Int2Int: a framework for mathematics with transformers

François Charton
FAIR, Meta – Ecole Nationale des Ponts et Chaussées
fcharton@meta.com

March 24, 2025

Abstract

This paper documents Int2Int, an open source code base for using transformers on problems of mathematical research, with a focus on number theory and other problems involving integers. Int2Int is a complete PyTorch implementation of a transformer architecture, together with training and evaluation loops, and classes and functions to represent, generate and decode common mathematical objects. Ancillary code for data preparation, and Jupyter Notebooks for visualizing experimental results are also provided. This document presents the main features of Int2Int, serves as its user manual, and provides guidelines on how to extend it. Int2Int is released under the MIT licence, at `https://github.com/f-charton/Int2Int`.

1 Introduction

1.1 Int2Int: Mathematics as a translation task

The transformer architecture [21], introduced in 2017 for machine translation, can be applied to problems of mathematics by rewriting problems and their solutions as sentences – sequences of words from a finite vocabulary – and training transformers, from generated examples, to "translate" the sequence representing a problem into the sequence representing its solutions. For instance, a transformer can be trained to add integers, by learning to translate a pair of integers, say 12 and 23, represented as a sequence of digits in base 10, `+,1,2,+,2,3`, into the sequence `+,3,5`, which represents their sum.

Previous research suggests that this approach can be applied to a wide range of mathematical problems, from symbolic integration [16] to arithmetic [19, 17, 5] and computing the eigenvalues of real symmetric matrices [4]. More recently, such techniques were used to solve hard and open problems, like symbolic regression [3, 9], discovering the Lyapunov functions that govern the global stability of dynamical systems [1] and generating competitive candidate solutions of hard combinatorial problems [6]. These results illustrate the potential use of transformers to assist mathematical discovery.

Applying transformers to a specific problem of mathematics can be a challenging task. While Python implementations of transformers can be found on open source repositories, such as HuggingFace [22], adapting these architectures to specific problems, generating training and test sets

of problems and solutions in the sequential format that transformers can process, and assembling the various part into a training and evaluation loop that can be run in reasonable time on a GPU-equipped machine, require specialised engineering skills that may be hard to come by in mathematical communities. Besides, navigating the many hyper-parameters that need to be tuned for a model to learn, running experiments and interpreting their results require background knowledge about machine learning. All these requirements compound into a high cost of entry, which may discourage interested mathematicians.

Int2Int strives at alleviating these constraints and flattening the learning curve, by providing a ready-to-use transformer framework which can be applied to a broad range of mathematical problems. It is specially targeted to problems that can be formulated as mappings from one sequence of integers onto another. For instance, predicting the rank and torsion of an elliptic curve from its parameters, or the n-th term in a sequence from the $n-1$ first. It also will work out of the box when the problems and solutions, already represented as sequences of tokens, are provided as an external file.

This paper serves as the user manual of Int2Int. After a brief introduction to supervised machine learning (sec. 1.2), and a description of the main components in Int2Int (sec. 1.3), I provide a hands-on tutorial on two problems: predicting properties of elliptic curves, and computing the greatest common divisor of two integers (sec. 2). Section 3 serves as the user manual, and presents the command-line parameters of Int2Int. Finally, section 4 provides guidelines for adapting the framework to new problems of mathematics.

As all open source software, Int2Int is a work in progress, and so is this document. The current version is 1.0.

1.2 Supervised training in a nutshell

Deep learning models learn from examples. In the supervised setting, the model is trained from pairs of problems and solutions (x, y), represented as sequences of tokens over a finite vocabulary. The model implements a mapping from input to output sequences, $\mathcal{M} : x \to y$, which depends on a large number (millions to billions) of parameters, the model *weights* w.

During training, inputs x_i are grouped into *batches* and fed into the model, which produces as many predicted outputs $\hat{y}_i = \mathcal{M}(x_i)$. These output are compared to the desired output y_i, using a differentiable loss function $\mathcal{L}(\hat{y}_i, y_i)$, and the *gradient* $\nabla_w \mathcal{L}(\hat{y}_i, y_i)$, the vector of partial derivatives of the loss with respect to the model weights, is automatically computed and averaged over the batch. The gradient is used to update the model weights, so as to reduce the training loss on the current batch. At this point, a new batch of input is created, and a new *optimisation step* begins.

Model weights, initialised at random, are updated after every optimisation step, hopefully allowing the model to learn, i.e. better predict the output. This is evaluated by calculating the model accuracy (i.e. the percentage of correct predictions), on a *validation set* of test examples that were not used during training. Evaluation happens once the model has seen a fixed number of training examples (300,000 by default), which we call an *epoch*. (Note: in other works, an epoch is defined as "one pass over the whole training set". This makes little sense in AI for Mathematics, where

training data are typically generated on the fly. For this reason, I define an epoch as a fixed number of examples, and make it a model hyper-parameter.) This process, a succession of optimisation steps, followed by a model evaluation (henceforth, the *training loop*) is repeated either for a fixed number of epochs, or until some accuracy level has been reached, or the experimenter is satisfied (or discouraged) by the results.

1.3 Int2Int in a nutshell

Int2Int, a framework for supervised learning, is made of several components.

The model is a parametric function that maps input to output sequences. It uses a cross-entropy loss function to measure the discrepancy between its predictions and the correct solutions, and can calculate the gradient of the loss with respect to its parameters (i.e. model weights). Int2Int currently implements transformers, LSTM [15] and GRU [7]. The model code can be found in files `src/model/transformer.py` and `src/model/lstm.py`.

The optimizer is in charge of updating the model weights after a batch has been processed. It uses the current value of the gradient (calculated by the model) as well as previous values, to compute a direction in the space of model parameters, and a *learning rate* to quantify the length of the "step down" along the direction of update. Different optimizers (SGC, Adam, AdamW) implement different techniques for selecting the direction of update. The optimizer is also in charge of varying the learning rate as training proceeds, a process known as *warm-up* at the beginning of training, when the learning rate linearly grows from zero to a fixed value, and *scheduling* in later stages, when the learning rate is gradually reduced as the model achieves better accuracy. In Int2Int, optimizers are implemented in the file `src/optim.py`. Int2Int is compatible with (and heavily relies on) PyTorch optimizers.

The data loader is in charge of reading the training and validation data, assembling it into batches, and feeding it into the model. The data can be found in a text file read by the data loader, or be generated on the fly. Specific data sampling strategies, like curriculum learning (presenting examples in a specific order), or repeating some examples[12], are implemented in the data loader. It is to be found in file `src/dataset.py`.

The trainer implements the main training logic. It loads and save the model, computes the gradient, calls the optimizer, and updates. It is implemented in `src/trainer.py`.

The evaluator implements the evaluation at the end of each epoch. It is a simplified version of the trainer, which computes the model predictions from the test data, but does not calculate the gradient, nor run the optimizer. Instead, it calculates and reports a number of metrics and statistics. It is found in `src/evaluator.py`

The model, optimizer, data loader and trainer are independent of the problem to be solved (up to the choice of hyper-parameters: different problems may require different settings of thel parameters). So long one uses the same architectures (transformers), they can be re-used without modification. The evaluator may require small changes if one wants to calculate specific metrics

(see also section 4.2). If the model is trained from files of pre-generated data (see section 3.4 for the required format), Int2Int can be used as provided. To generate data on a new math problem, two modules of Int2Int may have to be adapted.

The tokenizer converts the mathematical constructs that make up the problems and solutions into sequences of tokens that the transformer can process. Int2Int currently provides tokenizers for integers and arrays of integers. This allows for encoding more complex mathematical objects, like integer polynomials (an array of coefficients), or graphs, an array of binary values (their adjacency matrix) or a list of pairs representing their edges (see section 4.1. Additional tokenizers may be needed for different problems. This is implemented in `src/envs/encoders.py`.

The data generator is in charge of sampling instances of problems and solutions, either to feed the data loader or to save them into a file as future training or validation sets. It also implements prediction verification when several predictions can represent correct predictions. This is implemented in `src/envs/generator.py`.

Two other files, `train.py` and `src/envs/arithmetic.py` contain general-purpose code for running the training loop, and linking problem-specific elements (generators and tokenizers) to the rest of the code base. Many experiments in AI for mathematics rely on synthetic data. Int2Int includes functions for generating datasets, this is discussed in section 3.5.

2 Getting started: two worked-out examples

To get started with Int2Int, please clone Int2Int from the repository at `https://github.com/f-charton/Int2Int` (instructions in the Code button of GitHub). This should create a directory, named Int2Int (you may rename it) that contains (among other things) a file `train.py`, a `README.md` file that partially duplicates these instructions, and two subdirectories: `src`, which contains the source code, and `data` which contains the elliptic function dataset for our second worked-out example. You will also need to have installed `Python` (version 3.0 or better), `PyTorch` (`pytorch.org`), and `NumPy`.

2.1 Out of the box: learning the greatest common divisor of two integers

As a very first attempt running Int2Int, you may run from the `Int2Int` directory on your computer:

```
python train.py --dump_path /some_path_on_your_computer/ --exp_name
    ↪ my_first_experiment --exp_id 1 --operation "gcd"
```

Note: this assumes your computer has an NVIDIA GPU. If it does not, add `--cpu true` to the command line

This will train a transformer to compute the greatest common divisor of two integers. The training and test data are generated on the fly. The parameters `dump_path`, `exp_name` and `exp_id` indicate where the experimental results, and the trained models, will be saved: here in `/some_path_on_your_computer/my_first_experiment/1/`.
If you do not provide an `--exp_id`, Int2Int will generate one for you (a unique sequence of letters

and numbers). Try to provide an absolute path ofr `--dump_path`: relative paths (starting with `./` or `~/`) have bee reported to fail on some systems.

If everything goes right, a log will be displayed on your screen, It will also be saved in the file `/some_path_on_your_computer/my_first_experiment/1/train.log`.

The first lines of the log include:

- A list of the model hyper-parameters (default values in this experiment).
- The vocabulary, the list of symbols that the model can use.
- The number of trainable parameters.

Then the model will output lines of the form

```
INFO - 02/07/25 09:32:03 - 0:00:12 - 200 - 745.83 examples/s - 7457.25 words/s - ARITHMETIC: 0.7744 - LR: 1.0000e-04
INFO - 02/07/25 09:32:10 - 0:00:19 - 400 - 868.08 examples/s - 8679.18 words/s - ARITHMETIC: 0.5370 - LR: 1.0000e-04
INFO - 02/07/25 09:32:17 - 0:00:26 - 600 - 950.02 examples/s - 9499.33 words/s - ARITHMETIC: 0.4251 - LR: 1.0000e-04
INFO - 02/07/25 09:32:23 - 0:00:33 - 800 - 951.52 examples/s - 9514.31 words/s - ARITHMETIC: 0.3710 - LR: 1.0000e-04
INFO - 02/07/25 09:32:30 - 0:00:40 - 1000 - 952.58 examples/s - 9524.71 words/s - ARITHMETIC: 0.3344 - LR: 1.0000e-04
```

These log the time, elapsed time since the program was launched, number of optimization steps run (in multiples of the parameter `--report_loss_every`, 200 by default), tnumber of training examples, and tokens, processed per second (on average), training loss, and learning rate. Here, the model is processing a little less than 1000 examples per second, indicating that an epoch (300,000 examples by default) will be completed in a little more than 5 minutes. The learning rate does not change over time (because we did not use warmup or scheduling). Finally, and most importantly, the loss is decreasing, which indicates that the model is indeed learning. On average the loss should decrease, and remain stable once the model does not learn anymore. An increasing loss is the tell-tale sign that something is wrong (usually a bug in the code). These logs are written by function `print_stats()`, in `src/trainer.py`.

At the end of each epoch, defined by the parameter `--epoch_size` (300,000 examples by default), the model is evaluated on a test set of size `--eval_size` (10,000 by default). Test examples are evaluated in batches of `--batch_size_eval` (128 by default), and the model reports the number of correct solutions in each batch (here 110 and 104 out of 128).

```
INFO - 02/07/25 09:37:51 - 0:06:00 - (128/10000) Found 110/128 valid top-1 predictions.  Generating solutions ...
INFO - 02/07/25 09:37:51 - 0:06:00 - Found 110/128 solutions in beam hypotheses.
INFO - 02/07/25 09:37:51 - 0:06:00 - (256/10000) Found 104/128 valid top-1 predictions.  Generating solutions ...
INFO - 02/07/25 09:37:51 - 0:06:00 - Found 104/128 solutions in beam hypotheses.
```

After all evaluations are performed, a short report is printed. Here, the model correctly predicts 84.6% of the test GCD. GCD 1, 2, 4, 5, 8 .... (products of powers of divisors of the base) are almost always correctly predicted.

```
INFO - 02/07/25 09:37:53 - 0:06:02 - 8323/10000 (83.23%) examples were evaluated correctly.
INFO - 02/07/25 09:37:53 - 0:06:02 - 1:  5948 / 5949 (99.98%)
INFO - 02/07/25 09:37:53 - 0:06:02 - 2:  1591 / 1593 (99.87%)
INFO - 02/07/25 09:37:53 - 0:06:02 - 4:  371 / 375 (98.93%)
INFO - 02/07/25 09:37:53 - 0:06:02 - 5:  244 / 244 (100.00%)
INFO - 02/07/25 09:37:53 - 0:06:02 - 8:  93 / 93 (100.00%)
```

```
INFO - 02/07/25 09:37:53 - 0:06:02 - 10:  63 / 63 (100.00%)
INFO - 02/07/25 09:37:53 - 0:06:02 - 20:  10 / 10 (100.00%)
INFO - 02/07/25 09:37:53 - 0:06:02 - 40:  3 / 3 (100.00%)
```

Detailed evaluation metrics (see section 4.2) are also logged into a python dictionary, which can be loaded into a notebook to draw learning curves, etc. (see section 5).

Notes: If you run this experiments on a CPU-only machine (or if you want to ascertain the benefit of a GPU), you must add `--cpu true` to the command line. The training will be slower, by a factor of four for such a small model (it would be much worse for a larger model). You may want to set `--epoch_size` to a smaller value (e.g. 75,000 to keep running an epoch in about 6 minutes).

In this experiment, integers are encoded in base 1000, to reproduce the initial experiments from [5] (section 2), you can change the parameter `--base` to different values. The parameter `--operation` controls the problem you are solving (see `src/envs/arithmetic.py`). You can try other operations, such as `modular_add`, `modular_mul`, `fraction_compare`, `fraction_add`, `fraction_simplify`. Modular operations use modulo 67 by default, you can change this using the parameter `--modulus`.

2.2 Out of the box: predicting the rank of elliptic curves

Int2Int can also be run from a pre-computed dataset. The data files (one train set, one test set) should be text files, with one example per line, input and output, separated by a tab key, and written as tokens separated by spaces. See section 3.4 for more information about data files and how to produce them. In this example, we train a model on a dataset of elliptic curves, extracted from the LMFDB database [18]. The dataset can be found in the subdirectory `/data`, and contains 1,010,000 elliptic curves, defined by the five parameters in their Weierstrass equation $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$ and the ranks of the associated Mordell-Weil group. The train set has one million curves, the test set 10,000. The first four lines of the training set are:

```
+ 1 + 0 + 1 + 516 - 8 902 2
+ 1 + 0 + 0 - 3 881 - 72 555 0
+ 0 + 0 + 0 - 502 891 875 + 4 340 702 513 250 1
+ 0 + 0 + 0 - 138 477 + 19 834 180 2
```

The input are the five parameters of the elliptic curve, written in base 1000, and the output is the rank: $1, 0, 1, 516, -8902$ and 2 for the first curve, $0, 0, 0, -502891875, +4340702513250$ and 1 for the third.

To train a transformer to predict the rank of elliptic curves we run, from the `Int2Int` directory:

```
python train.py --operation data --dump_path /some_path_on_your_computer
--exp_name my_second_experiment --exp_id 1
--train_data /data/elliptic_rank.train --eval_data /data/elliptic_rank.test
```

As before, the model reports a training loss that drops, then saturates around 0.5.

```
INFO - 02/07/25 10:58:48 - 0:00:17 - 200 - 482.90 examples/s - 6903.34 words/s - ARITHMETIC: 0.6622 - LR: 1.0000e-04
INFO - 02/07/25 10:58:56 - 0:00:25 - 400 - 802.48 examples/s - 11466.61 words/s - ARITHMETIC: 0.5171 - LR: 1.0000e-04
INFO - 02/07/25 10:59:04 - 0:00:33 - 600 - 804.41 examples/s - 11495.62 words/s - ARITHMETIC: 0.5206 - LR: 1.0000e-04
```

```
INFO - 02/07/25 10:59:12 - 0:00:41 - 800 - 804.38 examples/s - 11476.53 words/s - ARITHMETIC: 0.5123 - LR: 1.0000e-04
INFO - 02/07/25 10:59:20 - 0:00:49 - 1000 - 802.78 examples/s - 11469.85 words/s - ARITHMETIC: 0.5041 - LR: 1.0000e-04
INFO - 02/07/25 10:59:28 - 0:00:57 - 1200 - 803.16 examples/s - 11487.58 words/s - ARITHMETIC: 0.5119 - LR: 1.0000e-04
```

After one epoch, the model correctly predicts the rank of 48.16% of elliptic curves in the test set. Curves of rank 0 are correctly predicted in 29.9% of test cases, rank 1 curves in 77.4%. After 100 epochs, the model achieves 49.9% accuracy, with rank 1 curves correctly predicted in 88.7% of test cases, rank 0 in 14.6, and rank 2 in 4.1%. Since rank 1 curves account for 49.6% of the test set, the model does slightly better than a "modal predictor", which always predicts the most common outcome.

3 Using Int2Int: command-line parameters

This section serves as the user manual for Int2Int. All parameters have default values (stated in the source code). Boolean parameters are not flags, i.e. they must be set as `--bool_param true`, or `--bool_param false` (not `--bool_param`).

3.1 Base parameters

`--dump_path`, `--exp_name` and `--exp_id` define the directory where experimental results are saved: `dump_path/exp_name/exp_id`. Absent directories will be created. If `--exp_id` is not specified, Int2Int will generate a random sequence of 10 letters and digits (cf. function `get_dump_path` in `src/utils.py`). When training begins, Int2Int will create two files in this directory: `params.pkl`, which contains the command-line parameters of the model, and `train.log`, which contains the experimental log and results. At the end of every epoch, the model will be saved as `checkpoint.pth`.

If the folder `dump_path/exp_name/exp_id` already exists, Int2Int will try to continue an existing experiment. It will reload the last saved model (`checkpoint.pth`) if it exists (alternatively, a path to another checkpoint can be provided as parameter `--reload_checkpoint`), and append to the log file (`train.log`).

Alternatively, the model can be initialised with a pre-trained model, by indicating its path in `--reload_model`. In this case, a new log file is created, and training starts afresh.

Saved models take a large amount of disk space. For this reason, only the last epoch is saved by default. The parameter `--save_periodic` allows for more regular saves. If set to 100, it will cause an additional save file to be created at epochs 100, 200, etc. (as `checkpoint-100.pth`, `checkpoint-200.pth`, etc.). The parameter `--validation_metrics` causes Int2Int to save the best models according to some evaluation metric. For instance, a `validation_metrics` with `valid_arithmetic_acc` would cause Int2Int to save the models with the highest accuracy on the validation set. To save the lowest value of the metric, instead of the highest, prefix the name of the indicator with `_`: `_test_arithmetic_xe_loss` saves the models with the lowest test loss. Several validation metrics can be used at the same time, by concatenating them, separated by commas: `valid_arithmetic_acc,_valid_arithmetic_xe_loss`. All available metrics, are logged at the end of each epoch, into a python directory. For the elliptic curve example, they are: `valid_arithmetic_xe_loss, valid_arithmetic_acc, valid_arithmetic_perfect,`

valid_arithmetic_correct, valid_arithmetic_acc_0; valid_arithmetic_acc_1, valid_arithmetic_acc_2, valid_arithmetic_acc_3, valid_arithmetic_acc_4. See section 4.2 for a discussion of metrics.

Int2Int will run until stopped (by killing the process), or for `--max_epoch` epochs (set by default to 100,000). If computing resources are available, it is usually advisable to let a model run for as long as possible, check its results periodically, and stop it once accuracy (or any metric you are interested in) stops improving. Alternatively, a `--stopping_criterion` can be defined, which will cause Int2Int to end after a given metric did not improve for a given number of epochs. For instance, the stopping criterion `valid_arithmetic_acc,100` will cause Int2Int to stop when the model accuracy, computed on the validation set, has not improved during 100 epochs.

`--epoch_size` is the number of training examples in one epoch, `--batch_size` the number of examples in a mini-batch. The ratio of these two quantities is the number of optimisation steps in one epoch. A larger batch size makes learning faster by reducing the number of optimization steps, but it may also hinder learning: since loss gradients are averaged over all examples in the case, a large batch size causes edge cases to be "diluted" into the rest of the batch. Besides, larger batches require more GPU memory, causing fatal out-of-memory errors when the video memory on the GPU is exceeded. Reducing the (training) batch size prevents out-of-memory errors from happening.

`--max_len` defines a maximum length for the input and output sequences. Any training or test examples with length longer than `max_len` will be ignored. This can be useful when some of your training examples are exceptionally long. In that case, all the training batch is padded to the length of the longest example, which may cause the model to request more memory than you have. Setting `max_len -1` disables this control.

3.2 Technical parameters

By default, Int2Int assumes a NVIDIA GPU is present to train the transformer, and that its `device_id` is 0. You can train transformers on a machine without a GPU, by setting `--cpu true`, but the process will be much slower. Evaluation, on the other hand, can be performed on a CPU. Running from a CPU can also be practical when debugging, since GPU parallelization introduces a lot of complications. For a local GPU, the model assumes `device_id=0`. You can change this by setting `--local_gpu` to a different number. Int2Int can also be run on a cluster using `Slurm` (set `--local_rank 0` to run on a single GPU). See `src/slurm.py` for more information on the use of Slurm.

Multi-GPU is supported under Slurm. At present, it allows splitting data batches on several GPU, which share the same copy of the model. In other words, a job run with batches of 200 on four GPU will use an effective batch size of 800. This can be useful when you train large models on long input or output sequence (e.g. 800 tokens or more), or when you work on machines with small GPU memory (8GB or less), and have to reduce the batch size to avoid fatal out-of-memory errors. In this setting, evaluation is performed on one GPU only. Model sharding, which allows a very large model to be split over several GPU, is not supported at the moment.

When training from a GPU, speed and memory usage can be improved by setting `--fp16 true` and `--amp 1`. This will cause all model parameters and gradients to be encoded as 16-bit floats, resulting in a smaller memory footprint and faster execution speed.

To achieve good training speed on a GPU, the data loader, in charge of feeding examples into the GPU, must operate at a fast rate. When training data is generated on the fly (i.e. you do not read data from a file, loaded in memory), you can set `--num_workers` to a value larger than 1, this will cause several CPU cores to generate data in parallel.

Int2Int relies on a random number generator to generate examples, initialise the model, and shuffle its training data. This generator is seeded (in `src/dataset.py/init_rng()`) by three parameters: the id of the data loader worker, so that different workers do not generate the same training examples, the local rank of the gpu, so that in multi-gpu mode different GPU do not use the same examples, and the parameter `--env_base_seed`. Setting `--env_base_seed` to a positive value will cause the experiment to be reproducible. If it is negative, the seed will be initialised at random.

3.3 Evaluation

At the end of each epoch, the model is evaluated, either on data from a file defined in `--eval_data`, or on a generated sample of `--eval_size` examples. Generated test sets are recreated at each epoch. Evaluation is performed in batches of `--batch_size_eval` examples. Because evaluation does not require gradient estimation, it uses less memory than training, and you can set the evaluation batch size to a larger value than the training batch size. With encoder-decoder transformers, you can use beam search by setting `--beam_search true` and `--beam_size` to a value larger than 1. This will cause the model to generate several solutions, and keep the best.

Specifically, when using beam search, the model decodes the output token by token by generating the `beam_size` most likely next tokens, computing the overall probability score of the resulting output, and keeping the `beam_size` most likely output. Thus, the `beam_size` likeliest first tokens are generated, then the `beam_size * beam_size` most likely pairs of 2 tokens, from which the `beam_size` most likely are retained. `beam_size * beam_size` sequences of three tokens are then generated, from which the most likely `beam_size` are retained, and so on, until sequences "terminate", either by outputing a special end-of-sequence character, or reaching a maximal length (set as `--max_output_len`).

With encoder-decoder and decoder-only transformers, evaluation tends to be slow, because the model must be called as many times as there are tokens in the output sequence. If you know the maximal length of the output, consider setting `--max_output_len` to this value plus 2 (for the beginning and end of sequence tokens). This will greatly accelerate evaluation.

By default, aggregated evaluation results are output in the train.log file. If you set `--eval_verbose` to 1 or 2, a file containing model predictions will also be saved at the end of every epoch. With `--eval_verbose_print true`, model predictions will also be written in the log file. If you set `--eval_verbose` to 2, beam search predictions for "perfect answers" (i.e. when the top-1 prediction is the solution from the test file) are exported, if set to 1, the beam is not logged for

perfect predictions.

When problem solutions can only take a small number of positive values (e.g. modular arithmetic with small moduli, greatest common divisors), setting `--export_pred true` will cause Int2Int to report model predictions for all desired outputs up to `--max_class`. This can provide insights about what the transformer is doing [5].

Finally, `--eval_only`, associated with `--reload_model` will cause the model to evaluate the model and return, without training. Alternatively, you can provide the path of the experiment you want to test in `--eval_from_exp`. Int2Int will then reload the model saved as `best_validation.pth`, where validation is the criterion passed as `--validation_metrics`, or `checkpoint.pth` (i.e. the last checkpointed model) if it does not exist.

3.4 Reading from files

Int2Int can read its training and test data from a file. The path to the training file must be passed in `--train_data`. If a training file is used, the `--reload_size` first examples in the file will be used (all examples if `--reload_size` is set to -1). The path to the test data should be defined in `--eval_data`. it is possible to evaluate the model on several test sets, by adding several paths separated by a comma. The first file will be the validation file, the next ones the test files. Evaluation results will be computed on the first `--eval_data_size` examples of all test sets (all examples if `--eval_data_size` is -1), and the performance metrics will be prefixed by the name of the evaluation set (`valid`, `test`, `test2`, etc.). If no path is provided, training and/or test data will be generated on the fly.

By default, all training examples are loaded in memory when Int2Int initialises. On machines with limited memory, this may cause runtime errors, or important slowdowns due to disk caching. Setting `--batch_load true` will cause Int2Int to load the entire training file in batches of `--reload_size` examples. The training examples will be used in order, until all training data is read, and a new batch is loaded.

When `--batch_load` is set to `false`, the data loader creates batches by randomly picking examples from the dataset. By default, examples are uniformly sampled, which means every training example has the same probability of being used. In [12], we showed that increasing repetition on a random subset of training examples can greatly improve model performance. This can be enabled by setting `--two_classes true`. Examples from the first `--first_class_size` of the training set will then be selected with probability `--first_class_prob`. These two parameters allow one to adjust the repetition levels in the two samples.

Int2Int file format. The data files read by Int2Int are plain text files. Each line contains one example, with tokenized input and output separated by a tab key. Tokens are written as strings separated by spaces. For instance, the pair $(10, 12)$ and its GCD 2, written in base 10, would be saved in the training file as `+ 1 0 + 1 2<TAB>+ 2`. In base 1000 it would be represented as `+ 10 + 12<TAB>+ 2`. For Int2Int to be able to process the file, all tokens used must be known to the tokenizer. By default, Int2Int knows all integers from 0 to $B - 1$, where B is the `--base` (1000 by default), the signs `+` and `-`, separators `<sep>`, `(` and `)`, and some special tokens `<SPECIAL_0>`

to <SPECIAL_9>, which you can redefine as you please. The list of symbols known to the system is defined in input_encoder.symbols and output_encoder.symbols, in src/envs/arithmetic.py and src/envs/encoder.py.

3.5 Data generation

When no training or evaluation datasets are specified, the model generates, and tokenizes, its train and test data. The code for data generation must be added to files src/envs/generator.py and src/envs/encoders.py (see sections 4.3 and 4.1). However, a few generators and tokenizers are provided as examples. The problem to be solved is defined by the parameters --operation. At present, 10 arithmetic operations are proposed. All integers are tokenized as sequence of digits in base --base, preceded by a sign token(+ or -) that also serves as a separator.

- matrix_rank: compute the rank of an integer matrix, of dimension --dim1 × --dim2, with entries in [– maxint, maxint] (defined by parameter --maxint.)
- fraction_add: add two fractions, $\frac{a}{b}$ and $\frac{c}{d}$, represented as a sequence of 4 integers from --minint to --maxint, return the sum in lowest terms
- fraction_product: multiply two fractions,
- fraction_simplify: simplify a fraction $\frac{a}{b}$ represented as a sequence of 2 integers from --minint to --maxint,
- fraction_compare: compare two fractions, $\frac{a}{b}$ and $\frac{c}{d}$, return 1 if $\frac{a}{b} > \frac{c}{d}$, 0 else,
- fraction_determinant: given two fractions, $\frac{a}{b}$ and $\frac{c}{d}$, return $ad - bc$,
- fraction_round: given two positive integers $a > b$, calculate the floor of a/b,
- modular_add: calculate the sum of two integers, from --minint to --maxint, modulo --modulus,
- modular_mul: calculate the product of two integers, from --minint to --maxint, modulo --modulus,
- gcd: calculate the greatest common divisor of two integers, from --minint to --maxint

Generated data can be exported to a file by setting --export_data true. This will create a data.prefix file in the log directory, which has the format recognised by Int2Int. To generate large training and test sets, I use the following procedure.

1. Run several instances of Int2Int with: --export_data true --cpu true --num_workers 20 --exp_name my_generation --base_env_seed -1. A large number of workers will accelerate generation. Several files named data.prefix will be created. Their size can be controlled by setting --epoch_size and --max_epochs to appropriate values. Alternatively, you can control the size generated periodically with the shell command wc -l my_generation/*/data.prefix, and kill the generating processes when you reach the file size you desire.

2. Concatenate the generated files into one
   cat my_generation/*/data.prefix > my_generation/data.raw.

3. Shuffle the file (just in case) shuf my_generation/data.raw > my_generation/data.shuf. You may also want to eliminate duplicate examples in the file, using uniq.

4. Split the 10,000 first elements as a validation set
   head -n 10000 my_generation/data.raw > my_generation/data.valid.

5. Split the 10,000 last elements as a test set
   tail -n 10000 my_generation/data.raw > my_generation/data.test.

6. Keep the remaining elements as your train set
   `tail -n +10000 my_generation/data.raw > my_generation/data.raw | head -n -10000 > my_generation/data.train`.

3.6 Model architecture

The default architecture implemented in Int2Int is the sequence-to-sequence transformer [21]. It is configured by default by setting `--architecture encoder_decoder`, and features a bidirectional encoder, and an autoregressive decoder, both multi-layer transformers, connected by a cross-attention layer. In effect, the encoder transforms the input sequence into some latent representation, and the decoder outputs the prediction, token by token, as a function of the latent representation of the input (accessed via the cross-attention) and the previously decoded output.

Transformer stacks depend on three parameters: the number of layers, the dimension of the embeddings (across all layers) and the number of attention heads. The dimension must be a multiple of the number of heads, in BERT and many other implementation, a ratio of 64 between dimension and number of heads is commonly observed. These translate into six parameters: `--n_enc_layers`, `--n_dec_layers`, `--n_enc_heads`, `--n_dec_heads`, `--enc_emb_dim` and `--dec_emb_dim`. There are no constraints on these parameters, apart from the fact that the embedding dimension must be divisible by the number of heads. When scaling models, keep in mind that the number of parameters grows linearly with the number of layers, and quadratically with the dimension (the number of heads has no impact). A general observation in AI for Maths is that small transformers suffice: less than 8 layers, in fact 4 is often enough.

By default, the transformer parameters are initialized with uniform weights (aka Kaiming initialisation). `--xav_init` uses Xavier initialisation. I never saw it make a difference, but the corresponding code (in `src/models/transformer.py`) demonstrates how to modify model initialisation. In a similar vein, the parameters `--gelu_activation` replaces the ReLU activations used in feed-forward networks by GeLU [14], and the parameter `--dropout` can be used to add dropout when training the feed-forward networks (a small amount of dropout e.g. 0.05, sometimes helps stabilise learning).

Finally, the feed-forward networks (FFN) in the transformer layers have one hidden layer by default. Parameters `--n_enc_hidden_layer`, `--n_dec_hidden_layer` increase this number (by convention the hidden layer dimension is always four times the transformer embedding dimension). Since it greatly increases the number of parameters, this parameter must be used with care. When deep FFN are used, dropout is often useful to stabilize learning. A one layer transformer with several hidden layers can be thought of as a "Multi-Layer Perception with attention".

Two parameters act on the transformer self-attention. `--norm_attention` normalises the output of the self-attention, and allows the temperature on the softmax to be trainable. `--attention_dropout` uses dropout when training the attention weights. A few years ago, both of these tricks were rumoured to be part of the "secret sauce" that allowed transformers to achieve better performance. I never noticed any positive impact (but I would be happy to be proven wrong).

Embeddings. Transformers use a token embedding and a positional embedding for the tokens in their input and output sequences. These embeddings transform the discrete tokens and positions into high-dimensional vectors that the transformer can process. The output decoder does the reverse operation, and transforms a sequence of high dimensional vectors into a probability distribution for the next output token. Both embeddings are computed by a trainable linear layer, that takes a one-hot representation of the input, and outputs a real vector. The positional and token embeddings are then summed, and serve as input to the first transformer layer. The role of the positional embedding is to tell the model that the position of a token carries meaning. It can be deactivated, making the transformer permutation invariant, by setting `--enc_has_pos_emb false` and `--dec_has_pos_emb false`. Int2Int makes positional embeddings trainable. Instead, the original transformer used a fixed sinusoidal embedding for token positions. This can be done by setting `--sinusoidal_embeddings true`. In encoder-decoder models, the auto-regressive decoder reuses the linear layer that serves as its token embedding, as its prediction layer. This default behaviour can be deactivated by setting `--share_inout_emb false`.

Shared layers: universal transformers. In deep learning architectures, model inputs are transformed by a succession of layers (hence the term "deep"). In shared layer models, some of the layers use the same weights. In practice, this amounts to feeding back a layer output as input, as in a loop. Such shared-layer transformers were first proposed by Dehghani under the name Universal Transformers [10]. Shared layers can be added to the encoder and decoder by setting the `--enc_loop_idx` and `--dec_loop_idx` parameters to values different from -1. A positive value will cause one transformer layer (0 being the first one) to be shared and iterated through several times. A value of -2 causes all layers to be shared and iterated in order (the default value of -1 means no shared layers). The number of iterations is set in `--enc_loops` and `--dec_loops`.

Setting the number of loops in a shared-layer models can be a challenge. Adaptive Computation Time (ACT) [13] proposed to make it a learnable parameter. This is implemented by setting the parameters `--enc_act` or `--dec_act` to `true` (the other `--act_` parameters correspond to the parameters described in the paper). ACT is notoriously hard to train. An alternative was proposed by Csordas et al. [8], which implements a copy-gate that decides, for each token, and at each iteration of the loop, whether the model representation is copied or processed by the transformer layer. Parameter `--gated` introduces copy-gates for all shared layers. `--enc_gated` and `--dec_gated` add gates for all layers in the encoder and decoder. Other parameters are as per the Csordas paper.

Other architectures. By default, Int2Int implements a sequence-to-sequence – encoder-decoder – architecture. Setting the `--architecture` parameter to `encoder_only` instantiates a BERT-like model [11], where a single transformer stack (with bidirectional attention) encodes the input sequence and the output is decoded by a linear layer. Note that this implies that model outputs are always shorter than input.

Setting `--architecture` parameter to `decoder_only` will instantiate a decoder-only model (e.g. GPT [20]), where input and output are concatenated, and the model is trained to predict the next token. This configuration is a work in progress.

Alternatively, Int2Int may use (sequence-to-sequence) recurrent architectures: either Long Short Term Memories (LSTM) [15] or Gated Recurrent Units (GRU) [7]. This is done via two parameters: setting `--lstm true` will instantiate a recurrent architecture. By default this will be a LSTM, but

setting `--GRU true` will use a GRU instead. The additional parameter `--lstm_hidden_dim` must be set (generally to a dimension larger than the encoder and decoder embedding dimension. At the moment, LSTM/GRU only work with greedy decoding (no beam search).

3.7 Optimizers

During training, the gradient of the (cross-entropy) loss function is accumulated over all examples presented to the model before an optimization step is performed. By default, this is done on one batch of `--batch_size` examples. If the model runs on several GPU, the gradients from each GPU are accumulated, and one optimization step processes n times the batch size (n the number of GPU). For large models processing long sequences, the GPU memory needed to compute gradients may grow large, and cause the batch size to be very low. This can be mitigated by setting `--accumulate_gradients` to a value larger than one. The gradient computations are then done on several batches (i.e. the optimisation step happens every `--accumulate_gradients` batches). In this case, please set `--amp 1`, to allow for adaptative scaling of the gradient (or accumulated gradients may cause overflow errors).

Int2Int can use most of the optimisers from PyTorch, by setting the parameter `--optimizer` to the name of the optimizer, and a comma-separated list of its parameters. For instance, `adam,lr=1e-4`, `sgd,lr=0.01` or `adagrad,lr=0.1,lr_decay=0.05`. A list of supported optimizers can be found in function `get_optimizer()` of `src/optim.py`.

Extreme values of gradients may cause overflow errors, to prevent these, you may set `--clip_grad_norm` to a low value (e.g. 5.0).

4 Extending Int2Int: generators, verifiers and tokenizers

This section deals with adapting Int2Int to a new math problem (for which you do not have precalculated training data). This requires adding, or modifying, the following components.

- The generator, a descendent of class `Generator` or `Sequence`, from `src/envs/generators.py`, which is responsible for generating pairs of problems and solutions (class function `generate()`) and verifying model predictions (class function `evaluate()`).
- Two tokenizers, descendants of class `Encoder` in `src/envs/encoders.py` for the input and output, which are responsible for encoding problems and solutions into sequences of tokens (function `encode()`), and decoding sequences into problems and solutions (function`parse()`).

Once a generator and two tokenizers are defined, all you have to do is to define them in the `__init__()` function of `ArithmeticEnvironment` (in `src/env/arithmetic.py`), by such code as:

```
if params.operation == 'my_operation':
     self.generator = generators.MyGenerator(params)
     self.input_encoder = encoders.MyIEncoder(params)
     self.output_encoder = encoders.MyOEncoder(params)
```

4.1 Tokenizers

The tokenizers already present in Int2Int will prove sufficient if your input and output are either integers (or elements of a finite set that you can map to integers), or arrays of integers (vectors,

matrices or tensors, or anything that can be mapped to them). Integers can be encoded either as `SymbolicInts` or `PositionalInts`. `SymbolicInts(min,max,prefix='')` are a finite set of tokens, mapped to integers from `min` to `max` (inclusive), and potentially prefixed by a letter. For instance, binary tokens would be tokenized as `SymbolicInts(0,1,'')`, integers between -10 and 10 as `SymbolicInts(-10,10,'')`, and the nodes of a graph with 20 nodes (N1 to N20) could be encoded as `SymbolicInts(1,20,'N')`. `PositionalInts(base)` are integers represented as strings of digits in base `base`, and prefixed by a sign. In base 10, 1024 would be tokenized as `+ 1 0 2 4`.

Arrays of integers (symbolic or positional) are tokenized as `NumberArray(params, max_dim, dim_prefix, tensor_dim, code)`. `code` indicates how the array elements are encoded (`pos_int` or `symbolic`). `tensor_dim` is the dimension of the array (1 for vectors, 2 for matrices). The array will be prefixed by `tensor_dim` tokens indicating its dimensions: a vector of 5 elements will be prefixed by `V5`, a 4×5 matrix by `V4 V5`. `dim_prefix` is the prefix for the dimension tokens (here 'V'), and `max_dim` is the maximum size of any dimension. Finally, params include information needed to encode the coefficients: `params.base` for positional integers, `params.min_int` and `params.max_int` for symbolic integers.

For instance, the 2×3 matrix

$$\begin{bmatrix} 10 & 12 & 23 \\ 44 & 56 & 35 \end{bmatrix}$$

would be tokenized by `NumberArray(params, 100, 'V', 2, 'symbolic')`, with `params.min_int=0` and `params.max_int=10`, as the sequence `V2 V3 10 12 23 44 56 35`. Encoded as positional integers in base 10, `NumberArray(params, 100, 'V', 2, 'pos_int')`, with `params.base=10`, it would be `V2 V3 + 1 0 + 1 2 + 2 3 + 4 4 + 5 6 + 3 5`.

Integers array are, of course, not limited to vector and matrices. A graph with N nodes could be represented by its adjacency matrix, `NumberArray(params, N, 'V', 2, 'symbolic')`, with `params.min_int=0` and `params.max_int=1`. It could also be represented as a list of edges, each edge a pair of nodes, so a vector of length $2E$ (E the number of edges), of nodes between 0 and N. `NumberArray(params, 2E, 'V', 1, 'symbolic')`, with `params.min_int=1` and `params.max_int=N`.

4.2 Verifiers and additional metrics

During evaluation, test examples are first evaluated by comparing the predicted sequence with the target sequence. The number of such *perfect matches* is reported as `valid_arithmetic_perfect`. If the match is not perfect, the function `src/envs/arithmetic.py/check_prediction()` is called, which first verifies that the predicted sequence decodes as a valid mathematical object (i.e. an integer, or a sequence of integers, and not a random sequence such as `+ + 1 -`). The number of well-formed predictions (correct or not), is reported as `valid_arithmetic_correct`. Finally, it calls `src/envs/generator.py/evaluate()`, which returns 0 if the prediction is incorrect, and 1 if it is correct. The perfect and correct predictions are reported as `valid_arithmetic_acc`. Note that when the output sequence associated to the solution to the problem is unique, correct solutions are perfect solution. In that case, `evaluate()` must always returns 0 (this is the default behaviour).

`perfect`, `correct` and `acc` are the three basic evaluation metrics in Int2Int. Additional problem-specific metrics can be introduced by calculating them in `evaluate()`. It returns 2 lists, the size of which are defined in `--n_eval_metrics` and `--n_error_metrics`. All these metrics take values in $[0, 1]$, are summed over all test examples, and reported as
`valid_arithmetic_acc_eval1`, `valid_arithmetic_acc_eval2`, etc. and
`valid_arithmetic_acc_error1`, `valid_arithmetic_acc_error2`, etc. For eval metrics, the number of perfect predictions are added to the indicator.

This allows the model to report "weak successes" such as close predictions, or correct predictions up to the sign, and specific error cases. Note that whereas eval metrics can be used with beam search (the best value over the beam is then reported), error metrics cannot.

Int2Int can also report accuracy on subgroups of the test set, via the function
`src/envs/arithmetic.py/code_class()`. It computes, for each test example, an integer representing its class. During evaluation, an accuracy per class is reported. The class could be anything: a measure of problem size (e.g. the size of a system, the dimension of a square matrix), a class of desired output (e.g. in modular arithmetic, the correct solution to the problem).

4.3 Generators

The `generate()` function, in `src/envs/generator.py` handles the creation of data examples, for training, evaluation, or data generation(i.e. when setting `--export_data true`). When working on a new problem, it will have to be rewritten, unless one learns and evaluate from an external data file. `generate()` uses an external random number generator – not the default Python or NumPy generator. This prevents different instances of the generator (when the model runs on several GPU, or uses several worker CPU) to generate the exact same data. The parameter `type` can be set to the dataset you are generating, to `train` for the training set, and `valid` for the evaluation set. This allows test examples to be drawn from a different data distribution than training examples.

`generate()` returns two mathematical objects, the problem and solution, to be fed into the tokenizers. The typical input is a list (or an array) of integers, the output could be an array, a list or a single integer. If generation fails, the function can return `None`, this (or any exception) will cause `generate()` to be called again.

5 Visualizing results

A Jupyter Notebook is provided in `tools/ReadXP.ipynb`. It provides basic functions to read output logs, build result tables and learning curves. You need to define the `dump_path` (as in Int2Int), the `exp_name` of the experiments you are investigating, and the `sterr_path` where the `stderr` output of your runs are being saved (assumed to be of the form `stderr_path/exp_name/*/`. The notebook will then collect all `train.log` files in these directories, and print tables and learning curves.

Acknowledgements

The original code of Int2Int was written by Guillaume Lample. Successive versions of this code base were released as code examples for different papers [5, 4, 9, 12]. The Harvard CMSA program on Mathematics and Machine Learning was the main inspiration for developing this program, several participants, notably, Edgar Costa, Barinder Banwait, Kyu-Hwan Lee, and Jim Halverson helped test an initial version. The following paper used such initial versions [2]. The elliptic curve dataset provided as a demo is based on data extracted from the LMFDB database by Barinder Banwait.

References

[1] Alberto Alfarano, François Charton, and Amaury Hayat. Global lyapunov functions: a long-standing open problem in mathematics, with symbolic transformers. *arXiv preprint arXiv:2410.08304*, 2024.

[2] Angelica Babei, François Charton, Edgar Costa, Xiaoyu Huang, Kyu-Hwan Lee, David Lowry-Duda, Ashvni Narayanan, and Alexey Pozdnyakov. Learning euler factors of elliptic curves. *arXiv preprint, arXiv:2502.10357*, 2025.

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

[4] François Charton. Linear algebra with transformers. *arXiv preprint arXiv:2112.01898*, 2021.

[5] François Charton. Learning the greatest common divisor: explaining transformer predictions. *arXiv preprint arXiv:2308.15594*, 2023.

[6] François Charton, Jordan S. Ellenberg, Adam Zsolt Wagner, and Geordie Williamson. Patternboost: Constructions in mathematics with a little help from ai. *arXiv preprint arXiv:2411.00566*, 2024.

[7] 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.

[8] Róbert Csordás, Kazuki Irie, and Jürgen Schmidhuber. The neural data router: Adaptive control flow in transformers improves systematic generalization. *arXiv preprint arXiv:2110.07732*, 2021.

[9] Stéphane d'Acoli, Pierre-Alexandre Kamienny, Guillaume Lample, and François Charton. Deep symbolic regression for recurrent sequences. *arXiv preprint arXiv:2201.04600*, 2022.

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

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *17th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019.

[12] François Charton et al. Emergent properties with repeated examples. *arXiv prepring arXiv:2410.07041*, 2024.

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

[14] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint, arXiv:1606.08415*, 2016.

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

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

[17] Nayoung Lee, Kartik Sreenivasan, Jason D. Lee, Kangwook Lee, and Dimitris Papailiopoulos. Teaching arithmetic to small transformers. *arXiv preprint arXiv:2307.03381*, 2023.

[18] The LMFDB Collaboration. The L-functions and modular forms database. `https://www.lmfdb.org`, 2024. [Online; accessed 29 December 2024].

[19] Rodrigo Nogueira, Zhiying Jiang, and Jimmy Lin. Investigating the limitations of transformers with simple arithmetic tasks. *arXiv preprint arXiv:2102.13019*, 2021.

[20] 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.

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[22] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2020.