2. Lots of toy examples of using transformers on mathy datasets.
3. Possible ways to use transformers in maths research

I am by no means an expert on transformers! I learned the most from the following sources:

- Andrej Karpathy's tutorial video: "Let's build GPT: from scratch, in code, spelled out."
- Makemore code: `https://github.com/karpathy/makemore`

Introduction to Transformers

- Transformers are a pivotal architecture in machine learning.
- Introduced by Vaswani et al. in 2017. ("Attention is all you need")
- Revolutionized Natural Language Processing (NLP) and other domains.

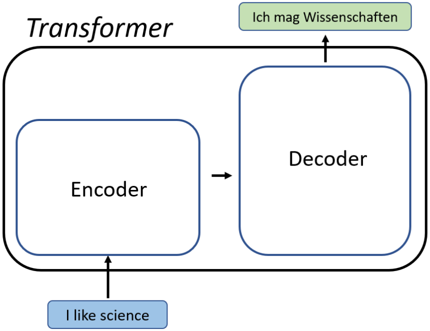


Figure: Transformer Architecture

Key Components of Transformers

1. **Self-Attention Mechanism:** Allows models to weigh the importance of different words in a sentence when making predictions.
2. **Positional Encoding:** Adds positional information to input sequences.
3. **Token Embedding:** Adds contextual information to input sequences.

Understanding Attention Mechanism

- The **Attention Mechanism** is a fundamental building block of Transformers.
- It allows the model to focus on different parts of the input sequence when making predictions.
- In a nutshell, attention assigns different weights to each input element.
- These weights determine how much each element contributes to the output.

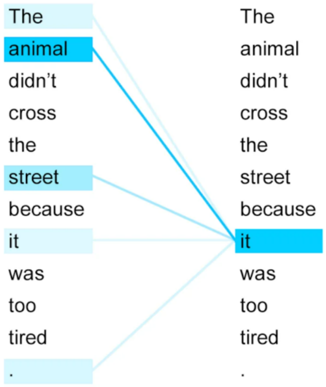
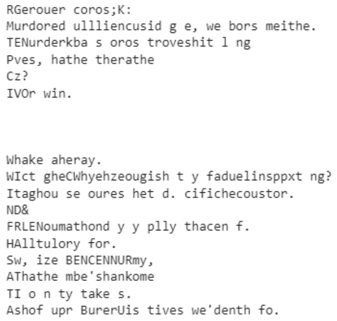


Figure: Attention Mechanism

The Bigram Model and the Need for Attention

- Consider the **Bigram Model**: Predicts the next letter based only on the previous letter, ignoring all other context.
- When trained on Shakespeare's novels, it produces text that lacks coherence and context.
- To improve this, we could take the average of all previous tokens.
- But taking a simple average may not capture context well.



```
RGerouer coros;K:
Murdored ullliencusid g e, we bors meithe.
TENurderkba s oros troveshit l ng
Pves, hathe therathe
Cz?
IVOr win.



Whake aheray.
WIct gheCWhyehzeougish t y faduelinsppxt ng?
Itaghou se oures het d. cifichecoustor.
ND&
FRLENoumathond y y plly thacen f.
HAlltulory for.
Sw, ize BENCENNURmy,
AThathe mbe'shankome
TI o n ty take s.
Ashof upr BurerUis tives we'denth fo.
```

Figure: Bigram Model

Self-Attention: A Closer Look

- In self-attention, each token in the input sequence emits two vectors: a **query** and a **key**.
- Think of the **query** as asking, "What am I looking for?" It represents the token's context of interest.
- The **key** is like asking, "What do I contain?" It represents the token's information.
- To compute affinities between keys and queries, we take dot products:

$$\text{Affinity}(Q_i, K_j) = Q_i \cdot K_j$$

- High alignment between a key and query results in a larger weight, indicating the importance of that token for the query.

Self-Attention: Computing the Context Vector

- After computing affinities between queries and keys, we apply a **softmax** function to obtain normalized weights:

$$\text{Attention}(Q_i, K_j) = \frac{\exp(Q_i \cdot K_j)}{\sum_j \exp(Q_i \cdot K_j)}$$

- These normalized weights, often called **attention scores**, represent how much each key contributes to the query.
- We then use these attention scores to take a weighted sum of the **values** associated with the keys:

$$\text{Context Vector}(Q_i) = \sum_j \text{Attention}(Q_i, K_j) \cdot V_j$$

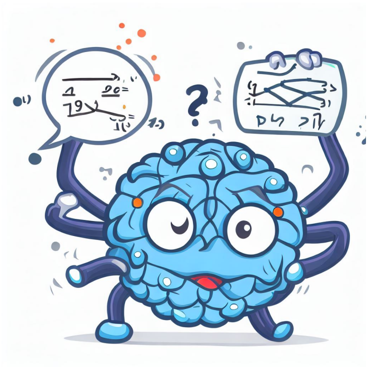
- The resulting **context vector** for each query captures relevant information from the entire sequence.

Queries, Keys, and Values

- In self-attention, we have three components at each token:
  1. **Query**: Think of it as saying, "Here is what I'm interested in." It represents the token's context of interest.
  2. **Key**: Consider it as answering, "Here is what I have." It represents the token's stored information.
  3. **Value**: Imagine it as conveying, "If you find me interesting, here is what I will communicate to you." It's the token's contribution to the output.
- Together, these networks enable self-attention to dynamically weigh and combine information from across the sequence.

Enough theory, let's do some maths!

Theory is Important,
But...
Let's Dive into Some Fun Toy Examples!



Experimenting with transformers

Let's create a dataset. It will contain 1000 strings of 0's and 1's, of length between 30 and 40. The strings will be completely random, with only one rule: the 13th and 22nd digits are always equal.

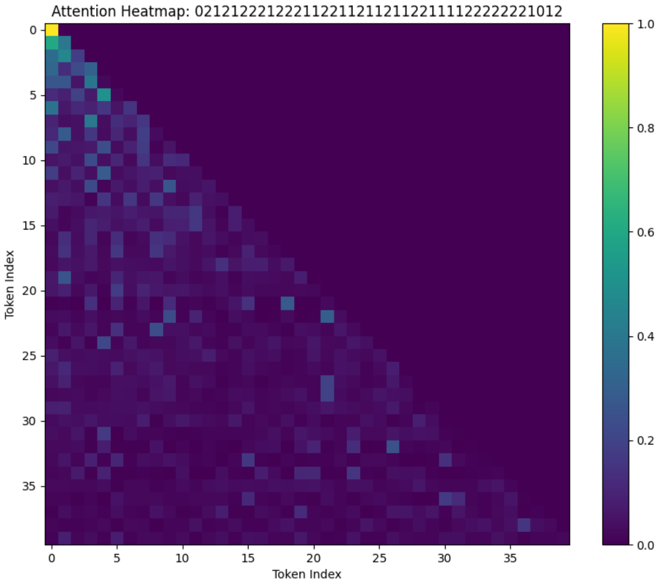
Let's train Makemore on this dataset for 15,000 iterations, and then generate 1000 more strings with it.

- What percentage of the output will have their 13th and 22nd digits equal?
- What will the attention heatmap of the transformer look like, after learning?

Experimenting with transformers

After 15,000 training steps, the success rate is **99.7%**.

Heatmap after 15,000 iterations



Explaining the lack of pattern in heatmap

I asked experts, everyone had a different explanation...

- Problem was too easy: the transformer managed to learn it without using its attention mechanism!
- It is unclear that transformers can really learn to pay attention to absolute positions – only relative positions and meanings.
- Two other explanations that were very technical and I didn't understand them.

My takeaway: we are still struggling to understand these things.

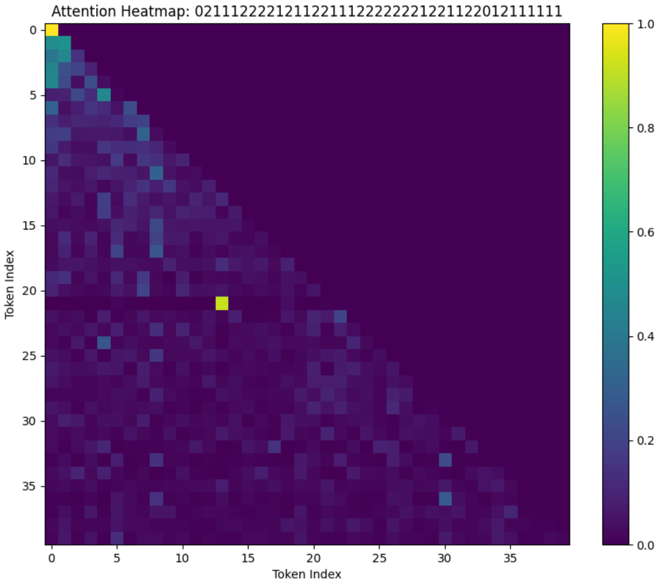
Regularization

There is something called a *weight decay*. The idea is that we don't just want to minimize the error, but also the total sum of weights of the neural net.

Goal: the network will learn a simpler function. Instead of memorizing the million entries in the dataset, it will be incentivized to learn the rule!

Let's add weight decay to the previous transformer, and see if it learns the rule properly!

Heatmap after a bit more training



Making sense of the heatmap

Adding weight decay made the transformer learn the rule, like we hoped...

... is what I would say, had I added the weight decay functionality before producing the picture.

I was planning to code up the weight decay, when I noticed that I had left the transformer training. When I checked on it, it had already learned the rule, even without weight decay.

How is this possible? Unclear. Maybe: this solution is the most stable, so it naturally fell into place...(?)

The transformer has achieved near-perfect results with a very complicated heatmap. True understanding came later.

Experimenting with transformers

We will come back to heatmaps in a second, but let's focus on what tasks the transformer can learn easily, and what tasks it struggles on.

Try to predict the success rate of Makemore if we train it on the following tasks for 15,000 steps! In each case, we will start with a dataset of 100,000 random strings of 0's and 1's, with a random length between 50 and 200, with one extra rule.

1. The 18th and 47th digits are equal.
2. The 18th digit is equal to the digit at position $n - 20$, where n is the length of the string.
3. The last two digits are equal.
4. $s[18] = s[39], s[31] = s[44], s[4] = s[28]$
5. $s[18] = s[39], s[31] = s[44], s[4] = s[-28]$

Results after 15,000 steps

1. The 18th and 47th digits are equal: **99%**
2. The 18th digit is equal to the digit at position $n - 20$, where n is the length of the string: **54%**
3. The last two digits are equal: **90%**
4. $s[18] = s[39], s[31] = s[44], s[4] = s[28]$: **99%**
5. $s[18] = s[39], s[31] = s[44], s[4] = s[-28]$: **51%**

After 150,000 steps everything gets above 99%.

Lessons learned

- Transformers can easily learn to pay attention to fixed positions.
- $s[18] = s[-20]$ is difficult. The rule to learn is "at some point I need to put the same digit as the 18th, and 20 steps later I need to produce End Of Word token".
- "Last two digits are equal" is a moderately hard problem. I don't fully understand why.

Let's build more intuition!

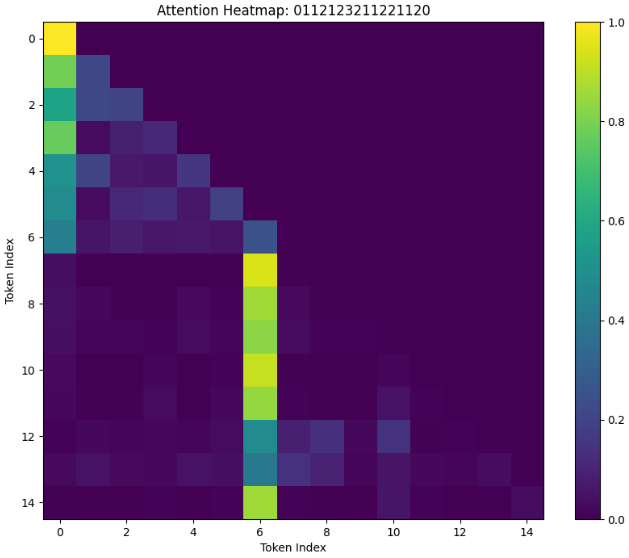
Dataset: random 0-1 strings of random length between 10 and 15, that also contain a unique "2" in a random position.

We train a transformer on this for a long time. What will the attention heatmap look like?

Hint 1: If we have generated only 0's and 1's so far in the first seven digits, which previous positions should the eighth digit pay attention to?

Hint 2: If the 8th digit is a 2, which previous digit should the 10th digit pay attention to?

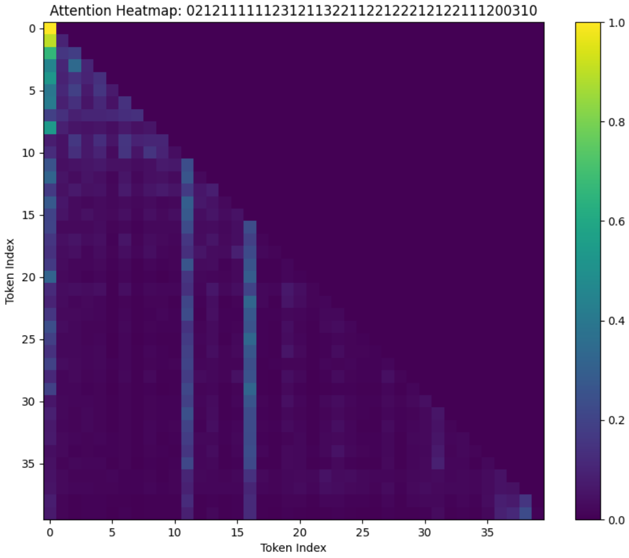
Heatmap: exactly one 2



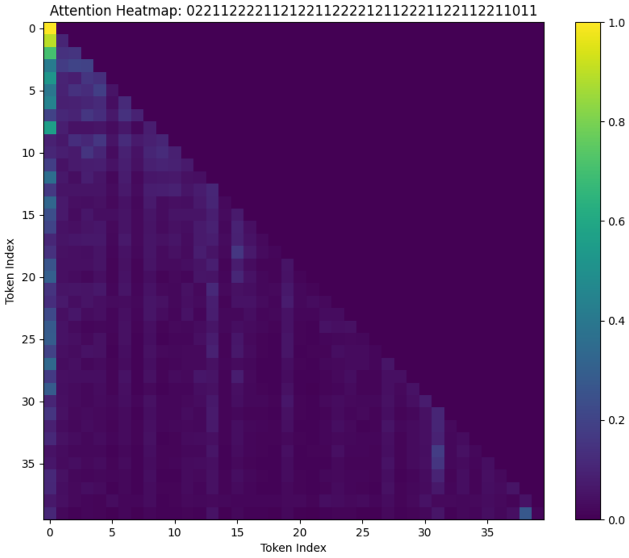
Let’s build more intuition!

Dataset: random 0-1 strings of random length between 30 and 40. With 50% chance we change precisely two randomly chosen digits to 2’s.

Heatmap: exactly zero or two 2's



Heatmap: exactly zero or two 2's



More toy examples

Can we generate Fibonacci-like sequences?

Input: first 7 terms of such sequences, as strings of '0'-'9' and ','

444,148,592,740,1332,2072,3404
231,355,586,941,1527,2468,3995
688,472,1160,1632,2792,4424,7216
19,829,848,1677,2525,4202,6727
355,404,759,1163,1922,3085,5007
526,183,709,892,1601,2493,4094
53,707,760,1467,2227,3694,5921
873,431,1304,1735,3039,4774,7813
...

What does the transformer see?

Note that the transformer doesn't have the ability to know that "5" is the number 5. It's just a symbol for them. The result would be the same if we input the strings

ššš‡ζš$‡º®ϒ‡Δš†‡ζ ≤≤ ϒ‡ϒ†Δϒ‡ ≤š†š
ϒ ≤ ζ‡ ≤ºº‡º$ő‡®šζ‡ζºϒΔ‡ϒšő$‡ ≤®®º
ő$$‡šΔϒ‡ζζő†‡ζő≤ ϒ‡ϒΔ®ϒ‡ššϒš‡Δϒζő
ζ®‡$ϒ®‡vs$‡ζőΔΔ‡ϒºϒº‡šϒ†ϒ‡őΔϒΔ

etc.

Fibonacci sequences: result

Output (transformer with 27k parameters, one hour of training):

```
932,751,1635,2406,4192,6407,10063
872,688,1530,2298,3759,6076,9898
873,83,955,1066,2969,3776,4287
303,186,488,687,1109,1827,3083
574,299,871,1116,1937,3165,5112
38,939,979,1922,2270,4043,7854
378,165,542,710,1265,1943,3270
972,700,1677,2307,4055,6371,10140
225,548,779,1353,2114,3481,5559
...
```

Looks vaguely like addition!

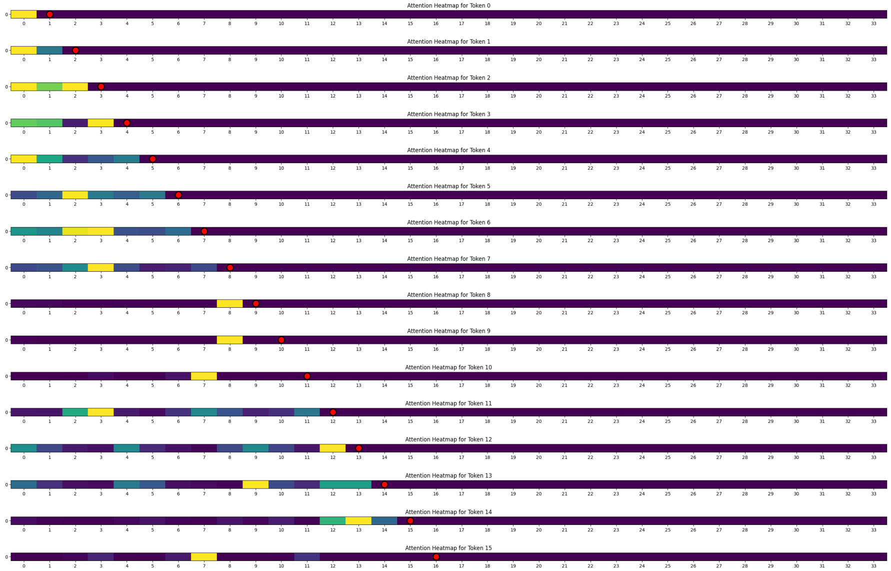
Fibonacci sequences: result

Output (transformer with 400k parameters, one day of training):

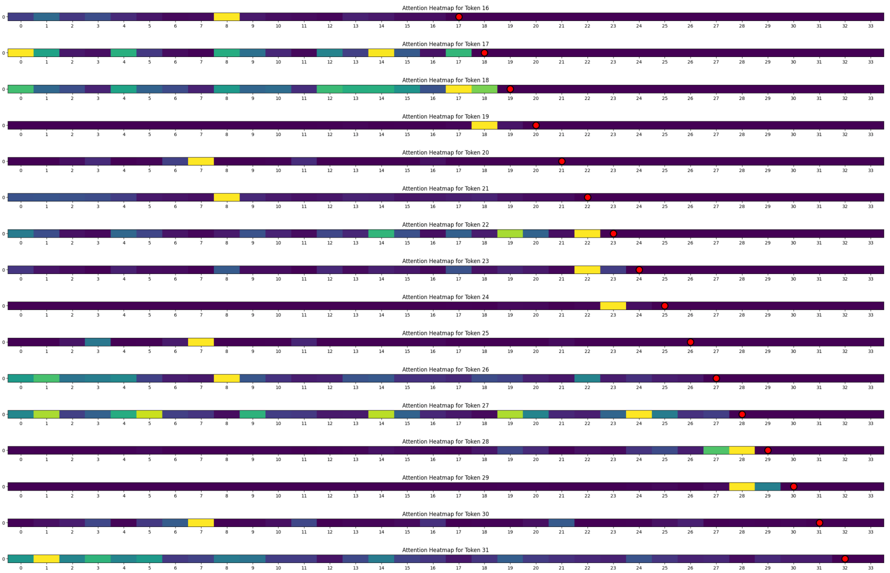
```
337,662,999,1661,2660,4321,6981
113,851,964,1815,2779,4594,7373
970,450,1420,1870,3290,5160,8450
80,182,262,444,706,1150,1856
57,945,1002,1947,2949,4896,7845
851,342,1193,1535,2728,4263,6991
374,832,1206,2038,3244,5282,8526
353,59,412,471,883,1354,2237
291,459,750,1209,1959,3168,5127
450,851,1301,2152,3453,5605,9058
...
```

All correct! But there are probably easier ways to do addition with a computer.

Fibonacci heatmaps



Fibonacci heatmaps



Interpreting the results

The transformer above cannot really be interpreted yet. It has learned some very complicated rules.

Weight decay tells the transformer to learn the "easiest rule possible", eventually.

Watch the talk by Neel Nanda called "A Walkthrough of Reverse-Engineering Modular Addition" for more info about grokking and understanding the functions learned by transformers

More computations with transformers

We’ve seen that transformers can learn to do some basic addition. What other types of calculations can we teach them?

Turns out, they can calculate gcd-s, do matrix multiplication, matrix inversion, and even calculate eigenvalues (for small matrices)! See the beautiful work of François Charton.

He also shows that when transformers get the results wrong, they still show a lot of understanding (they do not ”hallucinate absurd solutions”).

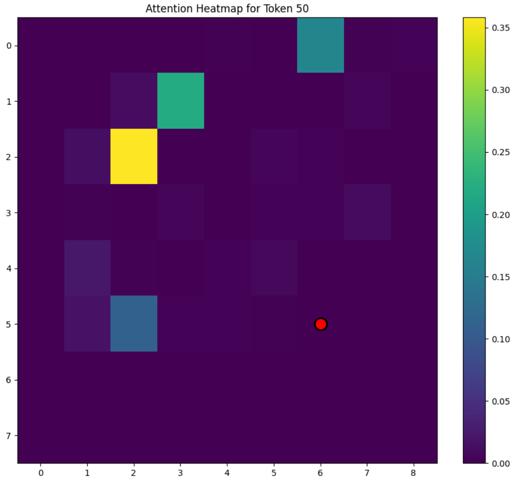
Time for a Live Experiment!

Let’s build more intuition!

Dataset: 0-1 sequences of length of length 64, that are such that if you organize them in a 8x8 grid, there are no two 1’s at Euclidean distance 4 or 5 from each other.

Hint: the algorithm used to produce this dataset matters. We generated these sequences sequentially. If we are allowed to put a 1 in the *i*-th position, we do so with probability 50%, otherwise we put a 0.

Heatmap: no distance 4 or 5



The transformer was never told that we are generating an 8x8 board, or what distances are! We just gave it a bunch of strings of length 64.

How can we use transformers in maths research?

We can teach transformers a bunch of things, but how can we use it in research?

Nobody really knows...
...but let's play with some ideas.

Transformers struggle on some tasks that are easy for humans, but they are able to learn patterns that are difficult for humans to understand.

Fibonacci part 2

Let's modify the Fibonacci experiment, and now start with this dataset:

521,142,663,679,13
248,602,750,32551,33301
855,244,1099,41551,
27,881,907,95591,96
738,518,1256,55491,567
686,854,1540,556552,558092

x_1 and x_2 are random numbers.
x_3 is obtained by summing x_1 and x_2, and replacing all 7's by 8's, and all 8's by 7's in the result.
x_4 is obtained by adding x_2 and x_3, adding 171, writing the result backwards and doubling all 5's.
x_5 is just $x_3 + x_4$, but we only write it out until the first occurrence of a 4.

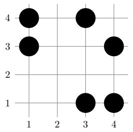
Transformer was able to learn this rule completely overnight.

How can we use transformers in maths research?

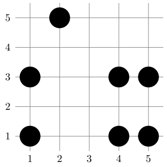
Let's pick a problem in maths where there is a mysterious pattern that we don't understand, and see if transformers can understand it better.

We will try to solve the following beautiful problem of Jordan Ellenberg: how many points can we choose in the $N \times N$ grid, without creating any isosceles triangles?

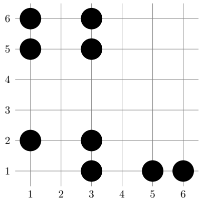
The isosceles triangle problem



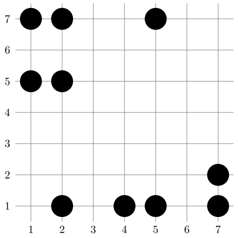
(a) $f(4) = 6$



(b) $f(5) = 7$

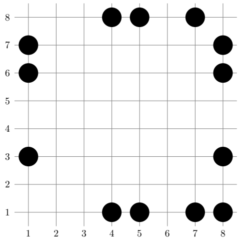


(c) $f(6) = 9$

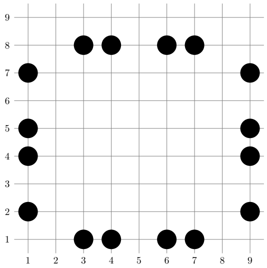


(d) $f(7) = 10$

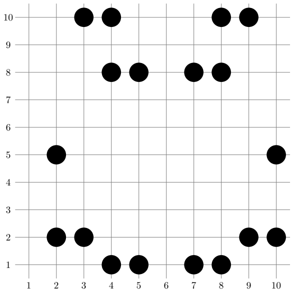
The isosceles triangle problem



(a) $f(8) = 13$



(b) $f(9) = 16$



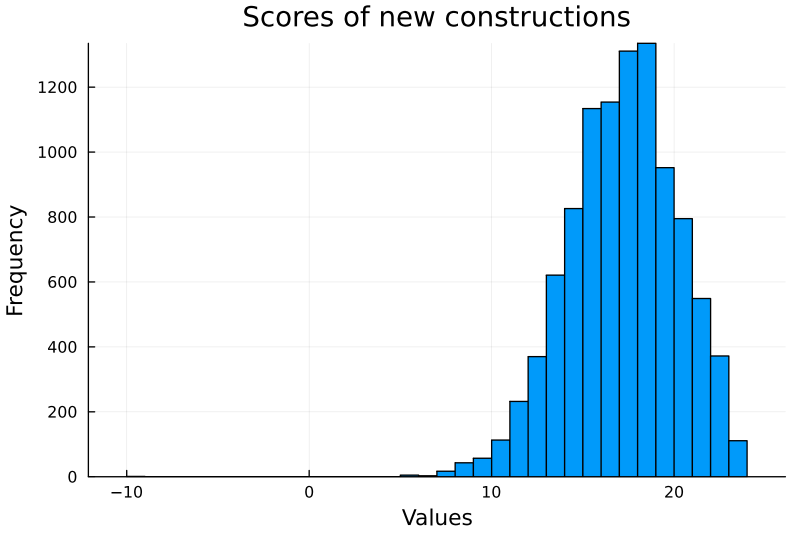
(c) $f(10) = 18$

How can we use transformers in maths research?

Imagine we have found the best constructions up to $n = 15$, but are struggling to find the best construction for $n = 16$.

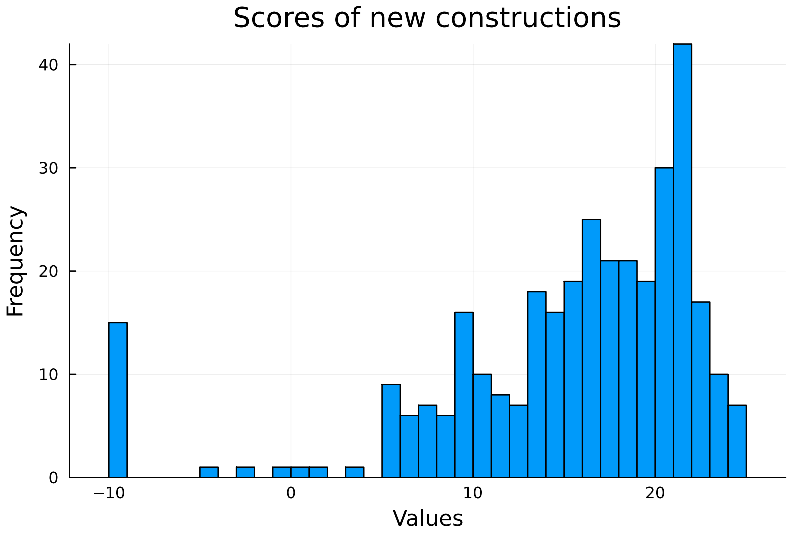
Idea: let's input all our attempts at $n = 16$ to the transformer, and get it to generate more attempts!

Scores of input constructions



Mean $=$ 16.7, Median $=$ 17, Avg of top 20% $=$ 20.7, max $=$ 23

Scores of new constructions produced by transformer



Mean = 15.9, Median = 17, Avg of top 20% = 21.9, max = 24.
Score of -10 means that the output did not have the right format

Evaluating results

We showed the transformer our best attempts on $n = 16$ and asked it to 'create more attempts like these'.

The output dataset has roughly the same mean, median as input dataset.

The spread of the output is bigger (why?). As such, with just a few hundred newly generated 'attempts' we found better constructions than we had in the input!

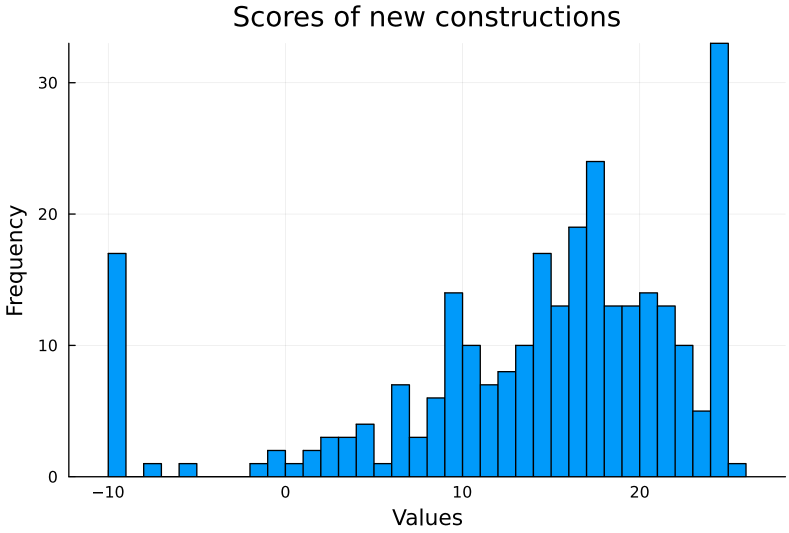
How to improve this score?

Let's add as an input also all the best constructions for $n < 16$!

We expect two things to happen:

1. Dataset is more confusing $\implies$ more illegal outputs, more outputs with low score
2. Model can learn some insights from smaller n values and apply it to $n = 16$ $\implies$ higher max score

Scores of new constructions produced by transformer



Mean = 15.4, Median = 16, Avg of top 20% = 23.4, max = 25.

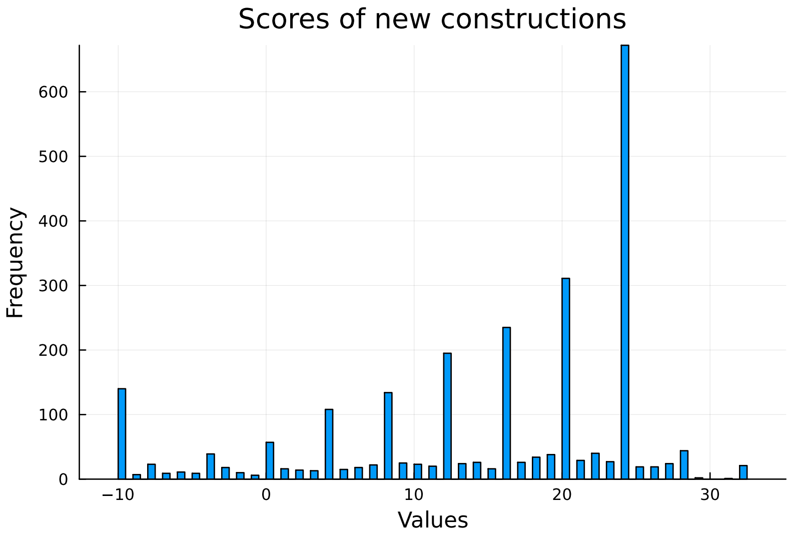
The isosceles problem

Great news! Inputting the best constructions for $n \leq 15$ has helped the transformer generate better scores for $n = 16$!

But the learning wasn't as flawless as in the previous experiments.

Let's try the same experiment for larger n. We input 100k "best" examples for $n \leq 20$, and 10k constructions for $n = 21$ with score 28. Here is the output after one night of training:

Result of experiment for larger n

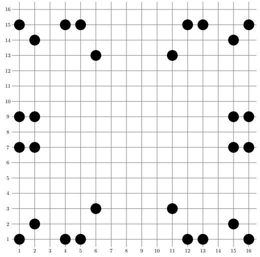


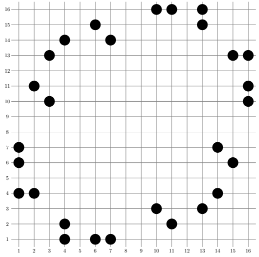
Evaluating the results

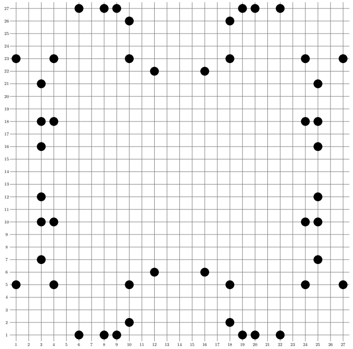
- Transformer has understood symmetry (hence the big spikes at scores divisible by 4)
- The model hasn't learned as much as in the previous experiment. Possible reasons:
  - Transformer not large enough
  - Not enough training

How can we improve things? (If we don't have the money to buy better GPUs?) Let's have a look at some of the better constructions we have generated so far.

The isosceles triangle problem


(a) $f(16) \geq 28$


(b) $f(16) \geq 28$


(c) $f(27) \geq 48$

How to improve our results?

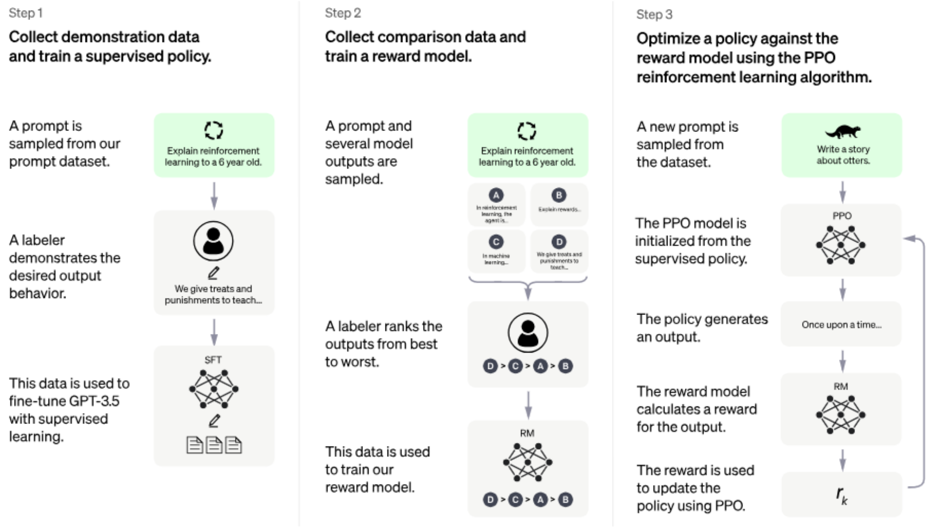
Imbalance of frequency of 0's and 1's, plus big hole in the middle $\Longrightarrow$ **tokenization** might help!

If you train a transformer on a big chunk of the internet, it will be good at babbling random internet-like text. If we input a question, we might get a news article with that title, or a list of more questions, etc – undefined behavior.

In order for it to become ChatGPT, researchers had to *align* it, or fine-tune it. This is the second stage to training good transformers.

We have a transformer that has understood something about the pattern. Now we have to teach it to be a good assistant.

Finetuning ChatGPT



Step 1

Collect demonstration data and train a supervised policy.

A prompt is sampled from our prompt dataset.

A labeler demonstrates the desired output behavior.

This data is used to fine-tune GPT-3.5 with supervised learning.

Step 2

Collect comparison data and train a reward model.

A prompt and several model outputs are sampled.

A labeler ranks the outputs from best to worst.

This data is used to train our reward model.

Step 3

Optimize a policy against the reward model using the PPO reinforcement learning algorithm.

A new prompt is sampled from the dataset.

The PPO model is initialized from the supervised policy.

The policy generates an output.

The reward model calculates a reward for the output.

The reward is used to update the policy using PPO.

Explain reinforcement learning to a 6 year old.
We give treats and punishments to teach...
SFT
Explain reinforcement learning to a 6 year old.
A
In reinforcement learning, the agent is...
B
Explain rewards...
C
In machine learning...
D
We give treats and punishments to teach...
D > C > A > B
RM
D > C > A > B
Write a story about otters.
PPO
Once upon a time...
RM
r_k

Other ways to improve results

- Give feedback on which new constructions are good or bad: trained transformers are very receptive to learning from new examples
- Vision transformers
- Graphormers
- Play with weight decay to improve generalization

Exploring the Future of Transformers in Mathematics

- We've delved into fun new ways to use Transformers to make progress on an open problem in mathematics.
- We haven't solved the problem, but we learned useful lessons along the way
- The truth is, nobody really knows what is the right way to use this technology in maths
- But therein lies the beauty of exploration: we are in a completely uncharted territory here
- I encourage you to experiment with new ideas to make this fascinating technology work in mathematics!

Exercise

Start with the dataset containing all your best constructions for the no-3-in-line problem. Can you improve them by using makemore?

Thank you!