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, which serve as fundamental building blocks in many scientific problems: basic operations on matrices, matrix inversion, eigenvalue and singular value decompositions. 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 (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 $m \times n$ 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 $M = Q^T D Q$, set as a $(n+1) \times n$ matrix, with (sorted) eigenvalues in 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), and complex nonlinear transformations involving the whole matrix, with cubic complexity (decompositions and inversion). For each problem, we generate datasets of pairs of matrices (I, O), by sampling random input matrices I (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 I). All coefficients in I 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. Table 1 provides examples of these encodings. Additional details and examples can be found in Appendix A.

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.**

Selecting an encoding is a trade-off. Long encodings (P10, P1000) embed knowledge about numbers that the model can leverage (e.g. numbers can be compared using their signs and exponents, addition and multiplication can be learned by memorizing small tables). Compact encodings use a larger vocabulary (harder to learn), but generate shorter sequences that facilitate training with transformers. In P10, a 20×20 matrix is a sequence of 2002 tokens, close to the practical limit of transformer implementations that use a quadratic attention mechanism. In FP15, it is only 402 tokens long.

2.2 RANDOM MATRIX GENERATION

In most of our experiments, we train models over datasets of random matrices with uniformly distributed coefficients in $[-A, A]$ (with $A = 10$). 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 (Mehta, 2004). As n increases, this 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 M, with gaussian coefficients, and compute their eigenvalue decomposition $M = PDP^T$, with P the orthogonal matrix of eigenvectors (uniformly distributed over the unit sphere since the coefficients are gaussian). We then replace D, the diagonal matrix of eigenvalues of M, with a diagonal D' sampled from another distribution. Finally, we recompute $M' = PD'P^T$, a symmetric matrix (because P is orthogonal) with eigenvalues distributed as we choose, and 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. All training is supervised, and minimizes the cross-entropy between model prediction and the correct solution. We use the Adam optimizer (Kingma & Ba, 2014) with a learning rate of 10^{-4}, an initial warm-up phase of 10,000 steps and cosine scheduling (Loshchilov & Hutter, 2016). All training data is generated on the fly, in batches of 64 examples.

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 (I, O) (I 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 $\|A\| = \sum_{i,j} |a_{i,j}|$, for $A = (a_{i,j})$, i.e. L^1 norm). For eigenvalue decomposition, we check that the solution (Q, D) predicted by the model can reconstruct the input matrix, i.e. $\|Q^T DQ - I\| < \tau\|I\|$. For singular value decomposition, we check that $\|USV - I\| < \tau\|I\|$. For matrix inversion, we check that $\|PI - Id\| < \tau\|Id\| = \tau$. The choice of the L^1 norm is important: norms like L^2 and L^∞ will favor models that correctly predict the largest coefficients in the solution. For eigen and 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, 0.5% is the best we can hope. In fact, a number x with mantissa 1.00 is subjected 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 (e.g. $6 \times 4, 2 \times 13$). This helps when comparing encodings: for larger dimensions, varying sequence lengths make comparisons difficult. We also study scaled-up versions of the 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. Permutations for rectangular matrices involve longer cycles. 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 problem: 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 permutations to learn. We train transformers with one layer, 256 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 slightly more difficult to train. Table 2 summarizes our results.

| | 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.** Transformers with 1 layer, 256 dimensions and 8 attention heads.

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 larger 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.** B1999 encoding, 512 dimension and 8 attention heads.

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.

Tolerance	P10 5x5	P1000 5x5	10x10	14x2	P1000 9x3	4x6	2x10	Variable 5-10 (P1000) Square	Rectangular
5%	100	100	100	99.3	99.9	100	100	72.4	41.7
2%	99.9	100	100	99.0	99.7	100	99.8	68.4	35.0
1%	98.5	99.9	99.9	98.7	99.5	99.9	99.2	60.1	20.1
0.5%	81.6	99.5	98.4	98.1	99.0	98.6	94.5	30.8	4.4

Table 4: **Accuracies of matrix-vector products, for different tolerances.** Fixed-size models have 1 or 2 layers, variable-size have 2 or 4. All model have 512 dimensions and 8 attention heads.

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.

Tolerance	Square matrices 5x5	5x5	2x13	2x12	Rectangular matrices 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 are P1000 unless specified. Models have 512 dimensions and 8 attention heads

4.4 EIGENVALUES

We now turn to non-linear problems that are usually solved by iterative algorithms. We train models with 4 or 6 layers in the encoder or the decoder to predict the eigenvalues of symmetric matrices. Over samples of 5×5 random matrices, we reach 100% accuracy at 5% tolerance, and 98.5% at 1% for all four encodings. For 8×8 matrices, we achieve 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, and 8×8 models in about 60 million.

We can overcome this limitation by training 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 512 dimensions and 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 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
Tolerance	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 depths.** All models have 512 dimensions and 8 attention heads

4.6 INVERSION

Inversion of 5×5 matrices proves more difficult than previous tasks, with accuracies of 73.6% for P10 models, and 80.4 for P1000 models (5% tolerance, 6-layer encoders and 1-layer decoders). Increasing the number of attention heads to 10 and 12 brings little 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 highest accuracies (90.0%) are achieved by asymmetric models: a 6-layer FP15 encoder with 12 attention heads, and a 1-layer P1000 decoder with 8 heads.

	P10	P1000			FP15/P1000	
Tolerance	8/8 heads	8/8 heads	10/8 heads	12/8 heads	10/4 heads	12/8 heads
5%	73.6	80.4	78.8	76.9	88.5	90.0
2%	46.9	61.0	61.7	52.5	78.4	81.8
1%	15.0	30.1	34.2	16.2	55.5	60.0
0.5%	0.2	3.1	5.9	0.1	20.9	24.7

Table 8: **5x5 matrix inversion.** All models have 6/1 layers, except P1000 10 heads, which has 6/6 (and 512 dimensions).

4.7 SINGULAR VALUE DECOMPOSITION

Whereas this task is 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 75.3% for the full decomposition.

	Singular values		Singular vectors	
	P10 2/2 layers	P1000 4/4 layers	P10 1/6 layers	P1000 6/6 layers
5%	100	100	71.5	98.9
2%	98.5	99.8	15.6	95.7
1%	84.5	86.7	0.4	75.3
0.5%	41.1	39.8	0	6.3

Table 9: **Accuracies of SVD for 4x4 matrices.** All models have 512 dimensions and 8 attention heads

5 OUT-OF-DOMAIN GENERALIZATION AND RETRAINING

In this section, we focus on the prediction of eigenvalues of symmetric matrices. To train our models, we generate random n×n matrices with 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. Their eigenvalues have a centered distribution with standard deviation $\sigma = \sqrt{n}s$, where s is the standard deviation of the coefficients ($s = A/\sqrt{3}$ when uniform). As n increases, this distribution converges to the semi-circle law (Mehta (2004)). Whereas Wigner matrices are very common in science, random matrices with different eigenvalue distributions (and non iid coefficients) appear 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, important to understand how models trained on Wigner matrices perform on matrices with different distributions of eigenvalues.

To this effect, we create test sets of 10,000 matrices with different distributions than the training set. First, we generate matrices with uniform iid coefficients (as in 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 (with 2% tolerance) for $\sigma_{test} \in [0.6\sigma_{train}, \sigma_{train}]$. However, model accuracy drops when σ_{test} is out of this range: 54% for $0.4\sigma_{train}$, 0% for $0.2\sigma_{train}$, 26% for $1.1\sigma_{train}$ and 2% for $1.3\sigma_{train}$. Out-of-distribution generalization only takes place when test set variance is lower, and not too far from (over 25% of) training set variance.

Then, we generate test sets of matrices with different eigenvalue distributions: positive eigenvalues (Wigner matrices with eigenvalues replaced by their absolute values), and eigenvalues distributions according to the uniform, gaussian or Laplace law (see section 2.2), with standard deviation σ_{train} and $0.6\sigma_{train}$. Over test sets with $\sigma_{test} = \sigma_{train}$, accuracies are 26% for Laplace, 25 for gaussian, 19 for uniform, and 0% for positive. Results are slightly better for test sets with lower standard deviation ($0.6\sigma_{train}$): 28, 44, 60 and 0% for Laplace, gaussian, uniform and positive, but out-of-distribution accuracies are low, and matrices with positive eigenvalues cannot be predicted at all.

To improve out-of-distribution accuracy, we train new models on datasets with different distributions of eigenvalues, and evaluate them on the test sets previously created. First, we generate matrices with uniform coefficients but variable standard deviation (by randomly selecting $A \in [1, 100]$ for each matrix). Unsurprisingly, models trained on this dataset achieve high accuracies on test sets of Wigner matrices with high or low variance. Performances also increase over the gaussian, uniform and Laplace-distributed test sets (from $25 - 60\%$ to $53 - 68\%$). Yet, matrices with positive eigenvalues cannot be predicted. Training models over a mixture of (Wigner) matrices with uniform iid coefficients and matrices with positive eigenvalues results in better prediction of positive eigenvalues, but degrades performances over all other tests sets.

However, models trained on a mixture of matrices with uniform coefficients and matrices with gaussian eigenvalues, or uniform iid and Laplace eigenvalues, achieve high accuracies over all test sets, as do models trained on matrices with Laplace eigenvalues only, or a mixture of uniform, gaussian and Laplace eigenvalues (all non-Wigner matrices). These experiments are presented in table 10.

This is an important result: it suggests that Wigner matrices, often considered as the default model for random matrices, might not be the best choice for training transformers. Models trained on

non-Wigner matrices (non-iid coefficients, limit distribution of eigenvalues not a semi-circle) generalize to matrices with iid coefficients, whereas the reverse is not true. This confirms that out-of-distribution generalization requires that particular attention is paid to training data generation.

| | 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**. All models have 512 dimensions and 8 attention heads, and use the P1000 encoding.

Models trained on matrices of a given size do not generalize to different dimensions, but 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. All models have 512 dimensions and 8 attention heads

6 DISCUSSION AND FUTURE DIRECTIONS

Our experiments demonstrate that transformers can be trained to solve problems of linear algebra, using randomly generated datasets. However, their accuracy depends on the encodings used to represent matrix coefficients. We introduce four encoding schemes, and 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 for longer sequences. For advanced problems like eigenvectors and inversion, asymmetric architectures, with a deep FP15 encoder and a shallow P1000 decoder, achieve the best performances. Our interpretation is that P1000 in the decoder facilitates training because the meaningful representation of output as (sign, mantissa, exponent) triplets allows for better error feedback during training. On the other hand, a FP15 deep encoder can provide more complex representations of the input matrix, while being easier to train thanks to the shorter sequences. Such asymmetric architectures also benefit from more attention heads (10 to 12) in the encoder, while less heads (4) in the decoder improve training stability at no cost in accuracy. Those asymmetric architectures deserve further study.

Most of our experiments focus on matrices with 5 to 10 lines and columns. Our results on the eigenvalue problem suggest that larger problems can be solved by training over matrices of variable size, or retraining over larger matrices. In this work, matrices of different dimensions are sampled in equal proportion and presented for training in random order. Varying their proportions and scheduling (i.e. curriculum learning) should result in better performance. Yet, as dimension increases, sequence lengths will reach the practical limits of quadratic attention mechanisms. 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. In terms of asymptotic complexity, matrix inversion (and the other non linear tasks) is usually handled by $O(n^3)$ algorithms (although $O(n^{2.37})$ methods are known). Since our sequence length is $O(n^2)$, transformers with quadratic attention mechanisms are $O(n^4)$. Linear attention would reduce this to $O(n^2)$.

The out-of-distribution experiments are our most significant results. They prove that models trained on random data can generalize to a wide range of test distributions. They also confirm the importance of wisely selecting training data distributions, a process that can be counter-intuitive. In our specific case, the "obvious" random model (Wigner matrices) is not the best for out-of-domain generalization. In fact, we show that sets of "special" matrices (non-iid coefficients with Laplace eigenvalues) can produce models with better capability for generalization, notably on Wigner matrices. This matches the intuitive idea that we learn more from edge cases than averages.

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 suggested 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 (e.g. 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 that measures how well the output layer approximates the correct solution, the network can be trained to find better and better approximations to the solution. 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 once, and can 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

We have shown that transformers can be trained over generated data 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 wide range of scientific problems, featuring numerical computations. We believe our results pave the way for wider applicability of transformers in science.

Reproducibility statement

The transformer implementation and the framework for running the experiments were used in several prior works, and rely on standard libraries (Pytorch for the models, Numpy for mathematical calculations). The model source code, data generation code, and parameters for experiments will be open-sourced and made publicly available. All experiments were run several times (on average 10 times), with multiple random seeds and light modifications of the hyperparameters (e.g. small changes in model size, weight initialization, activation functions), to guarantee their robustness.

Ethics statement

Given the subject of the paper, and the fact that all data used are randomly generated, we believe that no potential ethical concerns are raised by this research.

REFERENCES

Anonymous. Adaptive control flow in transformers improves systematic generalization. In *Submitted to The Tenth International Conference on Learning Representations*, 2022. URL `https://openreview.net/forum?id=KBQP4A_J1K`. under review.

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. *arXiv preprint arXiv:2106.10860*, 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. *arXiv preprint arXiv:2005.12872*, 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. *arXiv preprint arXiv:1904.10509*, 2019.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

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.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

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.

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. *arXiv preprint arXiv:2010.11929*, 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.

Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Christopher Hahn, Frederik Schmitt, Jens U. Kreber, Markus N. Rabe, and Bernd Finkbeiner. Teaching temporal logics to neural networks. *arXiv preprint arXiv:2003.04218*, 2021.

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

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.

Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, third edition, 1997.

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*. Academic Press, 3rd edition, 2004.

Rodrigo Nogueira, Zhiying Jiang, and Jimmy Lin. Investigating the limitations of transformers with simple arithmetic tasks. *arXiv preprint arXiv:2102.13019*, 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. *arXiv preprint arXiv:2009.03393*, 2020.

Victor M. Preciado and M. Amin Rahimian. Moment-based spectral analysis of random graphs with given expected degrees. *arXiv preprint arXiv:1512.03489*, 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. *arXiv preprint arXiv:1904.01557*, 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. *arXiv preprint arXiv:2107.07116*, 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. *arXiv preprint arXiv:2007.04825*, 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. *arXiv preprint arXiv:2006.04768*, 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. *2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2020b.

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. *arXiv preprint arXiv:2007.14062*, 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.

A   NUMBER ENCODINGS

Let x be a non-zero real number, it can be represented uniquely as $x = s.m.10^e$, with $s \in \{-1, 1\}, m \in [100, 1000[, e \in \mathbb{Z}$. Rounding m to the nearest integer n, we get the base ten, floating-point representation of x, with three significant digits:

$$x \approx s.n.10^e, (s, n, e) \in \mathbb{Z}^3$$

By convention 0 is encoded as $+0.10^0$. All our encodings are possible representations of the triplets (s, n, e). In this paper, we limit e to the range $[-100, 100]$, and n to the range $[100, 999]$.

In base N positional encoding, we encode s (the sign) and e (the exponent) as unique tokens: $+$ or $-$ for s, and a token from E-100 to E100 for e. The mantissa, n, is encoded as the representation of n in base N (e.g. binary representation if $N = 2$, decimal representation if $N = 10$), a sequence of $\lceil log_N(1000) \rceil$ tokens from 0 to N-1. Overall, a number will be encoded as a sequence of $\lceil log_N(1000) \rceil + 2$ tokens, from a vocabulary of $202 + N$ tokens.

For instance, $x = e^\pi \approx 23.14069$, will be represented by $+231.10^{-1}$, and encoded in P10 (base 10 positional) as the sequence [+,2,3,1,E-1], and in P1000 (base 1000 positional) as [+,231,E-1]. $x = -0.5$ will be represented as -500.10^{-3}, and encoded in P10 as [-,5,0,0,E-3], and in P1000 as [-,500,E-3]. Other bases N could be considered, as well as different bases for the exponent, and different sizes of the mantissa. In this paper, we use P10 to encode numbers with absolute value in $[10^{-100}, 10^{101}]$ as sequences of 5 tokens, using a vocabulary of 213 tokens (10 digits, 2 signs, and 201 values of the exponent), and P1000 as sequences of 3 tokens, with a vocabulary of 1104.

Balanced base $2a + 1$ uses digits between $-a$ and a (Knuth, 1997). For instance, in balanced base 11, digits range from -5 to 5 (an every day example of a balanced base can be found in the way we state the hour as "twenty to two", or "twenty past two"). Setting a to 999, we define B1999, and encode the sign an mantissa as a single token between -999 and 999. Numbers are then encoded on two tokens with a vocabulary of 2004.

Finally, we encode floating point numbers as unique tokens by rewriting any number $x = m10^b$, with $m \in [-999, 999], b \in [-(p+2)/2, (p+2)/2], p + 2 = 0, [2]$, and encoding it as the unique token FPm, b. This allows to represent numbers with 3 significant digits and a dynamic range of 10^{p+2}, using a vocabulary of $(18p + 2)10^3$ tokens. In this paper, we use $p = 16$: encoding numbers as unique tokens, with a vocabulary of $30,000$ (FP15).

B L^1, L^2 AND L^∞ NORMS FOR EVALUATION

We evaluate the accuracy of our trained models by decoding model predictions and verifying that they approximate the correct solution up to a fixed tolerance τ. In the general case, if the model predict a sequence seq_P, and the solution of the problem is O, we consider that the prediction is correct if seq_P can be decoded into a matrix P and

$$\|P - O\| < \tau\|O\| \tag{1}$$

For eigenvalue decomposition, we check that the solution (Q, D) predicted by the model can reconstruct the input matrix, i.e. $\|Q^T DQ - I\| < \tau\|I\|$. For singular value decomposition, we check that $\|USV - I\| < \tau\|I\|$. For matrix inversion, we check that $\|PI - Id\| < \tau\|Id\| = \tau$.

All our published results use the norm L^1: $\|A\| = \sum_{i,j} |a_{i,j}|$, for $A = (a_{i,j})$. In this section, we discuss the impact of using different norms, namely L^2 ($\|A\| = \sum_{i,j} a_{i,j}^2$), or L^∞ ($\|A\| = \max_{i,j} |a_{i,j}|$).

Using L^1 norm in equation 1 amounts to comparing the average absolute errors on the predicted coefficients ($P - O$) to the average absolute value of coefficients of O. Using L^2 compares the squared errors, and will bias the estimation towards large absolute errors, and coefficients of O with large absolute values. L^∞ will compare the largest absolute error to the largest coefficient in $|O|$. The choice of the norm has different impact for different problems. Figure 1 presents learning curves using the three norms for our best models on different problems.

For basic arithmetic operations (transposition, addition, multiplication), there is little difference between L^1 and L^2 accuracies, and therefore no reason to prefer one over the other for model
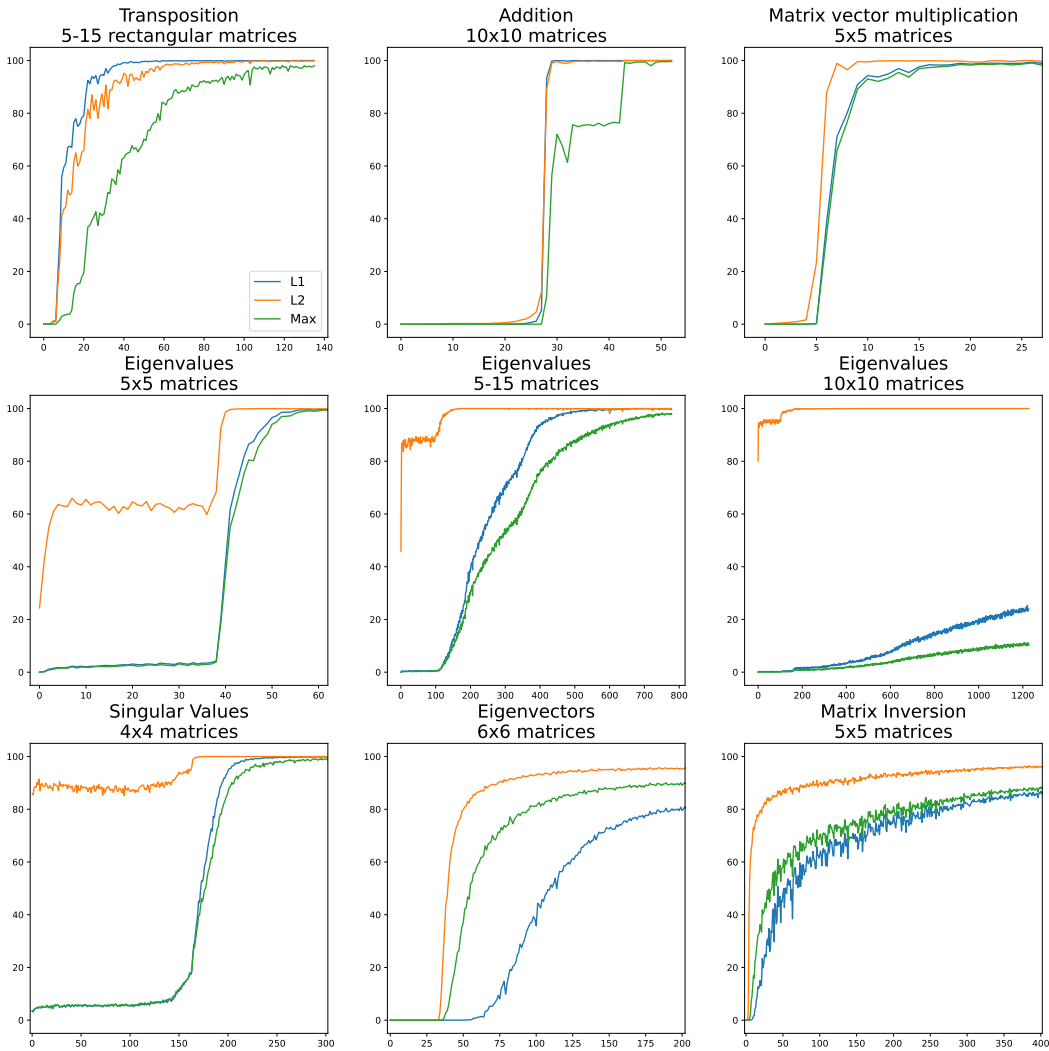
Figure 1: Learning accuracies for different problems measured with norms L^1, L^2 and L^∞.

evaluation. For eigen and singular value problems, L^2 accuracies reach a high value early during training, long before the model begins to learn according to the other norms. This is due to the fact that the eigenvalues of Wigner matrices tend to be regularly spaced over the interval $[-2\sigma, 2\sigma]$ ($\sigma = \sqrt{(n)}s$ with s the standard deviation of coefficients and n the dimension of the matrix). This means that, in many cases, the model can predict the largest absolute eigenvalues from the bounds of the interval (which can be estimated from the dataset). For this reason, L^2 accuracy is not a good evaluation metric for the eigenvalue or singular value problem. This is particularly clear in the 10×10 case: transformers struggle with such matrices, and L^1 and L^∞ accuracies remain very low even after a thousand epochs (300 million examples), but L^2 accuracy is close to 100% since the beginning of training. A similar phenomenon takes place for eigenvector calculations: L^2 and L^∞ accuracy rise steeply, long before the model begins to learn (according to the L^1 norm). This justifies the choice of L^1 as our evaluation norm.
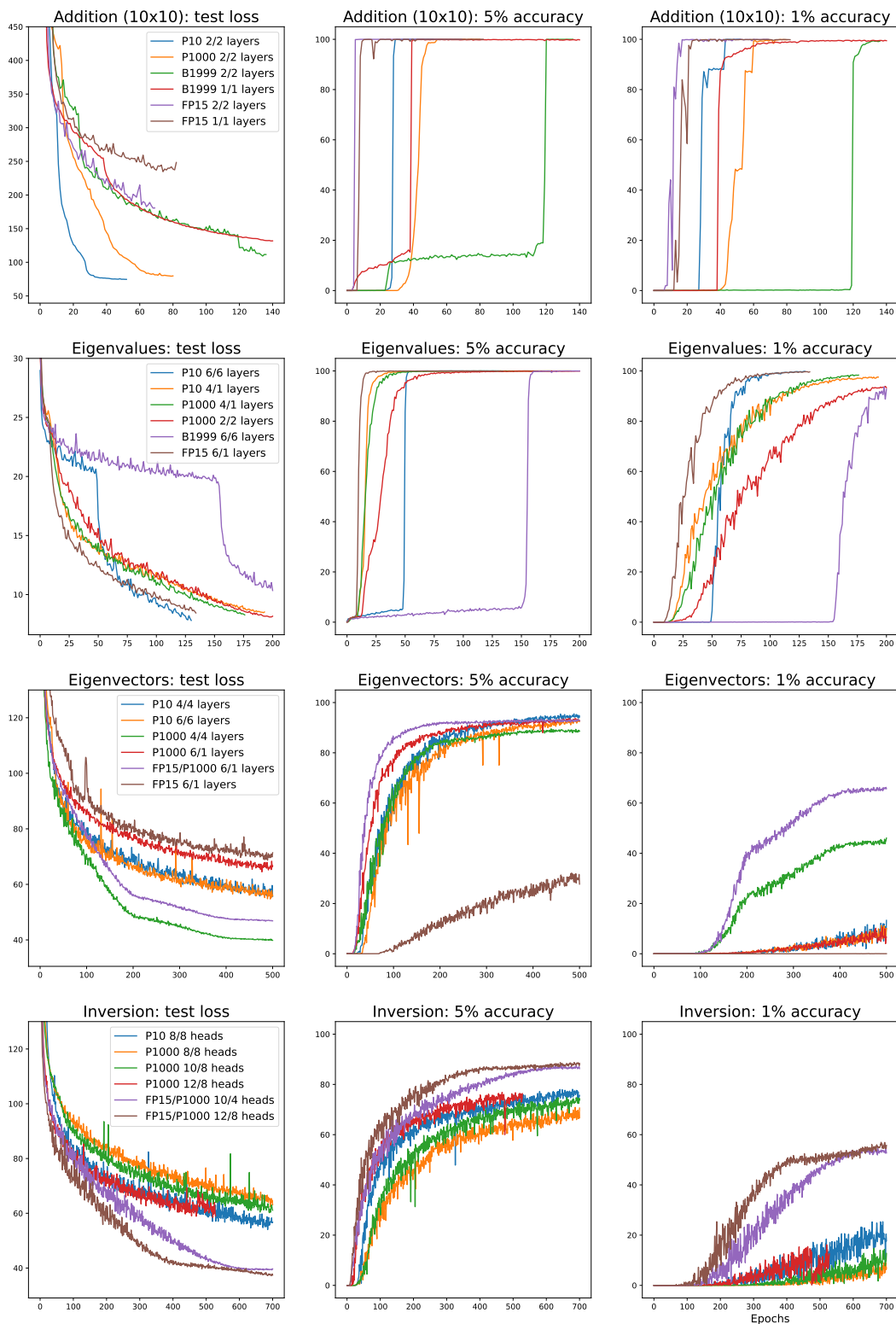
Figure 2: Learning curves for different problems. All problems except addition use 5×5 matrices. All models have 512 dimensions and $8/8$ heads (except when mentioned in the legend). Inversion models have $6/1$ layers. Epochs correspond to $300{,}000$ training examples. Test loss is cross-entropy.

C ADDITIONAL EXPERIMENTAL RESULTS

C.1 LEARNING CURVES FOR DIFFERENT ENCODINGS AND ARCHITECTURES

Figure 2 presents learning curves for loss and accuracy (with 5 and 1% tolerance) on different models, for four problems. These curves indicate the number of training examples needed for each problem. On average, our best models learn basic operations on matrices in less than 50 epochs (15 million examples). Training size requirement increases with operation complexity : from 30 million for eigenvalues, to 120 million for eigenvectors, and over 150 million for matrix inversion.

On the inversion problem, we experiment with the number of attention heads in the encoder. Increasing the number of head from 8 to 10 and 12 improves learning speed and accuracy. Over 12 heads, this benefit disappears: with 16 heads, our models need 800 epochs to train to 55% accuracy (with 5% tolerance). We believe that this reflects the trade-off being the number of heads (more heads catch more dependencies between elements in the input sequeunce) and the downsampling of attention patterns (when internal model dimension remains fixed).

Finally, we notice that the learning curves for the harder problems (eigenvalues, vectors and inversion) are noisy. This is caused by the learning rates: our models usually need small learning rates (510^{-4} before scheduling is typical) and there is a trade-off between low rates that will stabilize the learning curve, and larger rates that accelerate training.

C.2 MODEL SIZE

The two main factors influencing model size are depth and the number of dimensions (see Appendix F). In this section we discuss how model size impacts accuracy for addition of 10×10 matrices, multiplication of a 5×5 matrix by a vector, and computation of the eigenvalues of a 5×5 matrix. All models in this section are symmetric (same dimension and number of layers in the encoder and decoder) and have 8 attention heads.

For the addition task, tables 12 and 13 present the accuracy reached after 60 epochs (18 million examples) and the number of epochs (of 300,000 examples) needed to reach 95% accuracy, for models using the P1000 and B1999 encoding. Both encodings allow shallow architectures (1/1 and 2/2 layers) to learn addition with high accuracy, but the more compact B1999 support smaller models (256 dimensions). In terms of speed, with B1999, shallow models are learned very fast, but it takes a lot of examples to train deeper models. The opposite is true for P1000 models.

dimension	B1999				P1000			
	64	128	256	512	64	128	256	512
1/1 layers	31	7	82	100	0	0	1	40
2/2 layers	0	0	100	100	0	0	0	99
4/4 layers	0	0	0	14	0	0	0	98
6/6 layers	0	0	0	0	0	0	0	99

Table 12: **Accuracy of matrix addition for different model sizes.** 10×10 matrices, 60 epochs (18 millions examples). 5% tolerance

dimension	B1999				P1000			
	64	128	256	512	64	128	256	512
1/1 layers	-	-	76	15	-	-	-	96
2/2 layers	-	-	26	6	-	-	-	37
4/4 layers	-	-	70	63	-	-	-	53
6/6 layers	-	-	-	-	-	-	-	23

Table 13: **Learning speed of matrix addition for different model sizes.** Number of epochs needed to reach 95% accuracy (with 5% tolerance). 1 epoch = 300,000 examples.

Table 14 presents the learning speed of models of different sizes for the matrix/vector product and eigenvalue computation tasks (5×5 matrices, and P1000 encoding). For each problem, there exist a

minimal dimension and depth under which models struggle to learn: one layer and 128 dimensions for products, one layer or 128 dimensions for eigenvalues. Over that limit, increasing the dimension accelerates learning. Increasing the depth, on the other hand, bring no clear improvement in speed or accuracy.

	Matrix product			Eigenvalues			
	128	256	512	128	256	512	1024
1/1 layers	-	29	18	-	-	-	-
2/2 layers	24	12	7	-	102	36	23
4/4 layers	28	11	5	244	90	24	13
6/6 layers	24	10	6	-	-	129	16
8/8 layers	18	12	6	-	-	34	24

Table 14: **Learning speed of matrix and vector products and eigenvalue calculations for different model sizes.** Number of epochs needed to reach 95% accuracy (with 5% tolerance). 1 epoch = 300,000 examples. 5×5 matrices, P1000 encoding.

Finally, we experiment with larger models on larger problems. We trained models with 8 to 12 layers and 512 to 2048 dimensions on sets of 10×10 matrices, without success. As discussed in section 4.4, those problems are out of reach of the models we use in this paper (unless we use curriculum learning and train on mixed-size datasets). Increasing model size does not seem to help scaling to larger matrices.

C.3 MODEL PERFORMANCE ON DIFFERENT DATASETS

	FP15		P1000	
	4/1 layers	6/1 layers	4/1 layers	6/1 layers
Wigner matrices (iid coefficients)				
Uniform iid A=10	99.6	100	99.8	100
Gaussian iid A=10	99.8	100	99.8	100
Uniform iid A=1,100	99.0	99.2	99.8	100
Uniform iid A=1,1000	99.2	99.5	99.7	99.8
Non Wigner				
Positive A=10	12.7	100	100	100
Uniform A=10	8.2	10.8	99.9	100
Gaussian A=10	99.6	100	100	100
Laplace A=10	99.4	99.9	99.9	99.9
Gaussian+uniform+Laplace A=10	3.8	99.8	99.6	99.9
Wigner and non Wigner mixtures				
iid+gaussian A=10	99.5	99.9	98.0	99.7
iid+positive A=10	99.8	99.9	99.8	99.8
iid+Laplace A=10	99.6	99.8	99.6	99.5
iid+positive+gaussian A=10	99.8	99.9	99.7	99.9
iid+positive+Laplace A=10	99.0	99.8	99.6	99.8

Table 15: **In-distribution eigenvalue accuracy (tolerance 2%) for different training distributions.** All models have 512 dimensions, and 8 attention heads, and are trained on 5x5 matrices.

Table 15 sumarizes in-domain performance (i.e. accuracy when the test set is generated with the same procedure as the training set) for different datasets. On Wigner matrices (i.e. matrices with independant and identically distributed, iid, coefficients) uniformly or normally distributed, with fixed-range coefficients (i.e. all matrices in the dataset have coefficients uniformly sampled from the same interval), or variable-range coefficients (i.e. coefficient range vary from one matrix to another), all models achieve very high $(99 + \%)$ accuracy. The eigenvalues of Non-Wigner matrices with Gaussian or Laplace distributed eigenvalues, are also predicted to high accuracy by all models. Over matrices with positive or uniformly distributed eigenvalues, smaller models using the FP15 encoding prove difficult to train. Finally, on mixtures of Wigner and non Wigner matrices, all models predict to high accuracy.

D ALTERNATIVE ARCHITECTURES

D.1 OTHER SEQUENCE TO SEQUENCE MODELS : LSTM AND GRU

We experimented with two popular recurrent architectures: long short-term memories (Hochreiter & Schmidhuber, 1997), and gated recurrent units (Cho et al., 2014), on three tasks : addition of 5×5 and 10×10 matrices, eigenvalues and matrix inversion of 5×5 matrices. We experiment with sequence to sequence models, featuring an encoder and a decoder (LSTM or GRU), with 2 to 8 layers, and 1024 or 2048 hidden dimensions. The input and output sequences, encoded as in the rest of the paper, are pre-processed (and decodedà) via an embedding layer with 256 or 512 dimensions.

Addition, a very easy task for transformers (see section 4.2) proves difficult for LSTM and GRU. None of our models can learn addition of 10×10 matrices. Some models can learn addition of 5×5 matrices, but whereas transformers achieve 100% accuracy for all tolerances, our best LSTM and GRU only exceed 90% at 1% tolerance. GRU seem to perform better than LSTM on this task, and 2-layer models perform better than 4-layer models, but transformers have a distinct advantage over LSTM and GRU on addition.

	2 layers				4 layers			
Hidden dimension	1024		2048		1024		2048	
Embedding dimension	256	512	256	512	256	512	256	512
Long short-term memory								
5% tolerance	100	0	0	100	0	0	0	0
2% tolerance	98	0	0	100	0	0	0	0
1% tolerance	95	0	0	86	0	0	0	0
0.5% tolerance	34	0	0	1	0	0	0	0
Gated recurrent Units								
5% tolerance	100	100	0	100	0	100	0	0
2% tolerance	100	28	0	100	0	99	0	0
1% tolerance	44	0	0	91	0	74	0	0
0.5% tolerance	0	0	0	9	0	4	0	0

Table 16: **Matrix addition with LSTM and GRU.** 5×5 matrices.

Both LSTM and GRU can be trained to predict eigenvalues of 5×5 matrices with the same accuracy as transformers, for the P1000 and FP15 encoding (table 17). Matrix inversion, on the other hand, cannot be learned. Overall, these experiments show that other sequence to sequence architectures, LSTM and GRU, can learn tasks like eigenvalues and addition of small matrices. However, they are less efficient on addition (in terms of precision and scaling to larger matrices) and fail on more complex tasks, like matrix inversion.

	FP15						P1000					
Hidden dimension	1024			2048			1024			2048		
Layers	4	6	8	4	6	8	4	6	8	4	6	8
LSTM												
5% tolerance	100	100	100	100	100	6	100	100	5	100	100	100
2% tolerance	95	100	100	99	100	1	100	100	1	100	99	100
1% tolerance	78	98	99	91	98	0	97	98	0	100	92	99
0.5% tolerance	46	81	83	62	68	0	78	88	0	89	57	76
Gated recurrent Units												
5% tolerance	100	100	100	100	100	100	100	100	100	100	5	100
2% tolerance	98	99	100	100	100	100	99	100	100	100	1	100
1% tolerance	86	93	96	98	99	97	94	98	95	97	0	98
0.5% tolerance	53	68	75	78	83	65	65	76	63	75	0	66

Table 17: **Eigenvalue computation with LSTM and GRU.** 5×5 matrices.

D.2 SHARED-LAYER TRANSFORMERS: UNIVERSAL TRANSFORMERS

In the Universal Transformer (Dehghani et al., 2018), the stacked layers of usual transformer implementations are replaced by one layer that is looped through a fixed number of times (feeding the output of one iteration into the input of the next). This amounts to sharing the weights of the different layers, therefore greatly reducing the number of parameters in the model. This technique can be applied to the encoder, the decoder or both. The number of iterations is a fixed hyperparameter, but the original paper also proposed a halting mechanism inspired by Adaptive Computation Time (Graves, 2016), to adaptively control loop length at the token level. In this version, a stopping probability is learned for every position in the input sequence, and once it reaches a certain threshold, the layer merely copies the input onto the output. The iteration stops when all positions have halted, or a specific value is reached. A recent paper (Anonymous, 2022) proposed to use a similar copy-gating mechanism to skip iterations in a fixed-length loop. We experiment with these three variants (fixed length, adaptive halting, copy gating) on the addition (of 10×10 matrices), eigenvalue and matrix inversion tasks (5×5 matrices).

For the addition task, we train universal transformers with one layer and in the encoder and decoder, 256 or 512 dimensions and 8 attention heads. We use the B1999 encoding for the data. We experiment with looped encoder, looped decoder, and loop in both, a loop length of 4, copy-gating and ACT (the 4 loops in then a maximum number of iterations)and copy-gating. Table 18 summarizes our findings. Only models with encoder loops learn to add, and models with 512 dimensions learn with over 95% accuracy for all tolerances. Universal Transformers with one layer (looped-encoder only) perform as well as 2/2 transformers.

	5%	2%	1%	0.5%
Looped encoder				
256 dimensions, 4 loops	15	1	0	0
512 dimensions, 4 loops	100	100	100	100
256 dimensions, 4 loops, gated	97	66	41	29
512 dimensions, 4 loops, gated	100	100	100	100
256 dimensions, 4 loops, ACT	100	92	76	66
512 dimensions, 4 loops, ACT	100	100	98	96
Looped decoder	0	0	0	0
Looped encoder and decoder	0	0	0	0
2/2 transformer (baseline)	100	100	100	100

Table 18: **Accuracy of Universal transformers on matrix addition for different tolerances.** 10×10 matrices.

On the eigenvalue task, we experiment on the P1000 and FP15 encoding, with encoder-loop only 1/1 Universal Transformers with 4 or 8 loops. Universal transformers using the P1000 encoding achieve the same performances (with only one layer) than the transformers in our main research 4 loop transformers seem best, with gates not improving perfomance and ACT slightly degrading it. With the FP15 encoding, universal transformers become very difficult to train: only the 4 loop gated version achieves significant accuracy (still lower than the 6/1 transformers).

Finally, we experimented with matrix inversion, with FP15/P1000 and P1000/P1000 encodings, and 4 or 8 loops in the encoder. A gated universal transformer using FP15 in the input and P1000 in the output achieved 73% accuracy, a significant result albeit lower than the best result acieved with $6/1$ transformers using the same encodings (90%). With the P1000 encoding, the best universal transformers reach 55% accuracy, compared to 80% for their 6/1 transformer counterparts. Overall, Universal Transformers seem to achieve comparable performances with deep transformers (except on the inversion tasks), using less parameters. This makes shared layer transformers an interesting direction for future work.

	5%	2%	1%	0.5%
P1000				
4 loops	100	100	97	87
8 loops	100	99	93	69
4 loops, gated	100	100	98	91
8 loops, gated	100	100	99	90
4 loops, ACT	100	97	89	62
8 loops, ACT	100	95	77	42
FP15				
4 loops	4	0	0	0
8 loops	0	0	0	0
4 loops, gated	94	84	57	23
8 loops, gated	6	1	0	0
4 loops, ACT	4	0	0	0
8 loops, ACT	4	0	0	0
4/1 transformer (P1000 baseline)	100	100	99	89
6/1 transformer (FP15 baseline)	100	100	100	92

Table 19: **Accuracy of Universal transformers on eigenvalue computation for different tolerances.** 5×5 matrices.

E ADDITIONAL EXPERIMENTS

E.1 NOISY DATA

Experimental data is often noisy, and it is interesting to see how our models behave in the presence of random error. To this effect, we trained models to perform matrix addition and eigenvalue computations on noisy data, by adding a random gaussian error to all coefficients in the input (5×5) matrices. Three levels of noise were tested, with standard deviation equal to $1, 2$ and 5% of the standard deviation of the matrix coefficients ($\sigma = 5.77$ for uniform coefficients in $[-10, 10]$). For linear operations like addition, we expect the model to predict correct results so long tolerance τ remains larger than error.

Table 20 demonstrates that models can be trained on noisy data without loss of accuracy, so long the ratio between the standard deviation of error and that of the coefficients is lower than tolerance. Accuracy drops to about 40% when error levels are approximately equal to tolerance, and to zero once error exceed tolerance. It is worth noticing that model size and encoding have no apparent impact on robustness to noise.

	B1999				P1000			
	2/2 layers		4/4 layers		2/2 layers		4/4 layers	
	256	512	256	512	256	512	256	512
5% tolerance								
0.01σ error	100	100	100	100	100	100	99.4	100
0.02σ	100	100	99.8	100	100	100	100	100
0.05σ	41.5	41.2	41.7	41.6	39.3	41.2	39.4	40.7
2% tolerance								
0.01σ error	99.8	99.9	99.8	99.9	99.4	100	98.2	99.9
0.02σ	43.7	44.2	42.1	44.7	39.0	44.9	42.6	45.3
0.05σ	0	0	0	0	0	0	0	0
1% tolerance								
0.01σ error	39.8	41.7	39.6	44.0	36.6	44.0	28.9	44.6
0.02σ	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
0.05σ	0	0	0	0	0	0	0	0

Table 20: **Accuracy of noisy matrix addition for different error levels and tolerances.** 5×5 matrices.

A similar pattern appears in eigenvalue calculations (table 21), but trained models prove more resistant to noise in the data than for addition. For instance, the eigenvectors of matrices with error standard deviation up to 0.05σ can be learnt to high accuracy within 5% tolerance (vs 0.02σ for addiition). As before, model size has no impact on robustness. However, FP15 models seem more difficult to train over noisy data than P1000.

	FP15				P1000			
	4/4 layers		6/6 layers		4/4 layers		6/6 layers	
	512	1024	512	1024	512	1024	512	1024
5% tolerance								
0.01σ error	6.1	100	5.1	6.0	100	100	100	100
0.02σ	100	100	6.7	100	100	100	100	100
0.05σ	99.1	99.3	99.3	6.4	99.3	99.0	99.0	98.8
2% tolerance								
0.01σ error	0.7	99.8	0.5	0.8	99.3	99.6	99.9	99.8
0.02σ	97.0	97.1	0.8	88.4	97.3	97.9	93.1	95.4
0.05σ	37.9	38.4	40.6	0.5	40.1	37.3	37.5	35.3
1% tolerance								
0.01σ error	0.1	82.1	0.1	0.2	79.7	83.8	87.9	83.8
0.02σ	47.8	51.3	0.1	26.1	46.2	47.5	36.4	41.3
0.05σ	3.8	4.2	4.1	0.1	4.1	3.8	3.9	3.4

Table 21: **Accuracy of noisy eigenvalue computations, for different error levels and tolerances.** 5×5 matrices.

E.2 CO-TRAINING

We have shown that transformers can be trained to performed all the tasks mentioned above, training one specific model for each task. In this section, we experiment with co-training: learning several tasks at once. We add a token at the beginning of the input and output sequence indicating the task to be solved (e.g. `Transpose` or `Add`), and generate data by randomly selecting a task (with equal probability for all tasks) and producing the corresponding pairs.

We train transformers with 4 or 6 layers, 512 dimensions and 8 attention heads on eight datasets corresponding to different co-training tasks:

- Transpose and add (TA)
- Transpose, add and dot product (vector matrix multiplication) (TAD)
- Transpose, add, dot product and matrix multiplication (TADM)
- Transpose, add, dot product, matrix multiplication and eigenvalues (TADME)
- Transpose, add, dot product, matrix multiplication, eigenvalues and eigenvectors (TADMEF)
- Transpose, add, dot product, matrix multiplication, eigenvalues, eigenvectors and matrix inversion (TADMEFI)
- Eigenvalues, eigenvectors and matrix inversion (EFI)

Table 22 summarizes our findings. Lines correspond to a co-training tasks, columns to the performance achieved on this specific task (with 5% tolerance). Co-training over a mixture of basic operations (transposition, addition, dot products and multiplication: the TA, TAD and TADM tasks) learn to predict the results of all operations with almost perfect accuracy. Co-training on the basic operations and eigenvalue computations (the TADME task) allows the model to predict eigenvalues with 80% accuracy, in exchange for a loss of performances on the dot product task. In other experiments with this task, the model learned all basic operation to 100% accuracy(as in the TADM setting), and the eigenvalue to a few percents. Adding more tasks, eigenvectors and inversion, results in the same performance. Co-training on the advanced tasks only (eigenvalues, vectors and inversion) results in 100% accuracy on eigenvalue computation, 22% on eigenvectors, and 0 on inversion. These results demonstrate the feasibility of co-training on basic matrix operations, but also suggest that further research is needed if one wants to extend it to all the tasks considered in this paper.

	T	A	D	M	E	F	I
TA	100	100					
TAD	100	100	100				
TADM	100	100	100	100			
TADME	100	100	26	100	80		
TADMEF	100	100	100	100	3	0	
TADMEFI	100	100	100	100	3	0	0
EFI					100	22	0

Table 22: **Accuracy of co-training.** 5×5 matrices, 5% tolerance.

F   NUMBER OF PARAMETERS

The number of parameters in the sequence to sequence transformer we use in this paper can be calculated as follows.

- A self-attention mechanism with dimension d has $4d(d + 1)$ parameters: it is composed of four linear layers (K, Q, V and the output layer), with d input, d output and a bias.

- A cross-attention mechanism with d_e dimensions in the encoder, and d in the decoder has $2d(d + d_e + 2)$ parameters (K and V are $d_e \times d$ layers).

- A FFN with one hidden layer, d input and output, and h hidden units has $d(h+1)+h(d+1)$ parameters.

- A layer normalization with d dimensions has $2d$ parameters.

- An encoder layer with dimension d has a self-attention mechanism, a FFN with $4d$ hidden units (in our implementation) and two layer normalizations, for a total number of parameters of $12d^2 + 13d$.

- A decoder layer has a cross-attention layer (encoding dimension d_e) and a layer normalization on top of an encoder, for a total of $14d^2 + 19d + 2d_e d$ parameters.

- An embedding of dimension d for a vocabulary of w words will use dw parameters, and $2d$ more if it is coupled to a layer normalization.

- The final prediction layer with an output dimension of d and a decoded vocabulary of w words will use $(d + 1)w$ parameters (but in our case, dw will be shared with the decoder embedding).

Overall, the number of parameters for a transformer with n_e encoding layers with dimension d_e, $n * d$ decoding layers with dimension d_d, an input vocabulary of w_i words, an output vocabulary of w_o words and a positional embedding of w_p words (corresponding to the maximum sequence length) can be computed by the formula:

$$P = d_e(w_i + w_p + 2) + ((w_o + w_p + 2)d_d + w_o) + n_e d_e(12d_e + 13) + n_d d_d(14d_d + 2d_e + 19)$$

the four terms in the sum corresponding to the input embedding, the output embedding, the encoder and the decoder.

Table 23 provides the number of parameters for some of the models used in this paper. For the positional embedding, we set the number of words as the longest input and output sequence studied with that model.

Experiment	Model	Parameters
Transposition	1/1 layers 256 dims P10	2,276,171
	1/1 layers 256 dims P1000	2,737,871
	1/1 layers 256 dims B1999	3,297,554
	1/1 layers 256 dims FP15	17,045,441
Addition	2/2 layers, 512 dims, B1999	17,619,218
Matrix vector multiplication	2/2 layers 512 dimensions P10	15,578,443
	2/2 layers 512 dimensions P1000	16,500,943
	4/4 layers 512 dimensions P1000	31,213,775
Matrix multiplication	1/4 layers 512 dimensions P1000	21,756,623
	1/6 layers 512 dimensions P1000	30,164,687
Eigen decomposition	1/6 layers 512 dimensions FP15	58,751,937
	6/1 layers 512 dimensions FP15	53,493,697
	6/1 layers 512 dimensions P1000	24,906,447
	661 layers 512 dimensions P1000	45,926,607
Matrix inversion	6/1 layers 512 dimensions FP15/P1000	39,186,127

Table 23: **Number of parameters of transformers used in the paper.**

G   EIGENVALUE DISTRIBUTION OF WIGNER MATRICES, AN EMPIRICAL JUSTIFICATION

Figure 3 provides an empirical confirmation of the property of Wigner matrices mentioned in sections 2.2 and 5: the standard deviation of their eigenvalues is a function of their dimension and standard deviation of their coefficients only, and does not depend on the actual dsitribution of the coefficient. In particular, for coefficients with standard deviation $\sigma = 10/\sqrt{(3)} = 5.77$, we expect the standard deviation of their eigenvalue distribution to be $\sigma = 12.91, 18.26, 22.36$ and 25.81 for square matrices of dimension $5, 10, 15$ and 20.

For three distributions, uniform, Laplace and gaussian, and four dimensions ($5, 10, 15$, and 20), we generated 100 000 random matrices with the same standard deviation of coefficients, and computed their eigenvalues. Standard deviations are within 0.01 of theoretical values for all distributions and dimensions. It is interesting to note how the distributions (which correspond to the original coefficient distribution for $n = 1$) resemble the semi-circle as dimension increases.
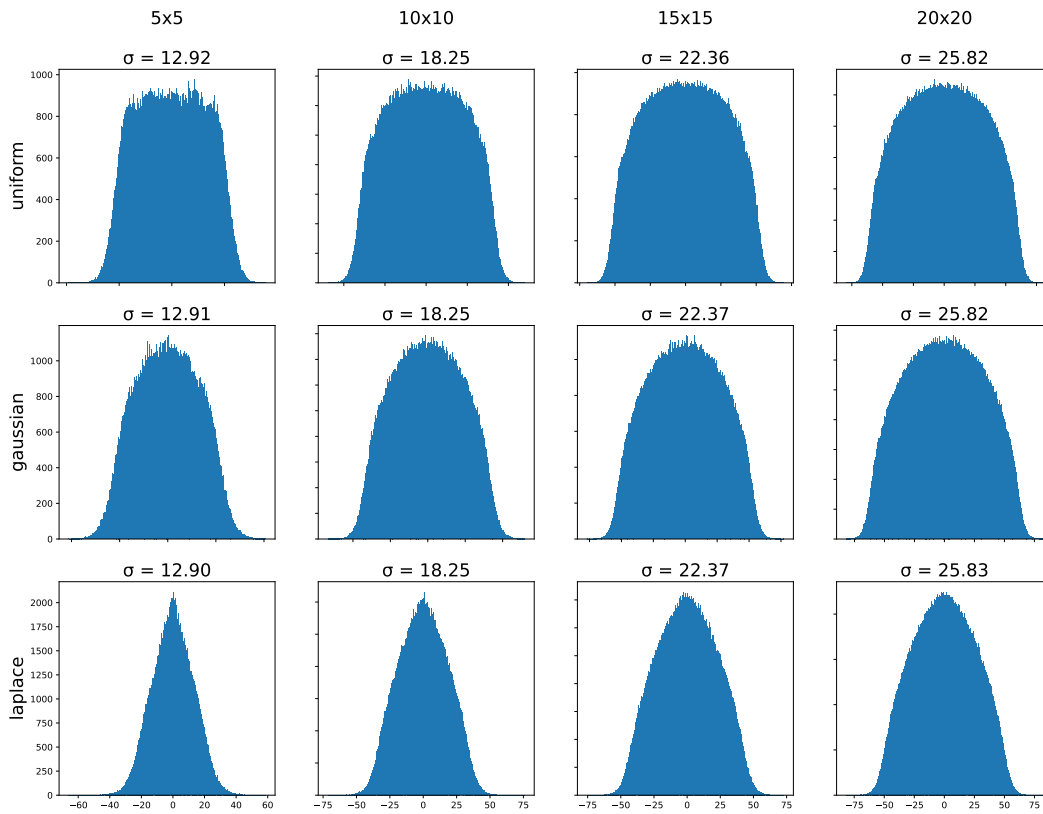
Figure 3: Empirical distributions of eigenvalues for Wigner matrices of dimension 5x5 (left) to 20x20 (right), with uniform (top), gaussian (middle) and Laplace (bottom) coefficients. All distributions computed from 100 000 random matrices.