LINEAR ALGEBRA WITH TRANSFORMERS

Anonymous authors
Paper under double-blind review

ABSTRACT

Most applications of transformers to mathematics, from integration to theorem proving, focus on symbolic computation. In this paper, we show that transformers can be trained to perform numerical calculations with high accuracy. We consider problems of linear algebra: matrix transposition, addition, multiplication, eigenvalues and vectors, singular value decomposition, and inversion. Training small transformers (up to six layers) over datasets of random matrices, we achieve high accuracies (over 90%) on all problems. We also show that trained models can generalize out of their training distribution, and that out-of-domain accuracy can be greatly improved by working from more diverse datasets (in particular, by training from matrices with non-independent and identically distributed coefficients). Finally, we show that few-shot learning can be leveraged to re-train models to solve larger problems.

1 INTRODUCTION

Since their introduction by Vaswani et al. (2017), transformers, originally designed for machine translation, were applied to various problems, from text generation (Radford et al., 2018; 2019) to image processing (Carion et al., 2020) and speech recognition (Dong et al., 2018) where they soon achieved state-of-the-art performance (Dosovitskiy et al., 2021; Wang et al., 2020b). In mathematics, transformers were used for symbolic integration (Lample & Charton, 2019), theorem proving (Polu & Sutskever, 2020), formal logic (Hahn et al., 2021), SAT solving (Shi et al., 2021), symbolic regression (Biggio et al., 2021) and dynamical systems (Charton et al., 2020). All these problems pertain to symbolic mathematics, or involve a large amount of symbolic computation. When working on these tasks, transformers manipulate mathematical symbols, just like words in natural language.

But mathematics are not limited to symbol manipulation: many practical applications involve numerical calculations, either exact (e.g. arithmetic) or approximate (e.g. function evaluation, numerical solutions of equations). The use of transformers for numerical computation has been less studied, and many early experiments with arithmetic have proved disappointing (Nogueira et al., 2021). This is, nevertheless, an important question: most problems in mathematics and science involve both symbolic and numerical computations. If we want transformers to solve these problems end-to-end, they need to be able to perform numerical calculations with high accuracy.

In this paper, we train transformers to compute solutions of problems of linear algebra: basic operations on matrices, matrix inversion, eigenvalue and singular value decompositions. These operations are fundamental building blocks for a large number of scientific problems. We introduce and discuss four encodings to represent problems and solutions as sequences that transformers can process, and train small transformers (up to 6 layers, 10 to 50 million trainable parameters) over generated datasets of random matrices. Trained models can compute approximate solutions to these problems (to a few percents of their L^1 norm) with over 90% accuracy (over 99% in most cases). We also show that they can generalize out of their training distribution, and be retrained to extrapolate to larger problems than the ones they were trained on. We believe these results pave the way for using transformers as end to end solvers for problems of mathematics and science.

After introducing the problems of linear algebra we are studying and presenting the encodings we use to represent them as sequences that can be used by our models, we discuss data generation, architecture and experimental settings. Then, we present our experiments on nine different problems, and discuss out-of-distribution generalization and few shot learning for eigenvalue computation. Finally, we discuss our results and future directions for research, and present related works.

2 PROBLEMS AND DATASETS

Let M and N be m×n matrices and $V \in \mathbb{R}^m$. We study nine problems of linear algebra:

- matrix transposition: find M^T, a $n \times m$ matrix,
- matrix addition: find $M + N$, a $m \times n$ matrix,
- matrix-vector multiplication: find $M^T V$, a vector in $\mathbb{R}^n$,
- matrix multiplication: find $M^T N$, a $n \times n$ matrix,
- eigenvalues: M symmetric, find its n (real) eigenvalues, sorted in descending order,
- eigenvectors: M symmetric, find D diagonal and Q orthogonal such that $M = Q^T DQ$, set as a $(n + 1) \times n$ matrix, with eigenvalues as its first line,
- singular values: find the n eigenvalues of $M^T M$, sorted in descending order,
- singular value decomposition: find orthogonal U, V and diagonal S such that $M = USV$, set as a $(m + n + 1) \times min(m, n)$ matrix,
- inversion: M square and invertible, find its inverse P, such that $MP = PM = Id$.

These problems range from basic operations on individual coefficients of the input matrices (transposition and addition), to computations involving several arithmetic operations over many coefficients (multiplication), to complex nonlinear transformations involving the whole matrix, with cubic complexity (decompositions and inversion). For each problem, we generate datasets of pairs of matrices (N, O), by sampling random input matrices N (see section 2.2), and computing the output O with a linear algebra package (NumPy linalg). When a problem has several input or output matrices, they are concatenated into one (for instance, the two $m \times n$ operands of the addition task are concatenated into one $m \times 2n$ matrix N). All coefficients in N and O are set in base ten floating-point representation, and rounded to three significant digits in the mantissa.

2.1 ENCODING MATRICES AS SEQUENCES

The input and output to our problems are matrices. To be processed by transformers, they need to be converted into sequences of tokens. We encode a $m \times n$ matrix by first coding its dimensions as two symbolic tokens (`Vm` and `Vn`), followed by its mn coefficients, encoded as sequences. Through this paper, we will use four encoding schemes for matrix coefficients: P10, P1000, B1999, and FP15.

In base 10 positional encoding (P10), a number is represented as a sequence of five tokens : one sign token (+ or −), 3 digits (from `0` to `9`) for the mantissa, and a symbolic token (from `E-100` to `E+100`) for the exponent. For instance 3.14 will be represented as 314.10^{-2}, and encoded as `[+, 3, 1, 4, E-2]`. P1000 (positional base 1000) provides a more compact representation by encoding the mantissa as a single token (from `0` to `999`), and representing a number as the triplet (sign, mantissa, exponent). B1999 (balanced base 1999) pushes this one step further by encoding together the sign and mantissa (from `-999` to `999`). Finally, FP15 encodes each floating point number $x = m10^b$ as a unique token `FPm/b` (with $b \in [-8, 8]$ and $m \in [-999, 999]$). Table 1 provides examples of encodings.

Encoding	3.14	$-6.02.10^{23}$	Tokens / coefficient	Vocabulary
P10	`[+, 3, 1, 4, E-2]`	`[-, 6, 0, 2, E21]`	5	210
P1000	`[+, 314, E-2]`	`[-, 602, E21]`	3	1100
B1999	`[314, E-2]`	`[-602, E21]`	2	2000
FP15	`[FP314/-2]`	`[FP-602/21]`	1	30000

Table 1: **Four encodings for matrix coefficients.**

Choicing an encoding is a trade-off. Long encodings (P10, P1000) embed knowledge about numbers. For instance, two numbers can be crudely compared by looking at their signs and mantissas, and multiplication and addition can be learned by memorizing small tables. Compact encodings, on the other hand, result in shorter sequences that are easier to learn with transformers. In P10, a 20×20 matrix is a sequence of 2002 tokens, close to the practical limit of most transformer implementations (that use a quadratic attention mechanism). In FP15, it is only 402 token long.

2.2 RANDOM MATRIX GENERATION

In most experiments, models are trained on datasets of random matrices with uniformly distributed coefficients over $[-A, A]$ (setting $A = 10$). Occasionally, we sample gaussian coefficients with the same standard deviation ($\sigma = A/\sqrt{3}$). In the symmetric case, these matrices are known as Wigner matrices. Their eigenvalues have a centered distribution with standard deviation $\sigma = \sqrt{n}s$, where s is the standard deviation of the coefficients ($s = A/\sqrt{3}$ in the uniform case) (Mehta, 2004). As n increases, the distribution converges to the semi-circle law ($p(\lambda) = \sqrt{4\sigma^2 - \lambda^2}/2\pi\sigma^2$) for all coefficient distributions with bounded variance. If the coefficients are gaussian, the associated eigenvectors are uniformly distributed over the unit sphere.

When investigating out-of-distribution generalization for the eigenvalue problem, we will need to generate random symmetric matrices with different distributions of their eigenvalues (corresponding to random matrices with non iid coefficients). To this effect, we randomly sample symmetric matrices with gaussian coefficients, compute their eigenvalue decomposition PDP^T, with P the orthogonal matrix of eigenvectors (uniformly distributed over the unit sphere since the coefficients are gaussian), replace D, the diagonal of eigenvalues, by with D', sampled from another distribution, and recompute $M = PD'P^T$. This allows us to generate symmetric matrices with any eigenvalue distribution, while keeping their eigenvectors uniformly distributed over the unit sphere.

3 MODELS AND EXPERIMENTAL SETTINGS

We use the standard transformer architecture introduced in Vaswani et al. (2017), with an encoder and a decoder connected by a cross-attention mechanism. Most of our models have 512 dimensions, 8 attention heads and up to 6 layers. We experiment with different number of layers, and attention heads in the encoder and decoder. Training is supervised. We minimize the cross-entropy between model prediction and the correct solution, using the Adam optimizer (Kingma & Ba, 2014) with a learning rate of 10^{-4}, an initial warm-up phase of 10000 steps and cosine scheduling (Loshchilov & Hutter, 2016). Training data is generated on the fly, in batchs of 64.

Every $300,000$ examples, $10,000$ random problems are generated and used to evaluate the model. When evaluating, we consider that a predicted sequence seq_P is a correct solution to the problem (N, O) (N and O the input and output matrices) if it can be decoded as a valid matrix P (several matrices for singular and eigen decomposition) that approximates the correct solution to a given tolerance τ ($\tau \in \{5, 2, 1, 0.5\%\}$) . For addition, transposition, multiplication, eigen and singular values we check that P verifies $\|P - O\| < \tau\|O\|$, with the L^1 norm ($\|A\| = \sum_{i,j} |a_{i,j}|$, if $A = (a_{i,j})$). For eigenvalue decomposition, we check that the solution (Q, D) predicted by the model can reconstruct the input matrix, i.e. $\|Q^T DQ - N\| < \tau\|N\|$. For singular value decomposition, we check that $\|USV - M\| < \tau\|M\|$. For matrix inversion, we check that $\|PN - Id\| < \tau$ (since $\|Id\| = 1$). The choice of the L^1 norm is important: norms like L^2 and L^∞ favor models that correctly predict the largest coefficients in the solution. For eigen or singular value problems, this amounts to predicting the largest values, an easier problem than the one we want to solve.

We consider different tolerances for different problems. Since we round numbers to three significant digits, a number x with mantissa 1.00 is subject to a maximal rounding error of 0.5% ($x \in]1.005, 0.995]$), which may accumulate when several (rounded) numbers are summed, and increase again when nonlinear operations are considered. When discussing results, we consider tolerances of 0% for transposition, which involves no arithmetic, 1% for basic matrix operations (addition, multiplication), and 2 or 5% for non linear operations (decomposition, inversion), but we usually provide results for all tolerance levels.

Most of our experiments focus on 5×5 square matrices, or rectangular matrices with as many coefficients ($6 \times 4, 2 \times 13$). This allows for comparison between encodings (for larger dimensions, varying sequence lengths complicate the analysis). We also study scaled-up versions of these problems (from 8×8 to 15×15), and datasets with matrices of variable dimensions (5-10 or 5-15). In this paper, we limit ourselves to problem that can be solved using small models (with up to 6 layers). Scaling to larger problems, and leveraging deeper architectures is left for future research.

4 EXPERIMENTS AND RESULTS

4.1 TRANSPOSITION

Learning to transpose a matrix amounts to learning a permutation of its elements,. For a square matrix, the permutation is composed of cycles of two elements. It becomes more complex for a rectangular matrix. This task involves no arithmetic operations: tokens from the input sequence are merely copied to the output, in different positions. We investigate two formulations of this problems: a fixed-size case, where all matrices in the dataset have the same dimension, and only one permutation is to be learned, and a variable-size case, where the dataset includes matrices of different dimensions, with as many different permutations to learn. We train transformers with one layer, 256 or 512 dimensions and 8 attention heads in the encoder and decoder, over datasets using our four encoding schemes. All models learn to predict the exact solution (with 0% tolerance) in more than 99% of test cases, for fixed-size matrices, square or rectangular, with dimensions up to 30×30. This holds for all encodings, and for input or output sequences up to 2000 tokens long. Similar accuracies are achieved for variable-size datasets: (over 99% for $5-15$ and 96% for $5-20$), with the rectangular cases proving more difficult to train. Table 2 summarizes our findings.

	Fixed dimensions							**Variable dimensions**			
								Square		Rectangular	
	5x5	10x10	20x20	30x30	5x6	7x8	9x11	5-15	5-20	5-15	5-20
P10	100	100	100	-	100	100	100	100	-	97.0	-
P1000	100	100	99.9	-	100	100	100	99.9	-	98.4	-
B1999	100	100	99.9	100	100	100	100	100	96.6	99.6	91.4
FP15	99.8	99.5	99.4	99.8	99.8	99.5	99.3	99.8	99.6	99.4	96.1

Table 2: **Exact prediction of matrix transposition for different matrix dimensions.**

4.2 ADDITION

Learning to add two $m \times n$ matrices amounts to learning the correspondence between the positions of input and output (as in the transposition task) and the algorithm for adding two numbers in floating point representation, which will be performed on mn pairs of elements. We train transformers with 1 or 2 layers, 8 attention heads and 512 dimensions. Sum of fixed-size matrices with dimensions up to 10, both square and rectangular, are predicted with over 99% accuracy within 1% tolerance (and over 98% within 0.5%), with all four encodings. As dimensions increase, models using the P10 and P1000 encodings become more difficult to train as input sequences grow longer: adding two 15×15 matrices involves 450 input coefficients, a sequence of 1352 tokens in P1000 and 2252 in P10. Nevertheless, FP15 models achieve 99.5% accuracy within 0.5% tolerance for 15×15 matrices and B1999 models 89.7% accuracy with 1% tolerance on 20×20 matrices. Variable-size matrices with dimensions up to 10 are predicted by 2-layer transformers using the $B1999$ encoding with over 99.5% accuracy within 1% tolerance. Over matrices with large dimensions (5-15), shallow models with 1 or 2 layers struggle, and their accuracy drops to 48 and 37% in the square and rectangular case. This can be mitigated by using deeper decoders: models with one layer in the encoder and 6 in the decoder achieve 77 and 87% accuracy on the same datasets. Table 3 summarizes our results.

	Fixed dimensions						**Variable dimensions**					
							Square			Rectangular		
Size	5x5	6x4	3x8	10x10	15x15	20x20	5-10	5-15	5-15	5-10	5-15	5-15
Layers	2/2	2/2	2/2	2/2	2/2	1/1	2/2	1/1	1/6	2/2	2/2	1/6
5%	100	99.9	99.9	100	100	98.8	100	63.1	99.3	100	72.4	99.4
2%	100	99.5	99.8	100	100	98.4	99.8	53.3	88.1	99.8	50.8	94.9
1%	100	99.3	99.7	100	99.9	87.9	99.5	47.9	77.2	99.6	36.9	86.8
0.5%	100	98.1	98.9	100	99.5	48.8	98.9	42.6	72.7	99.1	29.7	80.1

Table 3: **Accuracies of matrix sums, for different tolerances.**

4.3 MULTIPLICATION

Multiplication of a matrix M of dimension $m \times n$ by a vector $V \in \mathbb{R}^n$ amounts to computing the m dot products between V and the lines of M. Each calculation features n multiplications and $n-1$ additions, and involve one row in the matrix and all coefficients in the vector. The model must now learn the position of the $2n$ elements in the computation, and two operations (add and multiply). Experimenting with models with 1 or 2 layers, over 5×5 matrices, we observe that only models using the $P10$ and $P1000$ encoding can be trained to high accuracy. The $P1000$ encoding performs best, with little difference between two and one layer models. Accuracies over 99.9%, at 1% tolerance, are achieved by 2-layer transformers using the P1000 encoding for 5×5 and 10×10 square matrices. Comparable accuracies are achieved when multiplying rectangular matrices by vectors with the same overall number of coefficients (30). Experiments with datasets of matrices with variable size (from 5 to 10) achieve non-trivial performance (from 48% with 1% tolerance, to 72% with 5% tolerance, for square matrices). Results are sumarized in table 4.

	P10							Variable 5-10	
	5x5	5x5	10x10	14x2	9x3	4x6	2x10	Square	Rectangular
5%	100	100	100	99.3	99.9	100	100	72.0	41.7
2%	99.9	100	100	99.0	99.7	100	99.8	66.9	35.0
1%	98.5	99.9	99.9	98.7	99.5	99.9	99.2	47.6	20.1
0.5%	81.6	99.5	98.4	98.1	99.0	98.6	94.5	16.2	4.4

Table 4: **Accuracies of matrix-vector products, for different tolerances.** All models are P1000 unless indicated. Fixed-size models have 1 or 2 layers, variable-size have 2 or 4.

Multiplication of matrices M and P is a scaled-up version of the matrix-vector multiplication, which is now performed for every column in matrix P. As previously, only models using the P10 and P1000 encoding can be trained to predict to high accuracy. Over 5×5 matrices and rectangular matrices of similar size, trained model accuracy is the same as vector-multiplication (over 99% at 1% tolerance, see table 5), but deeper decoders (with 4 to 6 layers) are needed.

	Square matrices		Rectangular matrices							
	5x5	5x5	2x13	2x12	3x8	4x6	6x4	8x3	12x2	13x2
	P10 2/2 layers	1/4	4/4	4/4	2/6	1/4	1/6	1/6	1/6	1/4
5%	100	100	100	100	100	100	100	100	100	99.9
2%	100	100	100	100	100	100	100	100	99.7	99.8
1%	99.8	100	99.9	100	100	99.9	100	99.9	99.3	99.8
0.5%	64.5	99.9	97.1	98.5	99.6	99.7	99.5	99.5	99.0	99.8

Table 5: **Accuracy of matrix multiplication, for different tolerances**. Fixed-size matrices with $24-26$ coefficients. All encodings P1000 unless specified.

4.4 EIGENVALUES

We now turn to non-linear computations, usually solved with iterative algorithms. We train models with 4 or 6 layers in the encoder or the decoder. Over samples of 5×5 matrices, we reach 100% accuracy at 5% tolerance, and 98.5% at 1% for all four encodings. For 8×8 matrices, we have accuracies of 100 and 85% at 5 and 1% tolerance. Larger problems, however, prove difficult to learn: on 10×10 matrices, 25% accuracy at 5% tolerance is reached after 360 million examples. As a comparison, 5×5 models train to maximum accuracy in about 40 million examples, 8×8 models in about 60.

This limitation disappears once we train models on variable-size datasets. On samples of matrices with 5-10, 5-15 and 5-20 dimensions, we achieve 100% accuracy at 5% tolerance, and $88, 94$ and 45% at 1%. Using the 5-15 model, the eigenvalues of 10×10 matrices can be predicted with 100% accuracy at 2% tolerance, and 73% at 1%. Table 6 summarizes our results.

	Fixed dimensions							Variable dimensions		
	5x5	5x5	5x5	5x5	8x8	8x8	10x10	5-10	5-15	5-20
Encoding	P10	P1000	B1999	FP15	P1000	FP15	FP15	FP15	FP15	FP15
Layers	6/6	4/1	6/6	6/1	6/1	1/6	1/6	4/4	6/6	4/4
5%	100	100	100	100	100	100	25.3	100	100	100
2%	100	99.9	100	100	99.2	97.7	0.4	99.8	100	75.5
1%	99.8	98.5	98.6	99.7	84.7	77.9	0	87.5	94.3	45.3
0.5%	93.7	88.5	73.0	91.8	31.1	23.9	0	37.2	40.6	22.5

Table 6: **Accuracy of eigenvalues for different tolerances and dimensions.** All models have 8 attention heads, except the 10x10 model, which has 12.

4.5 EIGENVECTORS

This is an expanded version of the previous task: together with the eigenvalues, we predict the orthogonal matrix Q of eigenvectors. Over 5×5 matrices, models using the P10 and P1000 encoding achieve 97.0 and 94.0% accuracy with 5% tolerance. FP15 models fare less well, with an accuracy of 51.6%, but asymmetric models, with 6-layer FP15 encoder and 1-layer P1000 decoder, achieve 93.5% accuracy at 5% and 67.5% at 1% tolerance. The eigenvectors of 6×6 matrices can be predicted by P1000 models with an accuracy of 81.5%. Table 7 summarizes our results.

	5x5				6x6
	P10	P1000	FP15	FP15/P1000	P1000
	4/4 layers	6/6	1/6	6/1	6/1
5%	97.0	94.0	51.6	93.5	81.5
2%	83.4	77.9	12.6	87.4	67.2
1%	31.2	41.5	0.6	67.5	11.0
0.5%	0.6	2.9	0	11.8	0.1

Table 7: **Accuracies of eigenvectors, for different tolerances and dimensions.**

4.6 INVERSION

Inversion of 5×5 matrices proves a difficult task for transformers, with accuracies at 5% tolerance of 73.6% for P10 models, and 80.4 for P100 models (all with 6-layer encoders and 1-layer decoders). Increasing the number of attention heads to 10 and 12 brings no improvement in accuracy, but allows for faster training: 8 head models are trained to 75% accuracy in about 250 million examples, 10 and 12 head models in only 120. The best performances (89.4%) are achieved by asymmetric models: a 6-layer FP15 encoder with 12 attention heads, and a 1-layer P1000 decoder with 4 attention heads.

	P10	P1000			FP15/P1000	
	8/8 heads	8/8 heads	10/8 heads	12/8 heads	10/4 heads	12/4 heads
5%	73.6	80.4	78.8	76.9	87.0	89.4
2%	46.9	61.0	61.7	52.5	74.2	79.9
1%	15.0	30.1	34.2	16.2	49.6	57.8
0.5%	0.2	3.1	5.9	0.1	16.0	24.0

Table 8: **5x5 matrix inversion.** All models have 6/1 layers, except P1000 10 heads, which has 6/6.

4.7 SINGULAR VALUE DECOMPOSITION

Whereas this task related to eigen decomposition (the singular values of a symmetric matrix are the absolute values of its eigenvalues), it proves more difficult to learn: transformers with up to 6 layers, using the P10 or P1000 encoding, can predict the singular decomposition of 4×4, but not 5×5 matrices. Accuracies remain high, 100 and 86.7% for singular values (5 and 1% tolerance), and 98.9 and 73.5% for the full decomposition.

	5%	2%	1%	0.5%
Singular values				
P10 2/2 layers	100	98.5	84.5	41.1
P1000 4/4 layers	100	99.8	86.7	39.8
Singular vectors				
P10 1/6 layers	66.1	11.2	0.2	0
P1000 6/6 layers	98.9	95.6	73.5	5.5

Table 9: **Accuracies of SVD for 4x4 matrices.**

5 OUT-OF-DOMAIN GENERALIZATION AND RETRAINING

In this section, we focus on the prediction of eigenvalues of symmetric matrices. The random n×n matrices used to train our models have independent and identically distributed (iid) coefficients, sampled from a uniform distribution over $[-A, A]$. They belong to a common class of random matrices, known as Wigner matrices. As n becomes large, their eigenvalues are distributed according to a semi-circle law (see 2.2). For small values of n, their eigenvalue distribution is centered and has standard deviation $\sigma = \sqrt{n}s$, where s is the standard deviation of the coefficients ($s = A/\sqrt{3}$ when uniform). Whereas Wigner matrices appear in many problems of science, random matrices with different distributions of eigenvalues (and non iid coefficients) are also found in many practical cases. For instance, statistical covariance matrices have all their eigenvalues positive, and the adjacency matrices of scale-free and other non-Erdos-Renyi graphs have centered but non semi-circle distributions of eigenvalues (Preciado & Rahimian, 2017). It is, therefore, interesting to study how our models, trained on Wigner matrices, perform on matrices with different distributions of eigenvalues.

To this effect, we generate test sets of 10000 matrices with different distributions. We first generate test matrices with uniform iid coefficients (same distribution as the training set), but different standard deviation: $\sigma_{test} \in [0.1\sigma_{train}, 1.5\sigma_{train}]$. Over these test sets, our trained models achieve over 96% accuracy (2% tolerance) for $\sigma_{test} \in [0.6\sigma_{train}, \sigma_{train}]$. However, accuracy drops below $0.6\sigma_{train}$: 54% for $0.4\sigma_{train}$, and 0% for $0.2\sigma_{train}$, and over σ_{train}: 26% for $1.1\sigma_{train}$ and 2% for $1.3\sigma_{\text{train}}$. Out-of-distribution generalization takes place so long the variance of the test set is lower, and not too different from that of the training set.

We then generate test sets of matrices with positive eigenvalues (Wigner matrices with eigenvalues replaced by their absolute values), and with eigenvalues distributed according to the uniform, gaussian and Laplace law (generated as per 2.2), with standard deviation $\sigma_{test} \in [0.6\sigma_{train}, 1.2\sigma_{train}]$. For $\sigma_{test} = \sigma_{train}$, accuracies are 26% for Laplace, 25 for gaussian, 19 for uniform, and 0% for positive. Test results are slightly better for lower standard deviations ($\sigma_{test} = 0.6\sigma_{train}$): $28, 44, 60$, and 0%, but out-of-distribution accuracies remain low, and the eigenvalues of matrices with positive eigenvalues cannot be predicted at all.

To improve out-of-distribution accuracy, we create new datasets with different distributions, train models on them, and evaluate them on our test sets. First, we generate a dataset of matrices with uniform coefficients, but variable standard deviation (by randomly choosing $A \in [1, 100]$ for each matrice). Unsurprisingly, models trained on this dataset achieve high accuracies on samples with high or low variance. Performance on gaussian, uniform and Laplace-distributed eigenvalues is also improved, but matrices with positive eigenvalues still cannot be predicted. Training models over a mixture of matrices with uniform coefficients (semi-circle eigenvalues) and positive eigenvalues results in better prediction of positive eigenvalues, but degrades results on all other tests sets. Mixing matrices with semi-circle and gaussian eigenvalues, or semi-circle and Laplace eigenvalues, on the other hand, results in high accuracies for all test sets, as does training on a dataset of matrices with Laplace eigenvalues only, or a mixture of uniform, gaussian and Laplace eigenvalues. The results of these experiments are presented in table 10.

This is an important result, because it suggests that Wigner matrices, which are often considered the default model for random matrices, might not be the best choice for training transformers. Models trained on matrices with non-iid coefficients (i.e. with eigenvalues not following a semicircle law) generalize to matrices with iid coefficients. As we have seen, the reverse is not true. Overall, this confirms that out-of-distribution generalization is possible if particular attention is paid to how training samples are generated.

	Test set eigenvalue distribution										
	Semicircle			Positive		Uniform		Gaussian		Laplace	
$\sigma_{test}/\sigma_{train}$	0.3	1.0	1.2	0.6	1	0.6	1	0.6	1	0.6	1
iid coeff. A=10 (baseline)	12	100	7	0	0	60	19	44	25	28	26
iid coeff. $A \in [1, 100]$	99	98	97	0	0	68	60	65	59	57	53
Semicircle-Positive	1	99	14	88	99	45	23	31	23	17	20
Semicircle-Gaussian	88	100	100	99	99	96	98	93	97	84	90
Semicircle-Laplace	98	100	100	100	100	100	100	99	100	96	99
Laplace	95	99	99	100	100	98	98	97	98	94	96
Gaussian-Uniform-Laplace	99	100	100	100	100	100	100	99	100	97	99

Table 10: **Out-of-distribution eigenvalue accuracy (tolerance 2%) for different training distributions.**

Models trained on matrices of a given size do not generalize to different dimensions. However, they can be retrained over samples of matrices of different size. This takes comparatively few examples: a 5×5 model, that takes 40 million examples to be trained, can learn to predict with high accuracy eigenvalues of matrices of dimension 6 and 7 with about 25 million additional examples. Table 11 presents those results. The capacity of pre-trained large transformers (such as GPT-3) to perform few-shot learning is well documented, but it is interesting to observe the same phenomenon in smaller models.

Encoding	Retrain dimensions	Accuracy (5%)	Accuracy(2%)	Retrain examples
P10	5-6	100	99.9	10M
P10	5-7	100	93.6	25M
P1000	5-6	100	97.7	25M

Table 11: **Model accuracy after retraining**. Models trained over 5x5 matrices, retrained over 5-6 and 5-7. Overall performance after retraining (tolerance 5 and 2%, and number of examples needed for retraining.

6 Discussion and future directions

We discuss four encoding schemes for matrix coefficients. Our experiments suggest that P10 is generally dominated by P1000, which is also more economical, and that B1999 never really finds its use, as FP15 is more compact and P1000 more efficient. P1000 seems to be a good choice for problems of moderate size, and FP15 when sequence lengths grow. For advanced problems, like eigenvectors and inversion, architectures using a deep encoder with FP15 encoding, and a shallow decoder with P1000, seem to achieve the best performances. Our interpretation is that P1000 in the decoder facilitates training because, by representing output in a meaningful way (sign, mantissa, exponent), it provides the model with better "error signal" during training. On the other hand, a FP15-based deep encoder can implement complex representations of the input matrix, but is easier to train because of the shorter sequences. When using those asymmetric architectures, we note that the deep encoder often benefits from more attention heads, while reducing the number of heads in the shallow decoder improves training stability. We believe that such asymmetric architectures deserve to be better studied and understood.

Most of our experiments focus on matrices with 5 to 10 lines and columns. Experiments on the eigenvalue problem suggest that larger problems can be solved by training over matrices of variable size. In this work, we sample matrices of different dimensions in equal proportion, and present them for training in random order. Better performance might be achieved by leveraging curriculum learning (varying the proportion and scheduling of matrices of different dimensions). Yet, sequence lengths will reach the practical limits of quadratic attention mechanisms as dimension grows. Experimenting with transformers with linear or log-linear attention (Zaheer et al., 2021; Wang et al., 2020a; Vyas et al., 2020; Child et al., 2019) is a natural extension of our work. Throughout this research, we use small models. Our transformers have up to 6 layers and less than 50 million trainable parameters. BERT and GPT-1 (2018) had 12 and 120, GPT-2 (2019) 48 and 1.5 billion, and GPT-3 (2020) 96 and 175 billions. It would certainly be interesting to test large pre-trained models on such computational tasks.

We consider the out-of-distribution experiments as our most striking results. They suggest that models can generalize to a wide range of test distributions. This validates the idea of training models from generated datasets, but indicates that the choice of the training distribution is a decisive factor. In our specific case, we note that the "obvious" random model (Wigner matrices) is not the best for out-of-domain generalization, and that special distributions (matrices with non-iid coefficients with Laplace eigenvalues) can produce better models. This seems to confirm the intuitive idea that we learn more from specific and edge cases, that from averages. A similar situation might apply to other random models, for instance the Erdos-Renyi model for graphs.

7 Related work

Algorithms using neural networks to compute eigenvalues and eigenvectors have been proposed since the early 1990s (Samardzija & Waterland, 1991; Cichocki & Unbehauen, 1992; Yi et al., 2004; Tang & Li, 2010; Oja, 1992), and improvements to the original techniques have been proposed until recently (Finol et al., 2019). Similar approaches have been proposed for other problems in linear algebra (Wang, 1993a;b; Zhang et al., 2008). All these methods leverage the Universal Approximation Theorem (Cybenko, 1989; Hornik, 1991), which states that, under weak conditions on their activation functions, neural networks can approximate any continuous mapping (in our case, the mapping between the coefficients of a matrix and their associated eigenvalues and vectors). These approaches rely on the fact that eigenvalues and vectors appear in the solutions of particular differential equations involving the matrix coefficients (see, for instance, Brockett (1991)). By designing a neural network that represents this differential equation, with the matrix to decompose as the input, and the coefficients in the output layer as the solution, and by defining a loss function measures how well the output layer approximates the correct solution, the network can be trained to predict the solution to the problem. These techniques have two main limitations: they rely on a problem-specific network architecture, that has to be hand-coded, and computation is done at train time, which makes them slow, and implies retraining the network every time a new matrix is to be processed. In comparison, the techniques proposed in this paper are trained to compute at inference, for any matrix input.

Techniques have been proposed to train neural networks to compute basic mathematical operations, and use them as building blocks for larger components. Kaiser & Sutskever (2015) introduced the Neural GPU, that could learn addition and multiplication over binary representations of integers. Trask et al. (2018) proposed Neural Arithmetic Logic Units (NALU), that can learn to perform exact additions, substractions, multiplications and divisions by constraining the weights of a linear network to remain close to 0, 1 or -1. Both Neural GPU and NALU have been shown to be able to extrapolate to numbers far larger than those they were trained on. For matrix multiplication, Blalock & Guttag (2021) use learning techniques to improve on known approximate techniques.

Use of transformers in mathematics has mostly focused on symbolic computations. Lample & Charton (2019) showed that transformers could be trained to integrate functions and solve ordinary differential equations and, in a follow-up work (Charton et al., 2020), predict properties of differential systems. Transformers have also been applied to formal systems, in theorem proving (Polu & Sutskever, 2020) and temporal logic (Hahn et al., 2021). The use of sequence to sequence models for arithmetic and the exact solution of mathematical problem has been studied by Saxton et al. (2019). In a recent paper, Nogueira et al. (2021) point to the limitations of experiments on arithmetic.

8 Conclusion

In this paper, we have shown that transformers can be trained over generated datasets to solve problems of linear algebra with high accuracy, and that careful selection of the generative model for their training data can allow them to generalize out of their training distribution. This demonstrates that applications of transformers to mathematics are not limited to symbolic calculation, and can cover a wider range of scientific problems, featuring numerical computations. Our results on out-of-distribution generalization provide some justification to the idea that models trained over random data can be used to solve "real world" problems. Our results also stress the importance of data generation: by suggesting that natural random models, such as Wigner matrices, might not be the best generative procedure for training models, they open paths for future research.

REFERENCES

Luca Biggio, Tommaso Bendinelli, Alexander Neitz, Aurelien Lucchi, and Giambattista Parascandolo. Neural symbolic regression that scales. *arXiv preprint arXiv:2106.06427*, 2021.

Davis Blalock and John Guttag. Multiplying matrices without multiplying, 2021.

Roger W Brockett. Dynamical systems that sort lists, diagonalize matrices, and solve linear programming problems. *Linear Algebra and its applications*, 146:79–91, 1991.

Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers, 2020.

François Charton, Amaury Hayat, and Guillaume Lample. Learning advanced mathematical computations from examples. *arXiv preprint arXiv:2006.06462*, 2020.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019.

Andrzej Cichocki and Rolf Unbehauen. Neural networks for computing eigenvalues and eigenvectors. *Biological Cybernetics*, 68(2):155–164, 1992.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

Linhao Dong, Shuang Xu, and Bo Xu. Speech-transformer: A no-recurrence sequence-to-sequence model for speech recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5884–5888, 2018. doi: 10.1109/ICASSP.2018.8462506.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

David Finol, Yan Lu, Vijay Mahadevan, and Ankit Srivastava. Deep convolutional neural networks for eigenvalue problems in mechanics. *International Journal for Numerical Methods in Engineering*, 118(5):258–275, 2019.

Christopher Hahn, Frederik Schmitt, Jens U. Kreber, Markus N. Rabe, and Bernd Finkbeiner. Teaching temporal logics to neural networks, 2021.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4 (2):251–257, 1991.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Guillaume Lample and François Charton. Deep learning for symbolic mathematics. *arXiv preprint arXiv:1912.01412*, 2019.

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Madan Lal Mehta. *Random Matrices*. 3rd edition, 2004.

Rodrigo Nogueira, Zhiying Jiang, and Jimmy Lin. Investigating the limitations of transformers with simple arithmetic tasks, 2021.

Erkki Oja. Principal components, minor components, and linear neural networks. *Neural networks*, 5(6):927–935, 1992.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020.

Victor M. Preciado and M. Amin Rahimian. Moment-based spectral analysis of random graphs with given expected degrees, 2017.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Nikola Samardzija and RL Waterland. A neural network for computing eigenvectors and eigenvalues. *Biological Cybernetics*, 65(4):211–214, 1991.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models, 2019.

Feng Shi, Chonghan Lee, Mohammad Khairul Bashar, Nikhil Shukla, Song-Chun Zhu, and Vijaykrishnan Narayanan. Transformer-based machine learning for fast sat solvers and logic synthesis, 2021.

Ying Tang and Jianping Li. Another neural network based approach for computing eigenvalues and eigenvectors of real skew-symmetric matrices. *Computers & Mathematics with Applications*, 60 (5):1385–1392, 2010.

Andrew Trask, Felix Hill, Scott Reed, Jack Rae, Chris Dyer, and Phil Blunsom. Neural arithmetic logic units. *arXiv preprint arXiv:1808.00508*, 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pp. 6000–6010, 2017.

Apoorv Vyas, Angelos Katharopoulos, and François Fleuret. Fast transformers with clustered attention, 2020.

Jun Wang. A recurrent neural network for real-time matrix inversion. *Applied Mathematics and Computation*, 55(1):89–100, 1993a.

Jun Wang. Recurrent neural networks for solving linear matrix equations. *Computers & Mathematics with Applications*, 26(9):23–34, 1993b.

Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity, 2020a.

Yongqiang Wang, Abdelrahman Mohamed, Due Le, Chunxi Liu, Alex Xiao, Jay Mahadeokar, Hongzhao Huang, Andros Tjandra, Xiaohui Zhang, Frank Zhang, and et al. Transformer-based acoustic modeling for hybrid speech recognition. *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2020b. doi: 10.1109/icassp40776.2020.9054345. URL http://dx.doi.org/10.1109/ICASSP40776.2020.9054345.

Zhang Yi, Yan Fu, and Hua Jin Tang. Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix. *Computers & Mathematics with Applications*, 47(8-9): 1155–1164, 2004.

Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences, 2021.

Yunong Zhang, Weimu Ma, and Binghuang Cai. From zhang neural network to newton iteration for matrix inversion. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 56(7):1405–1415, 2008.