Transformers for Mathematics

Simons and SLMath Workshop on AI for Mathematics and Theoretical CS

Sean Welleck

April 8, 2025

github.com/wellecks/transformers4math-simons

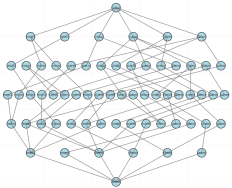
Carnegie Mellon University

AI for Mathematics

System	Lyapunov function
$\begin{cases} \dot{x_0} = -5x_0^3 - 2x_0x_1^2 \\ \dot{x_1} = -9x_0^4 + 3x_0^3x_1 - 4x_1^3 \end{cases}$	$V(x) = 6x_0^6 + 7x_0^4 + x_0^3 + 10x_0^2 + 8x_1^2$
$\begin{cases} \dot{x_0} = -x_0^5 - 4x_0^3 - 9x_0x_1^4 + 3x_0x_1^3 \\ \dot{x_1} = -3x_0^4x_1^2 - 10x_0^3x_1 + 3x_0x_1^2 - 7x_1^3 \end{cases}$	$V(x) = x_0^4 + 9x_0^2 + 3x_1^2$
$\begin{cases} \dot{x_0} = -3x_0^3 + 3x_0x_2 - 9x_0 \\ \dot{x_1} = -x_0^3 - 5x_1 + 5x_2^2 \\ \dot{x_2} = -9x_2^3 \end{cases}$	$V(x) = x_0^4 + 7x_0^2x_2^2 + 3x_0^2 + 4x_0x_2^2 + 3x_1^2 + 2x_2^4 + 10x_2^2$
$\begin{cases} \dot{x_0} = -8x_0x_1^2 - 10x_1^4 \\ \dot{x_1} = -8x_1^3 + 3x_1^2 - 8x_1 \\ \dot{x_2} = -x_2 \end{cases}$	$V(x) = 4x_0^2 - 2x_0x_1^2 + 6x_1^4 + 4x_1^2 + x_2^2$

Discovering Lyapunov functions

[Alfarano et al 2024]



Finding counterexamples

[Charton et al 2024]

```
 Try this:
• 🎉 rw [entropy_cond_eq_sum hX μ y]
   apply tsum_eq_sum
   intro s hs
   convert negMulLog_zero
   convert ENNReal.zero_toReal
   convert measure_mono_null _ (ful
   simp [hs]
```

Assisting in proofs

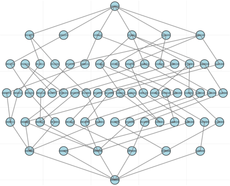
[LLMLean 2024]

AI for Mathematics

System	Lyapunov function
$\begin{cases} \dot{x_0} = -5x_0^3 - 2x_0x_1^2 \\ \dot{x_1} = -9x_0^4 + 3x_0^3x_1 - 4x_1^3 \end{cases}$	$V(x) = 6x_0^6 + 7x_0^4 + x_0^3 + 10x_0^2 + 8x_1^2$
$\begin{cases} \dot{x_0} = -x_0^5 - 4x_0^3 - 9x_0x_1^4 + 3x_0x_1^3 \\ \dot{x_1} = -3x_0^4x_1^2 - 10x_0^3x_1 + 3x_0x_1^2 - 7x_1^3 \end{cases}$	$V(x) = x_0^4 + 9x_0^2 + 3x_1^2$
$\begin{cases} \dot{x_0} = -3x_0^3 + 3x_0x_2 - 9x_0 \\ \dot{x_1} = -x_0^3 - 5x_1 + 5x_2^2 \\ \dot{x_2} = -9x_2^3 \end{cases}$	$V(x) = x_0^4 + 7x_0^2x_2^2 + 3x_0^2 + 4x_0x_2^2 + 3x_1^2 + 2x_2^4 + 10x_2^2$
$\begin{cases} \dot{x_0} = -8x_0x_1^2 - 10x_1^4 \\ \dot{x_1} = -8x_1^3 + 3x_1^2 - 8x_1 \\ \dot{x_2} = -x_2 \end{cases}$	$V(x) = 4x_0^2 - 2x_0x_1^2 + 6x_1^4 + 4x_1^2 + x_2^2$

Discovering Lyapunov functions

[Alfarano et al 2024]



Finding counterexamples

[Charton et al 2024]

```
 Try this:
• 🎉 rw [entropy_cond_eq_sum hX μ y]
   apply tsum_eq_sum
   intro s hs
   convert negMulLog_zero
   convert ENNReal.zero_toReal
   convert measure_mono_null _ (ful
   simp [hs]
```

Assisting in proofs

[LLMLean 2024]

Common building block: generating sequences with *Transformers*

Overview

In this tutorial:

1. What is a transformer?
2. Examples of using transformers on mathematical data
3. Interactive notebook sessions

Overview

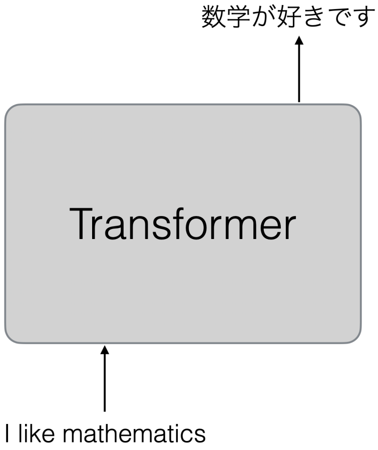
In this tutorial:

1. What is a transformer?
2. Examples of using transformers on mathematical data
3. Interactive notebook sessions

Notebooks: github.com/wellecks/transformers4math-simons

Thank you to Adam Zsolt Wagner for help in putting together this tutorial.

- Transformers are a neural network architecture.
- Key part of large language models (LLMs) and generative AI in many domains.
- Maps an input sequence to an output sequence.

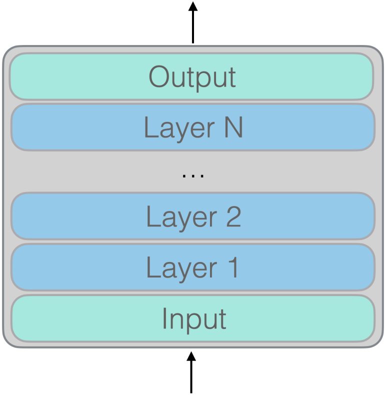


Introduction to Transformers

Transformers consist of multiple "layers" stacked together.

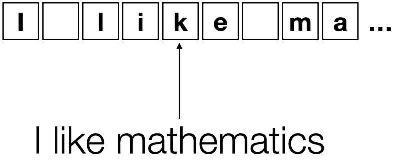
Key components:

- **Token embeddings**
- **Self-attention mechanism**



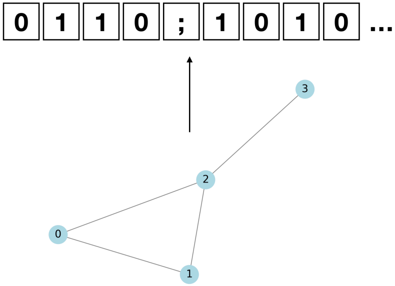
Representing inputs: tokenization

- *Tokenization*: represent an input as a sequence of discrete *tokens*
    - Example: a sentence as a sequence of characters



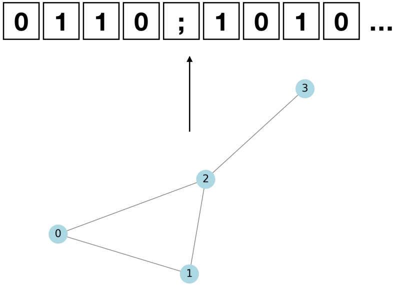
Representing inputs: tokenization

- *Tokenization*: represent an input as a sequence of discrete *tokens*
    - Example: a sentence as a sequence of characters
    - Example: a graph as an adjacency matrix string



Representing inputs: tokenization

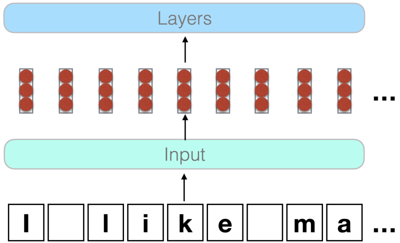
- *Tokenization*: represent an input as a sequence of discrete *tokens*
    - Example: a sentence as a sequence of characters
    - Example: a graph as an adjacency matrix string



Takeaway: transformers are general purpose! Just find a way to treat your input as a sequence of tokens and the transformer will accept it.

Representing inputs: token embeddings

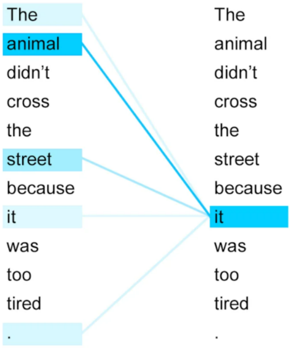
- Each token is associated with a vector called a *token embedding*.
- The token embeddings are passed to main layers of the transformer.



Main layers: *self-attention*

The **attention mechanism** is the fundamental building block of transformer layers.

- Lets the model focus on different parts of the input sequence when making predictions.
- Attention assigns different weights to each input element.
- These weights determine how much each element contributes to the output.



Attention

Why do we need attention?

Consider the bigram model:

- Predicts the next letter based only on the previous letter.
- Example: when trained on a list of names, it produces names that lack coherence and context.
- To improve this, we could take the average of all previous token embeddings.
- But taking a simple average may not capture context well.

```
['mmin',
 'jayvi',
 'm',
 'orynahaninarsaronzaiy',
 'bireli',
 'caiviron',
 'mamizanasshon',
 'i',
 'cka',
 'azemanartlougialecyamaitilee']
```

Names generated by the bigram model (`bigrams.ipynb`)

Why do we need attention?

Consider the bigram model:

- Predicts the next letter based only on the previous letter.
- Example: when trained on a list of names, it produces names that lack coherence and context.
- To improve this, we could take the average of all previous token embeddings.
- But taking a simple average may not capture context well.

```
['mmin',
 'jayvi',
 'm',
 'orynahaninarsaronzaiy',
 'bireli',
 'caiviron',
 'mamizanasshon',
 'i',
 'cka',
 'azemanartlougialecyamaitilee']
```

Names generated by the bigram model (`bigrams.ipynb`)

Takeaway: attention mechanisms *learn* to effectively use context.

Self-Attention: A Closer Look

- In self-attention, each token in the input sequence emits a **query** vector and a **key** vector.
  - 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.

Self-Attention: A Closer Look

- In self-attention, each token in the input sequence emits a query vector and a key vector.
  - 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:

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

Self-Attention: A Closer Look

- In self-attention, each token in the input sequence emits a **query** vector and a **key** vector.
  - 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:

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

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.

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:

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

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.

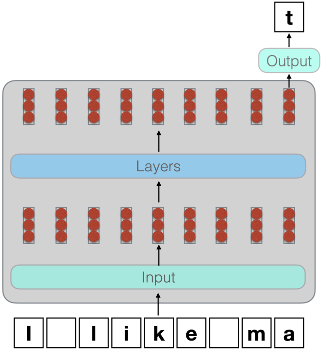
Queries, Keys, and Values

- Recap: In self-attention, we have three vectors 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.

Let's do something useful!

We still need to train the model so that it learns vector representations that are useful for a task. Example:

- **Language modeling**: make the vector at each position useful for predicting the next token

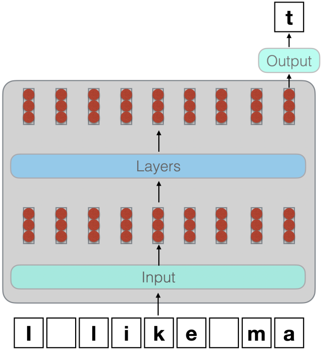


Language modeling

Let's do something useful!

We still need to train the model so that it learns vector representations that are useful for a task. Example:

- **Language modeling**: make the vector at each position useful for predicting the next token
  - Introduce an output layer that maps the vector to a probability for each possible next-token

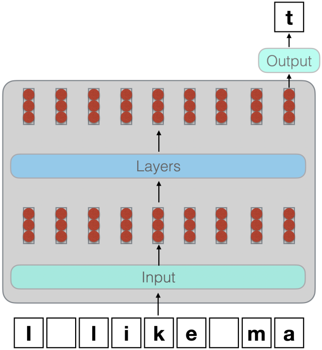


Language modeling

Let’s do something useful!

We still need to train the model so that it learns vector representations that are useful for a task. Example:

- **Language modeling**: make the vector at each position useful for predicting the next token
  - Introduce an output layer that maps the vector to a probability for each possible next-token
  - Minimize a loss that makes the next-token that appears in the training sequence more probable



Language modeling

Let's do something useful – language modeling

To train the transformer language model we supply a training set of sequences. The model is trained to predict each next token, i.e., increase the token's probability given the previous tokens.

```
['miko',
 'tenuun',
 'issachar',
 'katana',
 'amelie',
 'doha',
 'burk',
 'daran',
 'ryelan',
 'avelino']
```

Examples from the training set (`transformer.ipynb`)

Let's do something useful – language modeling

After training, we can generate new sequences with the language model. We generate one token at a time according to the model's next-token probabilities.

```
maryna
inattin
jairem
naisel
pesen
payceo
zachia
vilery
poana
```

Transformer-generated names that don't appear in the training set

Let’s do something useful – language modeling

After training, we can generate new sequences with the language model. We generate one token at a time according to the model’s next-token probabilities.

```
maryna
inattin
jairem
naisel
pesen
payceo
zachia
vilery
poana
```

Transformer-generated names that don’t appear in the training set

Enough names! Let’s generate mathematical things

Some math-related applications

- Toy examples
- Computations
- Discovery

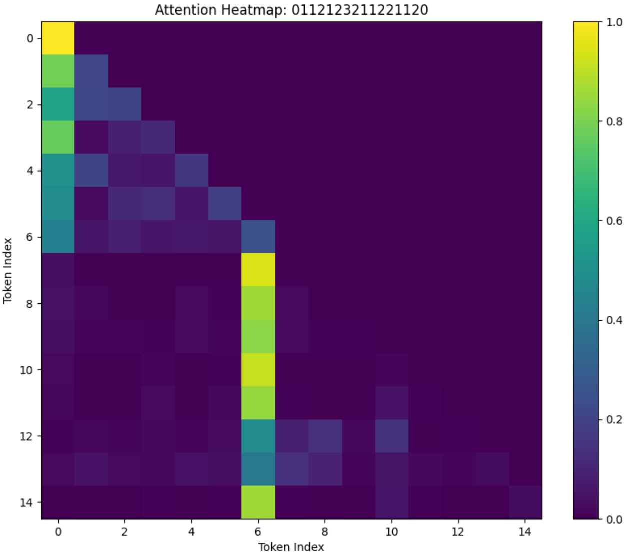
Toy examples: let's build some intuition!

Dataset: random {0,1,2} strings of random length between 10 and 15, that also contain a unique "3" in a random position.

- 110201032221
- 20232212010
- 0112123211221120

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

Heatmap: exactly one 3



Toy examples: let's build some intuition!

Dataset: each sequence is:

- a name
- the count of each unique character that occurs in the name

anna$\rightarrow$a2n2
simons$\rightarrow$s2i1m1o1n1
...

Toy examples: let's build some intuition!

- Let's generate 1000 sequences with the trained model:
  - 96.6% correct
  - 65.5% novel (not in training set)

```
['joangelo$j1o2a1n1g1e1l1',
 'salen$s1a1l1e1n1',
 'larbella$l3a2r1b1e1',
 'jilianna$j1i2l1a2n2',
 'jeliani$j1e1l1i2a1n1',
 'elysen$e2l1y1s1n1',
 'laniish$l1a1n1i2s1h1',
 'narvio$n1a1r1v1i1o1',
 'andersi$a1n1d1e1r1s1i1',
 'aldio$a1l1d1i1o1']
```

Example correct and novel generations

Toy examples: let's build some intuition!

- Let's generate 1000 sequences with the trained model:
  - 96.6% correct
  - 65.5% novel (not in training set)

```
['joangelo$j1o2a1n1g1e1l1',
 'salen$s1a1l1e1n1',
 'larbella$l3a2r1b1e1',
 'jilianna$j1i2l1a2n2',
 'jeliani$j1e1l1i2a1n1',
 'elysen$e2l1y1s1n1',
 'laniish$l1a1n1i2s1h1',
 'narvio$n1a1r1v1i1o1',
 'andersi$a1n1d1e1r1s1i1',
 'aldio$a1l1d1i1o1']
```

Example correct and novel generations

Takeaway: a bit noisy (96.6%), but it learns the pattern pretty well and is not just memorizing the training data!

Toy examples: let's build some intuition!

Accuracy on held out names: 97.3%

Most failure cases:

- Characters with count 3 or 4
  - `evelee`$\rightarrow$`e`<u>`3`</u>`v1l1`
- Model forgets to generate a character's count
  - `stockto`<u>`n`</u>$\rightarrow$ `s1t2o2c1k1`

Toy examples: let's build some intuition!

Accuracy on held out names: 97.3%

Adversarial cases:

- `aaa` → `ab1b1a2`
- `s3an` → `a1a1a1a1a1a1a1a1`
- ...

Some math-related applications

- Toy examples
- Computations
- Discovery

Some math-related applications

- Toy examples
- Computations
- Discovery

Simple computations: addition

Add two 4-digit integers:

- `2871+9281=12152`
- `1000+1000=2000`
- …
- Around 40 million unique unordered pairs.

Simple computations: addition

Add two 4-digit integers:

- `2871+9281=12152`
- `1000+1000=2000`
- ...
- Around 40 million unique unordered pairs.

Plan:

- Train a transformer language model on a dataset of 5 million addition problems.
- Evaluate it on problems that aren't in the training set.

Simple computations: addition

- Test accuracy: 100.0%
- It can generate full problems that aren't in the training set:

```
2036+2045=4081
2478+2311=4789
4024+4833=8857
8577+1026=9603
3619+3435=7054
5685+8368=14053
3106+6791=9897
```

New addition problems

Simple computations: addition

- Test accuracy: **100.0%**
- It can generate full problems that aren't in the training set:

```
2036+2045=4081
2478+2311=4789
4024+4833=8857
8577+1026=9603
3619+3435=7054
5685+8368=14053
3106+6791=9897
```

New addition problems

Can transformers learn some non-trivial computations?

Symbolic integration [Lample & Charton 2019]

Lample and Charton trained a transformer to take in an equation as a sequence of tokens, and output its integral as a sequence of tokens.

Equation	Solution
$y' = \dfrac{16x^3 - 42x^2 + 2x}{(-16x^8 + 112x^7 - 204x^6 + 28x^5 - x^4 + 1)^{1/2}}$	$y = \sin^{-1}(4x^4 - 14x^3 + x^2)$
$3xy\cos(x) - \sqrt{9x^2\sin(x)^2 + 1}y' + 3y\sin(x) = 0$	$y = c\exp\left(\sinh^{-1}(3x\sin(x))\right)$
$4x^4yy'' - 8x^4y'^2 - 8x^3yy' - 3x^3y'' - 8x^2y^2 - 6x^2y' - 3x^2y'' - 9xy' - 3y = 0$	$y = \dfrac{c_1 + 3x + 3\log(x)}{x(c_2 + 4x)}$

Problems that the transformer was able to solve, on which Mathematica and Matlab were not able to find a solution.

Symbolic integration [Lample & Charton 2019]

Lample and Charton trained a transformer to take in an equation as a sequence of tokens, and output its integral as a sequence of tokens.

Equation	Solution
$y' = \dfrac{16x^3 - 42x^2 + 2x}{(-16x^8 + 112x^7 - 204x^6 + 28x^5 - x^4 + 1)^{1/2}}$	$y = \sin^{-1}(4x^4 - 14x^3 + x^2)$
$3xy\cos(x) - \sqrt{9x^2\sin(x)^2 + 1}\,y' + 3y\sin(x) = 0$	$y = c\exp\left(\sinh^{-1}(3x\sin(x))\right)$
$4x^4yy'' - 8x^4y'^2 - 8x^3yy' - 3x^3y'' - 8x^2y^2 - 6x^2y' - 3x^2y'' - 9xy' - 3y = 0$	$y = \dfrac{c_1 + 3x + 3\log(x)}{x(c_2 + 4x)}$

Problems that the transformer was able to solve, on which Mathematica and Matlab were not able to find a solution.

Each integral can be **verified** by taking its derivative. As a result, we can **generate multiple candidates** and discard the incorrect ones.

Beyond integration

The recipe of synthesizing data and training a transformer language model has been explored for several other prediction problems:

Beyond integration

The recipe of synthesizing data and training a transformer language model has been explored for several other prediction problems:

- **Solutions to 1st and 2nd-order ODEs** [Lample & Charton 2019]

Input:

$$162x\log(x)y' + 2y^3\log(x)^2 - 81y\log(x) + 81y = 0$$

Generated solutions:

$$\frac{9\sqrt{x}\sqrt{\frac{1}{\log(x)}}}{\sqrt{c+2x}}$$

$$\frac{9}{\sqrt{\frac{c\log(x)}{x} + 2\log(x)}}$$

$$9\sqrt{x}\sqrt{\frac{1}{c\log(x) + 2x\log(x) + \log(x)}}$$

$$\ldots$$

Beyond integration

The recipe of synthesizing data and training a transformer language model has been explored for several other computations:

- Solutions to 1st and 2nd-order ODEs [Lample & Charton 2019]
- Properties of differential systems (e.g., stability) [Charton et al 2021]
- Global Lyapunov functions [Alfarano et al 2024]
- Frobenius traces from elliptic curves [Babei et al 2025]
- ...

Beyond integration

What problems are suitable? Some common themes:

- **Verifiable**. E.g. we can often verify that $\int f = g$ by taking its derivative with a computer algebra system.
- **Data generator**. A lot of ingenuity goes into creating a data generator, e.g. synthesizing random functions and pairing a function with its derivative

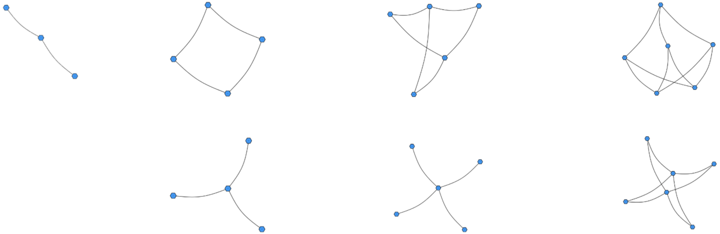
Some math-related applications

- Toy examples
- Computations
- Discovery

Generating constructions (based on [Charton et al 2024 [1]]

Problem

How many edges can an n-vertex graph have, if no three edges form a triangle?



Example triangle-free graphs for $n = 3$ to 6. From [1].

Generating constructions

Plan: generate many graphs that satisfy this triangle-free property

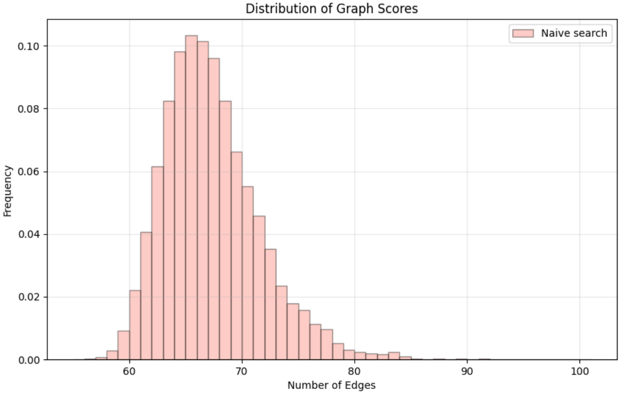
Generating constructions

Plan: generate many graphs that satisfy this triangle-free property

- **Naive search**: keep randomly deleting an edge that is in the maximum number of triangles. Then keep adding random edges that don't introduce a triangle.

$$\text{score}(G) = \#\text{edges}(G) \tag{1}$$

Generating constructions

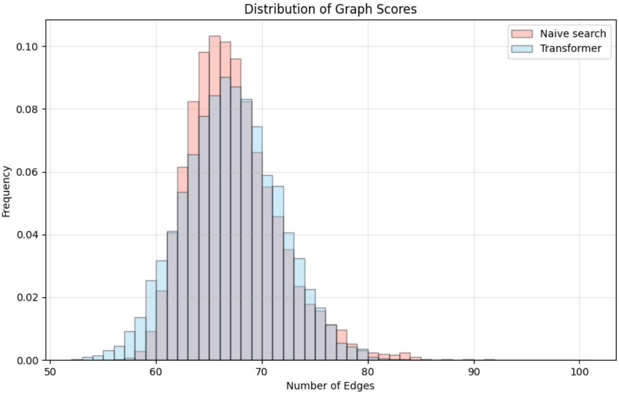


Score distribution for 100,000 graphs found by the naive search

Generating constructions

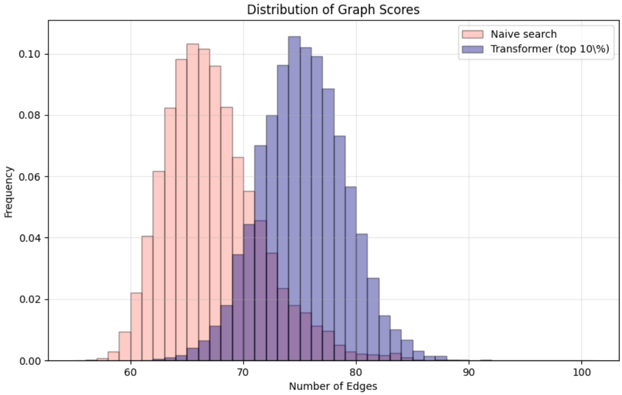
Let's represent each adjacency matrix as a string, and a train a transformer!

Generating constructions



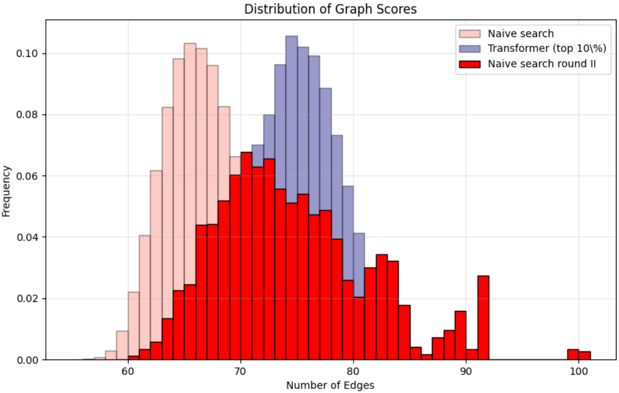
The transformer learns to generate graphs that have a similar score distribution.

Generating constructions



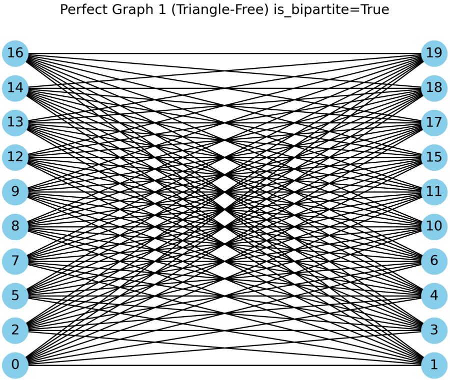
Instead, train the transformer on the top 10% of graphs from the local search.

Generating constructions



Running local search from this transformer's generated constructions results in several high-scoring graphs (score 100)

Generating constructions



The score 100 graphs are isomorphic, and are bipartite graphs

PatternBoost [Charton-Ellenberg-Wagner-Williamson 2024]

PatternBoost by Charton-Ellenberg-Wagner-Williamson generalizes this idea. It alternates between:

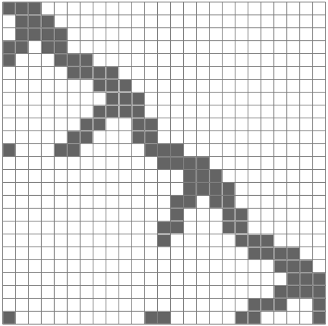
- **Local phase**: classical search algorithm produces constructions
- **Global phase**: train a Transformer on the best constructions, then generate candidates that are passed to the next local phase

They apply it to several non-trivial problems.

PatternBoost [Charton-Ellenberg-Wagner-Williamson 2024]

Question (Brualdi–Cao)

How large can the permanent of a binary $n \times n$ matrix be, that does not contain the pattern 312?

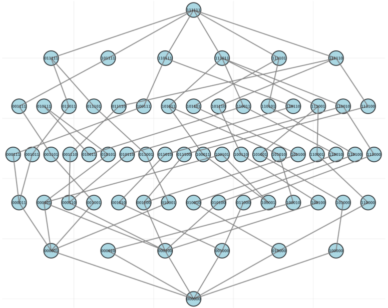


Example construction for $n = 25$ with permanent 5,101,230. The initial local search gets 641,000.

PatternBoost [Charton-Ellenberg-Wagner-Williamson 2024]

Conjecture (Graham & Harary 1992)

The maximum number of edges one can delete from the d-dimensional hypercube without increasing its diameter is $2^d + \binom{d}{\lfloor d/2 \rfloor} - 2$.



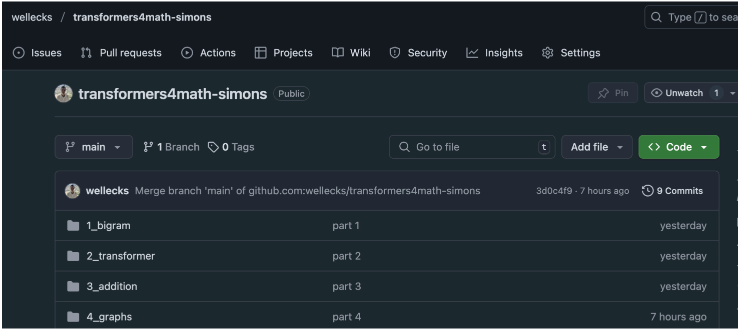
Counterexample discovered that has $d = 6$ with $81 < 2^6 + \binom{6}{3} - 2 = 82$ edges.

Recap

1. What is a transformer?
2. Examples of using transformers on mathematical data
    - Toy examples
    - Computations
    - Discovery
3. Interactive notebook sessions

Interactive notebook sessions

For the next two sessions we will provide you with interactive notebooks to work through.



https://github.com/wellecks/transformers4math-simons

Interactive notebook sessions

For the next two sessions we will provide you with interactive notebooks to work through.[1]

[1] https://github.com/wellecks/transformers4math-simons

Interactive notebook sessions

For the next two sessions we will provide you with interactive notebooks to work through.[1]

- **Transformers and language models**: see concretely how transformers and language models work.
  - `bigrams.ipynb`: the simplest possible language model.
  - `transformers.ipynb`: a transformer implementation and detailed code for training it as a language model.

[1] https://github.com/wellecks/transformers4math-simons

Interactive notebook sessions

For the next two sessions we will provide you with interactive notebooks to work through.[1]

- **Transformers and language models**: see concretely how transformers and language models work.
  - `bigrams.ipynb`: the simplest possible language model.
  - `transformers.ipynb`: a transformer implementation and detailed code for training it as a language model.

- **Makemore on mathematical sequences**: train a transformer using a library as a black box, and evaluate its generations.
  - `addition.ipynb`: train a transformer to do 4-digit addition as in the talk.
  - `graphs.ipynb`: train a transformer to generate triangle-free graphs and reproduce the plot from the talk.

[1] https://github.com/wellecks/transformers4math-simons

Thank you!

https://github.com/wellecks/transformers4math-simons

Special thank you to Leni Aniva, Jeremy Avigad, and Adam Wagner for helping with the preparation of this material.

Sean Welleck, CMU

`wellecks@cmu.edu`

F. Charton, J. S. Ellenberg, A. Z. Wagner, and G. Williamson.
Patternboost: Constructions in mathematics with a little help from ai, 2024.