Reinforcement learning for proofs

Carlos Simpson

Barcelona Mathematics and Machine Learning

online colloquium series

March 12th, 2026

Introduction

Recently, the question of computer formalization of mathematics has come into focus for the wider mathematical community. We may recall that the subject area has been developing strongly for many decades. Advances in artificial intelligence open up the prospect of making proof formalization available for mathematical reasoning in high level research contexts as well as many other applications.

We will look at the general problem of reinforcement learning for automated proofs. This is an important aspect of our Malinca ERC Synergy project. We'll describe some general aspects, and then specialize to the more specific examples that I have looked at so far. These are precise combinatorial questions in the classification of algebraic structures in the examples of semigroups, and triangulations.

Introduction

These projects have benefited from discussions with a whole bunch of people, including but not limited to the members of the Malinca project.

https://malinca.gitlabpages.inria.fr/malinca.gitlab.io/team/

I would like to thank all of the people whose comments have contributed to this work.

Malinca

Our project called *Mathematicae Lingua Franca* means the "lingua franca" of mathematics.

The *lingua franca* was the simplified language of the mediterranean basin allowing people to communicate simply and precisely to facilitate economic and commercial exchanges.

For us, the idea is that we are now faced more and more with mathematical communication questions involving different entities, notably human mathematicians and computers that might be able to assist our work, as long as the communication can be made with simplicity and precision.

Computer proof verification

One of the main aspects is the formalization and verification of proofs by computer. This is a classical topic, with some of the main programs today including ROCQ (the new name for COQ), LEAN, ISABELLE, MIZAR, METAMATH, AGDA, and many others.

Automated proof search using programs such as VAMPIRE is an important component of the theory. We note that in some cases, automated proof search programs are used for “Sledgehammer” tactics such as in ISABELLE.

Linguistics

In general, an exact approach to linguistics runs up against all the small variations that make language poetic and literary. We note, however, that the subset of natural language used to express mathematical writing with a usual level of rigour might well be more closely adapted to analysis by exact methods.

The “Malinca” intermediate language is thought of as a bridge between written mathematical texts and computer-formalized mathematics. The situation is nonetheless complex and we should really think of a range of intermediate levels progressing between the two sides.

Type theory

Nowadays, logic is most often expressed in terms of Bertrand Russell's *type theory*. It was originally introduced as a method of resolving the classical paradoxes.

A computer proof verification method will most often be based on some version of type theory, starting with de Bruijn's AUTOMATH in the 1970's.

It is interesting to note that computational linguistics can also be based on type theory, so we may hope to have a common theoretical framework for both sides of the Malinca project.

Type theory

A certain study of the logical axiomatic foundations is required, and the choice of foundational structure can impact the efficiency of programming.

These and related topics are also themes of study in our project, including:

- Homotopy Type Theory (HoTT)
- the geometrisation of reasoning
- game semantics for proofs
- translations between different representations of mathematics
- interactivity and bidirectionality of the formalization process

and many other topics that are farther from the themes of the present talk.

Searching for proofs

In a natural-language mathematical text, the author does not treat all the small details that would be necessary in order to formulate a document acceptable to a computer proof verification system.

This is one of the main obstacles to the formalization of wide segments of mathematics. It takes a lot of work to produce a document written in the formal proof language.

It will therefore be necessary to have a layer aiming to look for and fill in all of the small steps. Utilisation of AI or “machine learning” obviously comes to mind here, but there might also be other proof search algorithms that could be helpful too.

We note that such questions are the subject of a lot of current research.

Searching for proofs

Let us point out that these objectives are somewhat different from the current lines of research involving training LLM's to "solve math problems". For our purposes, we are not really interested in asking the computer to find any clever ideas. That would be the work of the human author of a mathematical text.

The goal is rather to assist the mathematician by dealing with the lower-level work needed to go from the level of a standard well-written mathematical text, towards a formal proof document.

One furthermore requests that the process should be transparent and reproducible.

Searching for proofs

This gets us to the topic of today's talk: we would like to investigate the utilisation of machine learning, and in particular reinforcement learning, to find proofs in a controlled environment.

Unfortunately, I do not have anything specific to report in the domain of proofs that would fit into regular mathematics texts. This is current work in progress that is not at a very extensive state of advancement.

We'll first discuss the general framework of the question and the potential structures that intervene. Then we'll go on to look at the examples that I have looked at, involving proofs for the classification of combinatorial objects.

Set theory

My own taste for the direction of axiomatic mathematical development runs in the “untyped” direction. Classical set theory following the Zermelo-Fraenkel axioms has provided the main foundational basis for pure mathematics.

This may be implemented perfectly well in a proof assistant, with a type E for sets, having a basic relation

```
inc : E -> E -> Prop
```

of elementhood. One defines for example

```
sub a b := forall x:E, inc x a -> inc x b
```

Proof structure in general

Typically, a proof is composed of numerous steps. The “time” of a proof may be viewed as a tree, with each instant corresponding to a node in the tree.

Each node corresponds to a context, like a context window in a formalized proof system. The root is the global context under discussion, that is to say, the thing we are trying to prove.

The tree is built up successively in steps, with the leaves divided into three groups: “done”, meaning that some conclusion has been obtained; “impossible” meaning that this branch has been found to be impossible, and “active” meaning that further work is needed.

Proof structure in general

In the logical case, “done” and “impossible” might be the same: that the conclusion False has been found. For a classification proof, “done” means that some objects are added to the classification whereas “impossible” means that no objects have been found.

At each “active” node, some proof step is needed requiring an *action*. This will in turn generate one or more new nodes (leaves) below that node, that could be active, done or impossible.

The new nodes that are generated come from the subgoals of a given step. For example, if we cut using the excluded middle on a proposition P, then we obtain two subgoals in which the context is augmented, in one case by P, and in the other case by (not P).

Proof structure in general

The question for an agent doing the proof (human or machine) is which action to take at each active node. For example, what would be a good choice of proposition P on which to apply the excluded middle.

This has an impact on the size of the proof: the total number of nodes needed until all the leaves become done or impossible.

This now indicates the *reinforcement learning problem*: for a given context associated to an active node, what is the best action to choose in order to complete the proof as quickly as possible.

A beginning scenario: combinatorial structure classification

The general problem of proof search for classical mathematical statements is complex. Much work has been done for example with the *Sledgehammer* tactic for ISABELLE, and analogous directions for MIZAR, LEAN, COQ, and first-order provers such as VAMPIRE. My student Boris Shminke wrote his thesis on this (Nice 2023).

The general setting remains difficult. We therefore turn our attention to a more restrained setting: the classification of finite combinatorial structures.

Given a type of structure, the question of classifying or enumerating the instances may be seen as a special type of proof, where the contexts associated to nodes in the proof tree correspond to collections of constraints on the product operations under consideration.

A beginning scenario: combinatorial structure classification

This setting has numerous benefits from the viewpoint of applications of machine learning:

- The array of proof steps to choose from at each stage is very homogeneous
- It is not too difficult to program the results of each proof step choice
- The classification generates large quantities of data that can then be used to train the machine.

The first topic will be the simplest case of semigroups. These constitute one of the simplest cases in universal algebra of equational theories where the signature has a single binary operation and the basic equation is associativity. It provides the occasion to apply techniques of reinforcement learning.

Semigroup classification

To introduce the problem, recall that a *semigroup* is a set A together with a binary operation $\cdot$ subject to the axiom of associativity:

$$\forall x, y, z \in A, \ \ (x \cdot y) \cdot z = x \cdot (y \cdot z).$$

We are interested in classifying finite semigroups with a given size $|A| = n$, usually subject to various other constraints as we'll see below.

The counting question has been considered for semigroups of small sizes, with the difficulty growing exponentially.

Various articles from around 2010 by Andreas Distler, Chris Jefferson, Tom Kelsey, Lars Kotthoff, and James Mitchell gave the solution for semigroups of order ≤ 10, and in particular gave precise formulas for the 3-nilpotent case.

Graded 4-nilpotent semigroups

After the 3-nilpotent case, the largest piece of the classification question is taken up by the 4-nilpotent semigroups. These are ones where any word of length 4 has product the zero element. We may create a 4-step filtration whose last step consists of only the zero element.

Consider the *graded case*: with sets A and B, let $I := \{0, 1\}$, and we'll look for semigroups of the form

$$M = A \sqcup B \sqcup I$$

where the multiplication consists of operations

$$\mu : A \times A \to B^0 := B \cup \{0\}$$

$$\phi : A \times B \to I, \quad \psi : B \times A \to I.$$

Graded 4-nilpotent semigroups

Denote $a = |A|$ and $b = |B|$.

Our semigroup M will be called *graded* with ranks $(a, b, 1)$, of overall size $n = a + b + 2$.

This notion of “graded object” should best be understood using the notion of algebra in the tensor category of pointed sets, that we should think of as $\mathbb{F}_1$-algebras.

The split structure described above is what is obtained by taking the “associated-graded object” for the filtration.

Graded 4-nilpotent semigroups

The table for the operation ϕ is a tensor of size $a \times b$ filled with $0, 1$'s.

The product of symmetric groups $S_a \times S_b$ acts on the set of such tables, and one way to remove a lot of symmetry is to fix a set of representatives for equivalence classes of tables under this action.

This still leaves us with many tables having nontrivial symmetry groups, typically $\mathbb{Z}/2$ or $\mathbb{Z}/2 \times \mathbb{Z}/2$, and we can furthermore try to fill in some elements of the other table ψ in such a way as to break these symmetries.

This is roughly speaking how we choose the set of initial conditions.

Tensor representation

It is interesting to program the algebraic structures in Pytorch using tensorial methods.

A position of the proof (i.e. a "context" within our general terminology) may be called a *template*. It is a 3-dimensional boolean tensor of size $n \times n \times n$ denoted prod with the following meaning:

$$\mathrm{prod}[x, y, p] == 0 \quad \text{means } x \cdot y \neq p$$

whereas

$$\mathrm{prod}[x, y, p] == 1 \quad \text{means } x \cdot y \text{ might equal } p.$$

Initial position

The initial position of the proof, where anything is possible, is the tensor of ones

```
prod = torch.ones((1,n,n,n),dtype = torch.bool)
```

then, as the proof goes on, we refine this by adding various 0's indicating that some products are not there.
(Here the 0'th dimension is always a *batch* variable: the structure of Pytorch tensor operations allows us to treat ~ 1000 proof positions at once).

In practice a refinement of the initial position is usually chosen before we start, corresponding to the classification of semigroups satisfying some additional properties (eg nilpotent, graded, bands, etc.).

Filter: impossible

The proof position is *impossible* if for any one value of (x, y) the column $p \mapsto \mathrm{prod}[:, x, y, p]$ is empty, that is to say

```
possible = ((prod.any(3)).all(2)).all(1)
impossible = ~possible
```

These yield boolean vectors whose length is the batchsize (the 0'th dimension of the tensors), telling us which locations correspond to a possible or impossible template.

Filter: done

The proof position is *done* if, for each (x, y) there is exactly one value of p such that $\mathrm{prod}[:, x, y, p] == 1$ with the others equal zero. We measure this as follows:

```
stats = prod.to(torch.int64).sum(3)
done = (stats == 1).all(2).all(1)
```

In this case, the choice of the valid p provides us with the multiplication table

$$x \cdot y = p \Leftrightarrow \mathrm{prod}[:, x, y, p] == 1$$

Filter: done

In order to declare the position as ‘done’ we also ask that the associativity condition be verified.

A relaxed associativity condition may also be considered, where we just require that the triple products are uniquely defined and the same for the left and right parenthetization.

The ‘done’ positions then need to be iso-filtered as we have discussed above. This can be done on-the-fly so only the iso-filtered list needs to be conserved in memory.

Processing

The associativity axiom implies that for a given template, we can sometimes add additional zeros. This can be programmed directly using Pytorch boolean tensor operations. This processing step is the costly part of the computations.

Processing

To give an idea, here is the code to calculate the tensor of x, y, z, p such that $(x \cdot y) \cdot z$ can be p. The coordinates are ordered i, x, y, a, z, p where i is the batch variable and a is the new variable representing possible products $x \cdot y$.

```
prod_xya = prod.view(length,n,n,n,1,1).expand(length,n,n,n,n,n)
prod_azp = prod.view(length,1,1,n,n,n).expand(length,n,n,n,n,n)
left_triple_product = (prod_xya & prod_azp).any(3)
```

We similarly define the right triple product, then the triple product possibilities are the & of these two.
From there, we can go backwards to obtain deductions yielding new 0's in `prod`.

Proof positions

We now look at the structure of a classification proof. A proof step consists of a *cut* obtained by choosing some (x_i, y_i).

A given position ρ (corresponding to a tensor `prod`) then generates a sequence of positions $\rho_1, \ldots, \rho_k$.

The number k of these positions is the number of 1's in the column $\rho(x_i, y_i, -)$.

Each position ρ_i corresponds to declaring one value of $\rho(x_i, y_i, p)$ equal to 1 and the other ones equal to zero.

Proof tree

A partially (or fully) completed proof corresponds to a *proof tree* with each node decorated with a position or template ρ.

The root of the tree is the initial position.

At each node, a cut is chosen, and the nodes below it are the $\rho_1, \ldots, \rho_k$ generated by the cut.

Development of the tree stops below any node that is labeled 'done' or 'impossible', and the classification answer is given by the collection of 'done' nodes in the finished proof tree.

Proof tree

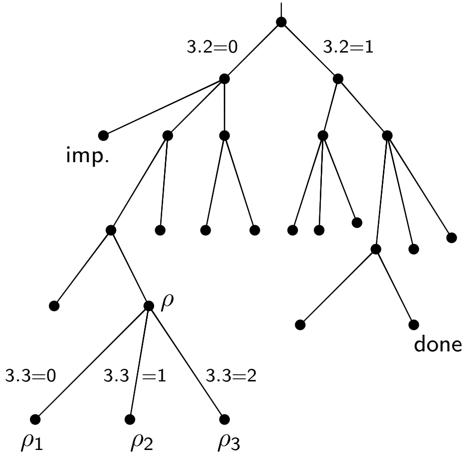


Figure: A small piece of a proof tree

Choice of cuts

Note that to choose a cut we should choose (x_i, y_i) such that there are at least two different nonzero elements in the column (we say that ρ is *active* and that such a location is *available*).

If that isn't possible, then the position is either done or impossible. The proof is finished when there are no further active nodes.

The process shares a commonality with the notion of "abelian sandpile" in that the order of choosing the active nodes doesn't make a difference.

Choice of cuts

However, the choice of which cut (x_i, y_i) to choose for an active node ρ_i, can make a big difference in the length of the proof.

We measure the size of the proof by counting the active nodes in the proof tree.

The leaves are the ‘done’ or ‘impossible’ nodes, an alternative way of counting would be to count those too, or we could include a weight for them.

The reinforcement learning task

This situation is a microcosm for the situation in a more general, and much more complicated proof, where a proof tactic or some kind of reasoning method needs to be chosen at each step.

Our “learning task” is to try to figure out the best choice of cut (x_i, y_i) at a given node ρ_i in the proof tree. The objective is to minimize the total number of active nodes.

This is very similar to a strategy game, so the success of *AlphaGo* and *AlphaZero* motivates us to try to apply reinforcement learning.

The reinforcement learning task

We observe that this minimization question is local: the part of the proof tree under a given node ρ_i depends only on the cuts chosen at ρ_i and at the nodes under it. The global minimization is obtained by choosing a cut at ρ_i that is going to minimize the number of active nodes below ρ_i.

This has an RL aspect: that number is going to depend on our future strategy of cuts chosen at the nodes below.

The reinforcement learning task—a first version

A first version of the learning strategy is therefore:

- Learn the function

  $$\mathrm{Val}(\rho) := \text{ number of nodes below } \rho \text{ for our current strategy}$$

- At ρ choose the cut (x, y) that minimizes

  $$\mathrm{Pol}(\rho; x, y) := \sum_{j=1}^{k} \mathrm{Val}(\rho_j)$$

  where $\rho_1, \ldots, \rho_k$ are the positions generated by the cut at (x, y).

Value and policy networks

It turns out to be a good idea to train **two** networks:

- `network_value` to predict $\mathrm{Val}(\rho)$ (the input is an $n \times n \times n$ boolean tensor and the output is a scalar float, actually the log of the value)
- `network_policy` to predict $\mathrm{Pol}(\rho; x, y)$ (the input is an $n \times n \times n$ boolean tensor and the output is an $n \times n$ float tensor, again the log).

The strategy is determined by the `network_policy` network: at a position ρ the chosen cut will be the value of available (x, y) that minimizes the answer.

Value and policy networks

The first basic reason for the usefulness of having two networks is one of speed: during the proof we need to calculate the array of $\mathrm{Pol}(\rho; x, y)$ for all available values of (x, y) and then choose (x, y) that minimizes it.

The original definition of $\mathrm{Pol}(\rho; x, y)$ involves processing the results ρ_j of the cuts (if we assume that $\mathrm{Val}(\rho_j)$ is measured on processed templates). This is costly, and doing it for all the values of (x, y) would take a lot of time.

For reasons that remain mysterious, it seems to be the case that this division into two networks also improves the reinforcement learning.

Collecting data

Once this framework is set, the basic issue is how to obtain data on which to train the networks. This seems to be a fairly subtle question.

A first idea would be to do a proof (say, with a random choice of cuts), but prune some of the active nodes at each stage, in order to generate proof positions.

Collecting data

The difficulty is that the proof tree is generally speaking highly imbalanced. If we imagine that it is binary, then after the first 10 levels there will be $2^{10} = 1024$ leaves. Of those, only a very small number contribute significantly to the proof tree after 20 levels, so that at stage 20 there might be 10000 nodes that came from, say, 10 out of the 1024. And so on.

The depth can go up to 50 or 60 in the cases that can be calculated.

A naive pruning approach doesn't yield adequate data.

Collecting data

Of course, for small examples, one can just use the full proof itself, but that doesn't really represent the full amount of difficulty of the question.

The general problem of how to generate adequate samples is, I think, one of the main problems in the subject. It will get more difficult in the transition towards more general types of proof situations.

Training data

Once we have a set of sample positions, then we use an iterative procedure to train the value and policy networks. We note that

$$\mathrm{Val}(\rho) = 1 + \sum_{j=1}^{k} \mathrm{Val}(\rho_j)$$

where the positions $\rho_1, \ldots, \rho_k$ are generated by the cut (x, y) that is the minimum for $\mathrm{Pol}(\rho; x, y)$.

Training data

We can just program this, using `network_policy` to determine the choice of x, y and then (under `torch.no_grad()`) use `network_value` to determine the pieces of the sum. Remember that our networks train for the log values so we exponentiate, take the sum and take the log.

Surprisingly enough, that seems to work.

We do that calculation separately for each minibatch.

Training data

Of course, for training `network_policy` we just need to calculate the policy values for sample values of (x, y). A minibatch consists of a collection of positions ρ_i and a randomly chosen available collection of cuts (x_i, y_i).

In particular, we don't try to train all of the array elements at the same time (that doesn't appear to be a good idea).

Network architecture

For network architecture: the first tries were to use a basic deep network with several fully connected layers, and a skip connections obtained by cat'ing them together at the end before the last layer. That worked reasonably well.

This forms the basis for further experimentation on what kind of network architecture best allows the computer to learn to reason with this kind of algebraic structure encoded in a multiplexed tensor. More complicated architectures were investigated in the later project on triangulations.

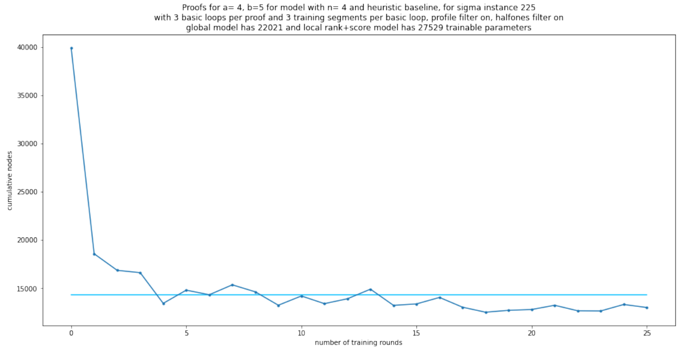
Variable minibatches

The training process is standard too. I found it to be useful to mix the sizes of minibatches, and the number of gradient descent steps for each minibatch.

The idea is that in each training stage we over-train on a small minibatch or two, then train more lightly on some wider minibatches. Here again it could provide a source of experimentation.

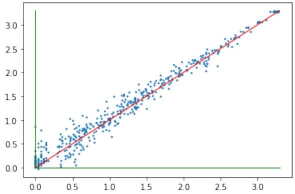
As usual, my only knowledge about this comes from experimenting (after initially using the basic setups that one finds explained in many places on the web).

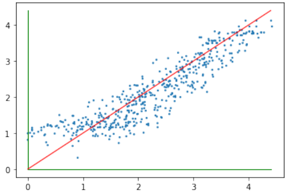
Some results



For a graded 4-nilpotent case of type $(4, 5, 1)$ the machine is able to decrease the number of active nodes, and do better than the heuristic strategy.

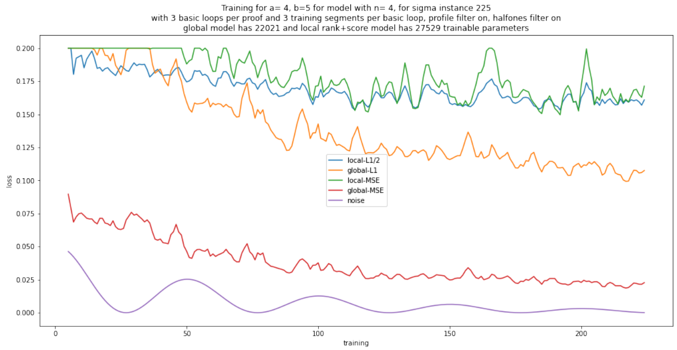
Some results





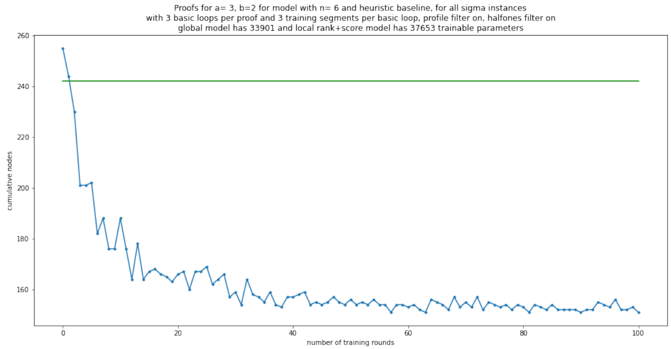
These picture the performance of the Value and Policy networks after learning.

Some results



This pictures the evolution of the loss function in the course of training.

Some results



In a smaller case of type $(3, 2, 1)$ (a graded 4-nilpotent semigroup of size 7) with all initial conditions put together, the machine gets down and bounces on the theoretical minimum number of nodes in the proof.

Other results

Consider the example of a nilpotent semigroup of size n without the grading. We fix the order of elements corresponding to a filtration that is strictly decreased by the multiplication, that is to say if $x_i \cdot x_j = x_k$ then $k < \min(i, j)$ unless $k = 0$.

The initial template for $n = 4$ looks like

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 10 & 10 \\ 0 & 0 & 10 & 210 \end{bmatrix}$$

where we put at x, y the numerals of locations p such that $prod[x, y, p] = 1$, so for example $3 \cdot 3$ can be $0, 1$ or 2.

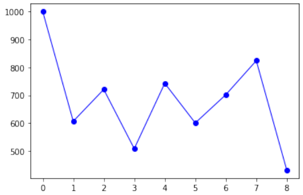
Other results

In this case it is best to use the alternative characterization of ‘done’ to determine stopping: whenever both multiplications $(x \cdot y) \cdot z$ and $x \cdot (y \cdot z)$ are uniquely determined and equal to the same element, for each x, y, z, we stop. The number of multiplication tables is then the product of the `stats`. For $n = 6$ the total number is 89251.

Other results

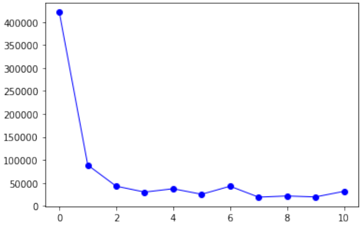
We do a first proof using random choice of cuts, use that to generate some samples, then train on those samples (resulting in the generation of new samples), and prove using the model. Iterate the last two steps.

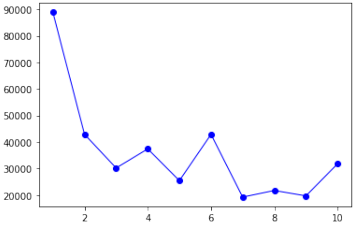
For the nilpotent case of size 6 this gave the following numbers of active nodes:



Other results

For nilpotent of size 7:





Other results

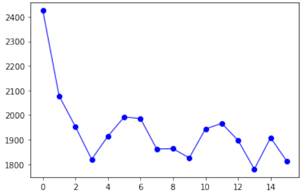
One may also implement the case of a 'band'. This is a semigroup in which every element is idempotent $x \cdot x = x$. In that case, we add an additional filter imposing that a certain pre-order (defined by inclusion of one element in the left ideal generated by another one) should be compatible with the numerical order.

In general, removal of symmetries is a main problem as we go towards higher sizes of semigroups.

Other results

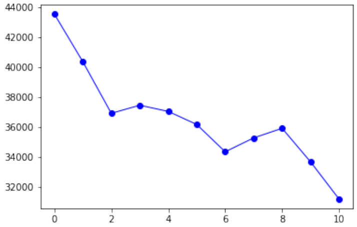
We do the same cycle: start with a random choice of cuts then learn and prove using the updated model.

For the case of bands of size 5 the results were as follows:



Some results

For bands of size 6:



A global approach

These results show that a machine can learn to do proofs in an efficient manner. However, they are unsatisfactory from the viewpoint of speeding up the process overall. Indeed, the amount of training necessary to obtain a significant improvement in the size of the proof, takes much longer than the computation time that is saved.

In parallel, for certain cases, we make a somewhat curious observation: just choosing the cuts in a standard lexicographic ordering on the (x, y) (that is to say, choose the largest or smallest pair that is available), yields a pretty good result. We'll see that below.

A global approach

This phenomenon comes, undoubtedly, from our choice of initial conditions. In order to remove symmetry, we try to factor out the choice of ordering on the elements as much as possible. One way of doing this is to impose some kind of lexicographic ordering on some part of the multiplication table; another way is to choose a part of the table and do a sieve operation to divide by the permutations of subsets of elements.

It is therefore perhaps not surprising that the standard ordering might have some special properties.

A global approach

Still, it remains a little surprising that we can fix a global ordering on the set of (x, y), use that to choose each cut, and obtain a reasonable result.

I was therefore curious as to whether a learning process could replicate or even improve this.

This led to a process that I'll now describe. It is entirely heuristic-based, involving a guess (informed by experimentation) of a quantity to look at.

A global approach

The setup is that we are going to establish a global ordering on the set of cut locations (this is actually a subset of the set of all (x, y) in view of the initial conditions that already restrict the available locations, for example in our $n = 4$ nilpotent template the only available locations are the foursome in the bottom right).

At any stage of the proof, we will have already used up a first collection of cuts, and the "next in line" is the one to be chosen. Notice that this could be farther along than just the next one after our previously chosen collection, since some locations might become unavailable.

(Available means `stats > 1`)

A global approach

Let σ denote a collection of cut locations. We now establish a quantity that is to be learned. The quantity depends on σ, and is supposed to be an average over the templates in our sample pool. For each template, we impose the cuts determined by all the locations in σ, and count the available locations.

Then, we process the resulting template, and count the available locations (there will be less). The gain (number of new unavailable locations) is our quantity ascribed to σ and the template.

To generate samples, we take an average over a few different templates from the sample pool.

Sampling

A word about the sample pool of templates to be used here. Rather than sampling from the active nodes, we sample from done and impossible nodes. When imposing the cuts at σ locations, it means that for each $(x, y) \in \sigma$ we impose the value that is given by our template, or choose a value if that is empty (impossible template).

This process is necessary: if we just chose a random value for each $(x, y) \in \sigma$ that would very often lead to an impossible processed version. We need to get a collection of sample templates that is as far along as possible in the proof tree.

I don't see any way of guaranteeing anything about the distribution of our samples, which is a fundamental problem.

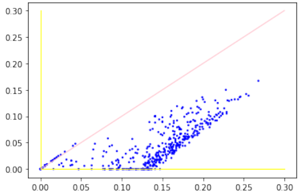
An heuristic quantity to maximize

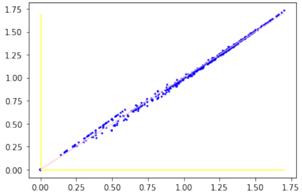
We define in such a way some quantity depending on σ, designed to reflect the "power" of this collection of cuts. The basic idea is to take into account somehow the fact that various choices of cuts working together are going to reinforce each other.

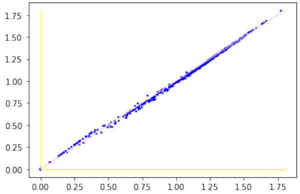
The neural network is trained to give a predicted function $A(\sigma)$.

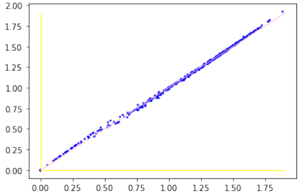
Here are some pictures of the accuracy of the network improving during training.

An heuristic quantity to maximize









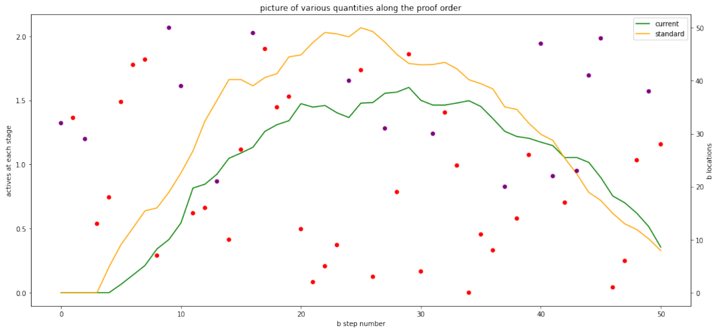
An heuristic quantity to maximize

Now, suppose we are given an ordering o of the set of node locations. This yields a family of subsets starting with $\sigma_0 = \emptyset$ and adding each location along the way. Set

$$\mathbf{A}^{\mathrm{glob}}(o) := \sum_{i=0}^{b} A(\sigma_i).$$

(One can also experiment with a weighted sum, but the simple unweighted sum seems to work just fine.)

An heuristic quantity to maximize



The quantity is the integral under the curve. The orange curve is the standard ordering, the green curve a random ordering.

Stochastic discrete descent

We would like to look for the ordering o that maximizes $\mathbf{A}^{\mathrm{glob}}(o)$. For that we use a 'genetic' process that could be called *stochastic discrete descent*. Namely, we choose a subset of say 5 locations and look at all permutations of these in the ordering; take the new ordering that maximizes $\mathbf{A}^{\mathrm{glob}}(o')$ among these locally permuted orders.

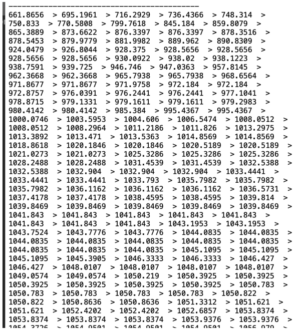
Then repeat the process with random subsets. After a while it stabilizes. That is going to be the order we use for the proof.

Stochastic discrete descent

The fact that we can obtain a reasonable maximization by using small permutations, would seem probably to be related to a property of our function $A(\sigma)$, something like that it mostly depends on combinations of small finite numbers of locations.

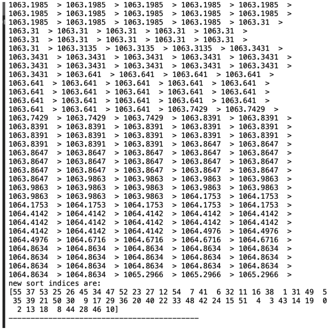
Other versions of a genetic algorithm may be envisioned, such as testing a wide collection of transpositions at each step (apparently there are many such techniques.)

Stochastic discrete descent



```
------------------------------------------
661.8656  > 695.1961  > 716.2929  > 736.4366  > 748.314  >
750.833  > 770.5808  > 799.7618  > 845.184  > 859.8079  >
865.3889  > 873.6622  > 876.3397  > 876.3397  > 878.3516  >
878.5453  > 879.9779  > 881.9982  > 889.962  > 890.8309  >
924.0479  > 926.8044  > 928.375  > 928.5656  > 928.5656  >
928.5656  > 928.5656  > 930.0922  > 938.02  > 938.1223  >
938.7591  > 939.725  > 946.746  > 947.0363  > 957.8145  >
962.3668  > 962.3668  > 965.7938  > 965.7938  > 968.6564  >
971.8677  > 971.8677  > 971.9758  > 972.184  > 972.184  >
972.8757  > 976.0391  > 976.2441  > 976.2441  > 977.1041  >
978.8715  > 979.1331  > 979.1611  > 979.1611  > 979.2983  >
980.4142  > 980.4142  > 985.384  > 995.4367  > 995.4367  >
1000.0746  > 1003.5953  > 1004.606  > 1006.5474  > 1008.0512  >
1008.0512  > 1008.2964  > 1011.2186  > 1011.826  > 1013.2975  >
1013.3892  > 1013.471  > 1013.5363  > 1014.8569  > 1014.8569  >
1018.8618  > 1020.1846  > 1020.1846  > 1020.5189  > 1020.5189  >
1021.0273  > 1021.0273  > 1025.3286  > 1025.3286  > 1025.3286  >
1028.2488  > 1028.2488  > 1031.4539  > 1031.4539  > 1032.5388  >
1032.5388  > 1032.904  > 1032.904  > 1032.904  > 1033.4441  >
1033.4441  > 1033.4441  > 1033.793  > 1035.7982  > 1035.7982  >
1035.7982  > 1036.1162  > 1036.1162  > 1036.1162  > 1036.5731  >
1037.4178  > 1037.4178  > 1038.4595  > 1038.4595  > 1039.814  >
1039.8469  > 1039.8469  > 1039.8469  > 1039.8469  > 1039.8469  >
1041.843  > 1041.843  > 1041.843  > 1041.843  > 1041.843  >
1041.843  > 1041.843  > 1041.843  > 1043.1953  > 1043.1953  >
1043.7524  > 1043.7776  > 1043.7776  > 1044.0835  > 1044.0835  >
1044.0835  > 1044.0835  > 1044.0835  > 1044.0835  > 1044.0835  >
1044.0835  > 1044.0835  > 1044.0835  > 1045.1095  > 1045.1095  >
1045.1095  > 1045.3905  > 1046.3333  > 1046.3333  > 1046.427  >
1046.427  > 1048.0107  > 1048.0107  > 1048.0107  > 1048.0107  >
1049.0574  > 1049.0574  > 1050.219  > 1050.3925  > 1050.3925  >
1050.3925  > 1050.3925  > 1050.3925  > 1050.3925  > 1050.783  >
1050.783  > 1050.783  > 1050.783  > 1050.783  > 1050.822  >
1050.822  > 1050.8636  > 1050.8636  > 1051.3312  > 1051.621  >
1051.621  > 1052.4202  > 1052.4202  > 1052.6857  > 1053.8374  >
1053.8374  > 1053.8374  > 1053.8374  > 1053.9376  > 1053.9376  >
1054.3726  > 1054.9501  > 1054.9501  > 1054.9501  > 1056.979  >
```

Starting the permutations



```
1063.1985  > 1063.1985  > 1063.1985  > 1063.1985  > 1063.1985  >
1063.1985  > 1063.1985  > 1063.1985  > 1063.1985  > 1063.1985  >
1063.1985  > 1063.1985  > 1063.1985  > 1063.1985  > 1063.31  >
1063.31  > 1063.31  > 1063.31  > 1063.31  > 1063.31  >
1063.31  > 1063.31  > 1063.31  > 1063.31  > 1063.31  >
1063.31  > 1063.3135  > 1063.3135  > 1063.3135  > 1063.3431  >
1063.3431  > 1063.3431  > 1063.3431  > 1063.3431  > 1063.3431  >
1063.3431  > 1063.3431  > 1063.3431  > 1063.3431  > 1063.3431  >
1063.3431  > 1063.641  > 1063.641  > 1063.641  > 1063.641  >
1063.641  > 1063.641  > 1063.641  > 1063.641  > 1063.641  >
1063.641  > 1063.641  > 1063.641  > 1063.641  > 1063.641  >
1063.641  > 1063.641  > 1063.641  > 1063.641  > 1063.641  >
1063.641  > 1063.641  > 1063.641  > 1063.7429  > 1063.7429  >
1063.7429  > 1063.7429  > 1063.7429  > 1063.8391  > 1063.8391  >
1063.8391  > 1063.8391  > 1063.8391  > 1063.8391  > 1063.8391  >
1063.8391  > 1063.8391  > 1063.8391  > 1063.8391  > 1063.8391  >
1063.8391  > 1063.8391  > 1063.8391  > 1063.8647  > 1063.8647  >
1063.8647  > 1063.8647  > 1063.8647  > 1063.8647  > 1063.8647  >
1063.8647  > 1063.8647  > 1063.8647  > 1063.8647  > 1063.8647  >
1063.8647  > 1063.8647  > 1063.8647  > 1063.8647  > 1063.8647  >
1063.8647  > 1063.9863  > 1063.9863  > 1063.9863  > 1063.9863  >
1063.9863  > 1063.9863  > 1063.9863  > 1063.9863  > 1063.9863  >
1063.9863  > 1063.9863  > 1063.9863  > 1064.1753  > 1064.1753  >
1064.1753  > 1064.1753  > 1064.1753  > 1064.1753  > 1064.1753  >
1064.4142  > 1064.4142  > 1064.4142  > 1064.4142  > 1064.4142  >
1064.4142  > 1064.4142  > 1064.4142  > 1064.4142  > 1064.4142  >
1064.4142  > 1064.4142  > 1064.4142  > 1064.4976  > 1064.4976  >
1064.4976  > 1064.6716  > 1064.6716  > 1064.6716  > 1064.6716  >
1064.8634  > 1064.8634  > 1064.8634  > 1064.8634  > 1064.8634  >
1064.8634  > 1064.8634  > 1064.8634  > 1064.8634  > 1064.8634  >
1064.8634  > 1064.8634  > 1064.8634  > 1064.8634  > 1064.8634  >
1064.8634  > 1064.8634  > 1065.2966  > 1065.2966  > 1065.2966  >
new sort indices are:
[55 37 53 25 26 45 34 47 52 23 27 12 54  7 41  6 32 11 16 38  1 31 49  5
 35 39 21 50 30  9 17 29 36 20 40 22 33 48 42 24 15 51  4  3 43 14 19  0
  2 13 18  8 44 28 46 10]
------------------------------------------
```

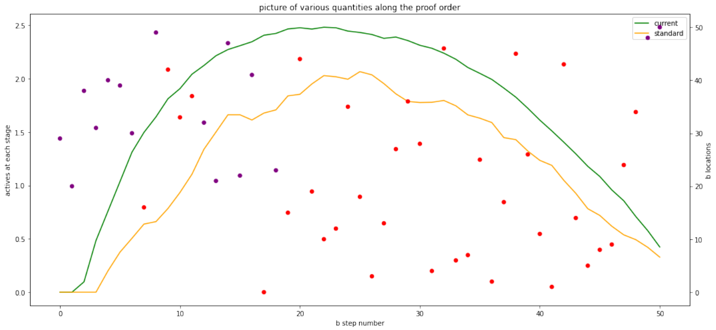
After 500 steps

Some results

We now describe the results. We'll consider graded 4-nilpotent semigroups of size 11 with graded ranks $(4, 5, 1)$. We further restrict to a subcase where the row is specified with a maximal number of 1's for some parts of the multiplication table, in the present case four 1's and one 0.

The preparation of initial conditions for this case takes a little over 40 seconds. This involves some proving work aimed at enforcing some lexicographic ordering on parts of the multiplication table. This part is the same for all the subsequent proof techniques.

The maximlzed quantity



Now the green curve represents our quantity for the maximized order.

Static results

Here are the results that don't use machine learning.

- A random choice of ordering yields a proof in 2101 steps, having 6.28 million active nodes, and taking around 14 minutes (on colab GPU).
- The standard order, going by decreasing lexicographic order on the cut locations (x, y), yields a proof in 979 steps, having 2.89 million active nodes, and taking 6 minutes 26 seconds.

Learning results

Our process for learning an optimal ordering involves the following steps:

- Collecting some samples, by doing several pruned versions of proofs.
- Learning the function $A(\sigma)$.
- Doing the genetic “stochastic discrete descent” algorithms to find a (near-to) optimal ordering on the locations.
- Then we do the proof to see how well this did.

Learning results

Collecting 10000 samples, then training for 10 rounds of training, and doing 400 permutation steps, yields the following timing and results:

Sampling: 24 seconds
Learning: 46 seconds
Descent: 23 seconds
Proof: in 1032 steps with 3.06 million active nodes and taking 445 seconds (7 minutes 25 seconds).
Full time: almost 9 minutes.

We may stress that this process is done in a way that doesn't know about the standard ordering. The full time including training and all, does significantly better than the random proof order, although not as well as the standard order.

Learning results

We re-did this experiment with a little more training, in order to see if we can do better than the standard order.

Collecting 20000 samples, then training for 15 rounds, and doing 500 permutation steps, yields the following timing and results:

Sampling: 48 seconds
Learning: 70 seconds
Descent: 29 seconds
Proof: in 877 steps with 2.61 million active nodes, taking 383 seconds (6 minutes 23 seconds).
Full time: 8 minutes 50 seconds.

Conclusion on the global approach

The full process is significantly better than the random order. Furthermore, the proof is shorter than that of the standard order (showing that the standard order, while very good, is not in fact optimal). However, if we include sampling and training time the standard order still wins.

It seems reasonable to expect that this kind of process would do better, on even bigger cases that are currently beyond my computer power.

It may be that the best choice could be a hybrid version where we choose a global order but updated it from time to time as we progress through the proof. In this type of setup, the order in which the active nodes are chosen would make a difference.

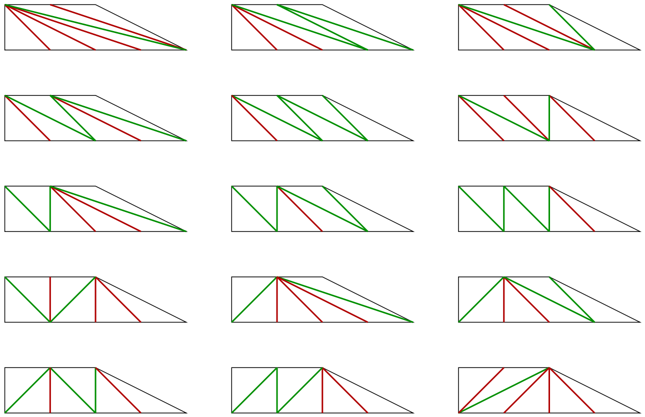
Classification of triangulations

We now turn to a different kind of combinatorial classification question.

When I explained the work on semigroups to Elizabeth Gasparim, she immediately saw that a similar technique should apply to the **classification of triangulations** of planar lattice polygons, a question she was interested in for Mathematical Physics reasons.

This has become a long-term joint project, still in progress, but with some outputs that can be reported.

Classification of triangulations



Lattice polygons and triangulations

Fix the standard lattice in the plane $\Lambda := \mathbb{Z}^2 \subset \mathbb{R}^2$. A *lattice polygon* is a polygon in $\mathbb{R}^2$ whose vertices are in Λ. It is therefore given by a sequence of *edges* that are pairs of points $((x, y), (u, v))$ such that $((u - x), (v - y))$ is *primitive* i.e. not a multiple of an integer vector.

In practice we fix a square inside Λ to form a grid, say of size 6×6, and consider edges contained in this grid. This allows to create look-up tables for the edges and related information.

Notably, we need to know which pairs of edges cross each other. A *polygon* is a closed cycle of edges that don't cross each other and yielding a positive area.

Lattice polygons and triangulations

We would like to look at *lattice triangulations* : these are triangulations of our polygon such that the vertices are on points of Λ and the triangles are of minimum area 0.5.

The basic question is to enumerate the triangulations of a given polygon.

The general motivation for Gasparim's question was the need for having an understanding of these triangulations for classification of certain resolutions of singularities in toric geometry needed in mathematical physics.

I do not currently understand precisely what quantities we would like to calculate, so on that side of things, as well as many others, this remains a work in progress.

Lattice polygons and triangulations

A basic point about these triangulations is that there are a lot of them. In the case of rectangles or more generally polygonal regions under a piecewise linear graph, a recurrence relation allows to make numerical computations (Kaibel-Ziegler 2003, Orevkov 2022).

For example, a 6×7 rectangle has 12169409954141988707186052 triangulations.

Lattice polygons and triangulations

If $f(m,n)$ is the number of triangulations of an $m \times n$ rectangle, the *capacity* is defined as

$$c(m,n) := \frac{\log_2 f(m,n)}{mn}$$

and this is going to be roughly 2, at least between 1 and 3. A lot is known about asymptotics of c.

Non-convex polygons are going to have less triangulations with respect to their areas, so we can expect very roughly to have something like 4^{Area} triangulations.

The Ray-Seidel algorithm

We are going to focus on running the algorithm that was proposed in (Ray-Seidel 2004). Their article concerned triangulations with vertices on an arbitrary set of interior points, in our case we use the points of Λ interior to the polygon.

They propose the following algorithm:

- Given a polygon P, choose an edge e of P.
- Look at all the primitive triangles t inside P containing e as an edge.
- For each one of these, generate a new polygon or pair of polygons P'.
- Apply recursively to the new polygons P' and combine the results.

The Ray-Seidel algorithm

This “divide and conquer” process is an example of *dynamic programming*, and one may quite generally imagine the application of techniques such as we discuss here, to a wide array of such problems. (The semigroup question considered above also falls into this category.) We restrict to our instance.

Some improvements have been proposed (Alvarez-Seidel 2013, Marx-Miltzow 2016, . . .).

An optimal dynamic programming algorithm should probably be used if we aim to compute triangulations on a large scale.

Such will nevertheless be more difficult to program and the basic Ray-Seidel algorithm is convenient for exploring the topic.

The Ray-Seidel algorithm

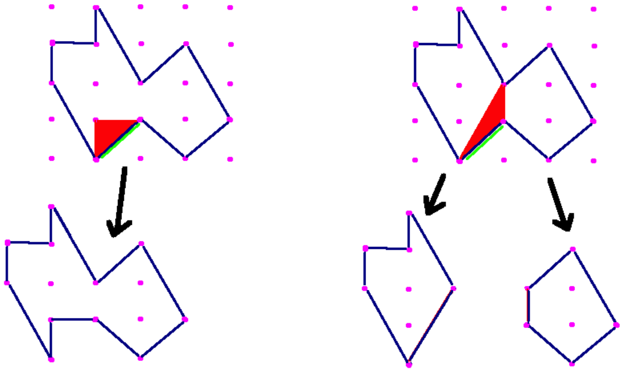
In the case where the triangle does not cut P in two, the set of triangulations containing t is isomorphic to the set of triangulations of P'. There are two sub-cases to treat depending on whether t shares one or two edges with P.

However, it can happen that the triangle t cuts P into two polygons P' and P''. Then, the set of triangulations containing t is the product of the set of triangulations of P' with the set of triangulations of P''.

In this case, we pursue the algorithm separately for P' and P''. The proof time is the sum rather than the product. There is clearly a significant gain of time to be had when this happens.

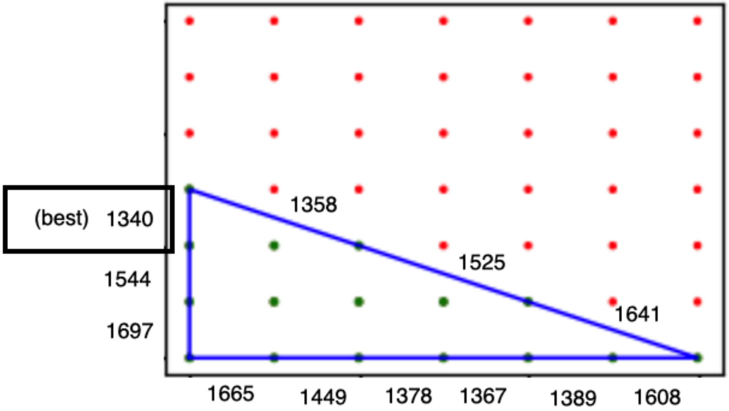
Their algorithm therefore raises the question of finding a good *strategy* : how to choose the edge e at each stage so as to minimize the number of steps. Measure by counting the number of *active nodes* in the proof tree.

Strategy



Two of the possibilities obtained after a cut along the edge in green, leading to $1 + 2 = 3$ resulting polygons. (There is also a third possibility not pictured.)

Strategy



Here are the minimal numbers of proof steps after choosing that edge.
The problem is to predict which is the best edge.

Average versus minimal proof length

We use as a basic test case the square of size 3×3 having 46456 triangulations.

Applying our version of the Ray-Seidel algorithm with a random choice of edge at each step, leads to a proof tree with around 25000 active nodes.

We can, on the other hand, exhaustively compute all the possible proofs for an example of this small size. On anything too much bigger that will be impossible.

The minimal number of active nodes is 3291. It depends of course on the specific details of how you count, for example any time we get to a polygon contained in a unit square that is considered as done.

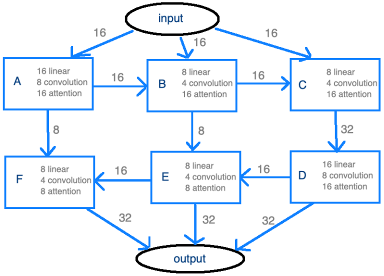
The best edge to choose at the first stage is the one in the middle; choosing a corner edge leads to around 3900 active nodes.

For this example then, the difference between the size of the proof with a random choice of edges, and the minimal size, is around a factor of 7. We can expect this factor to grow by a lot for bigger examples.

Using machine learning

The basic schema will be to input a polygon into the machine and ask it to tell us which edge to choose; we then construct the proof by using the suggested edge at each stage.

Network architecture



Using machine learning

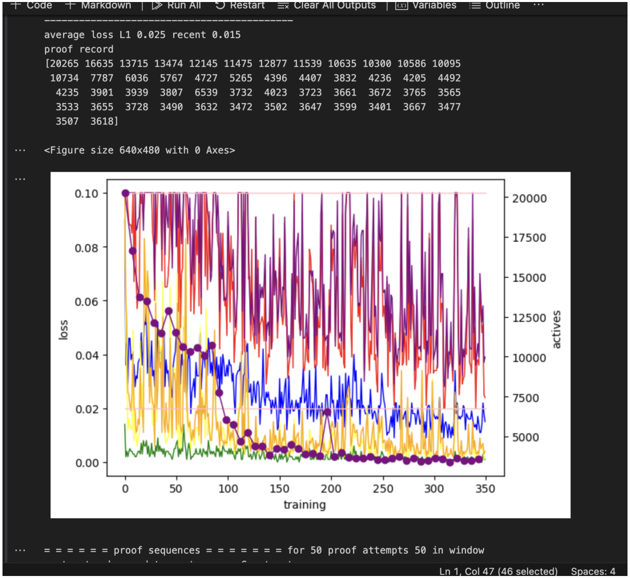
With an optimized choice of neural network architecture and training hyperparameters, we have been able to obtain a good result on this first example: the machine can find a proof with around 3320 active nodes (with a best of 3308). This is less than 1% over the minimal amount.

Unfortunately, the amount of training time necessary is large. We do not have results indicating a net time gain, however that might be possible on larger examples.

It is not clear at this stage whether the total machine time (training + computation) can be smaller using a machine learning model than with a classical algorithm where one might try basic heuristics to reduce the size of the proof.

In the semigroup case, that was the subject of our “global approach” slides.

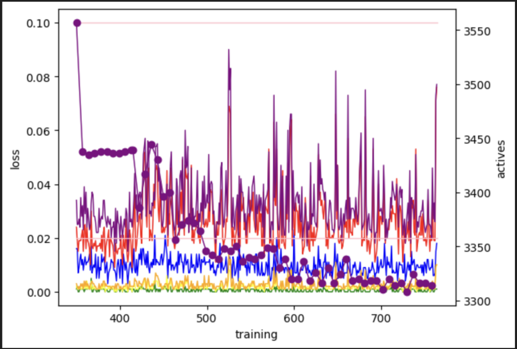
Training results on the 3 × 3 case



```
-------------------------------------------
average loss L1 0.025 recent 0.015
proof record
[20265 16635 13715 13474 12145 11475 12877 11539 10635 10300 10586 10095
 10734  7787  6036  5767  4727  5265  4396  4407  3832  4236  4205  4492
  4235  3901  3939  3807  6539  3732  4023  3723  3661  3672  3765  3565
  3533  3655  3728  3490  3632  3472  3502  3647  3599  3401  3667  3477
  3507  3618]

<Figure size 640x480 with 0 Axes>

= = = = = = proof sequences = = = = = = = for 50 proof attempts 50 in window
```

Training results on the 3 × 3 case

```
    proof record
    [3557 3438 3435 3436 3438 3438 3436 3436 3438 3439 3439 3386 3417 3444
     3430 3396 3400 3356 3370 3374 3372 3364 3346 3342 3338 3348 3346 3350
     3336 3340 3338 3342 3349 3348 3330 3338 3320 3320 3336 3318 3326 3316
     3330 3316 3324 3338 3318 3320 3316 3318 3318 3310 3320 3314 3316 3308
     3324 3316 3316 3314]

... <Figure size 640x480 with 0 Axes>

...
```

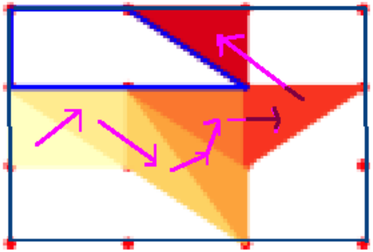


```
... = = = = = = proof sequences = = = = = = = for 110 proof attempts 60 in window
```

We find a proof with 3308 active nodes, after a little over 100 iterations of the basic training sequence. The neural networks have undergone around 12,000 (Adam) gradient descent steps.

Training results on the 3 × 3 case

Here is a sample piece of the resulting strategy:



In this example it seems pretty close to what the human user would intuitively select. However, there are many intermediate nodes where the machine finds a significantly better choice than what a human user would be able to guess.

Architecture

Following the idea introduced in AlphaEtc, we set up two neural networks training side-by-side:

- The **value** network tries to predict the number of active nodes below the current node in the proof tree; its output is a scalar for each input polygon.
- The **policy** network tries to predict the combined value outputs obtained from the resulting collections of polygons, for each edge in the input polygon; its output is a vector, of length equal to the perimeter, for each input polygon.

The policy network is what we use to determine the choice of edge at each step: we take the edge whose policy value is the smallest.

The value network is used to help train the policy network.

Architecture: reinforcement learning

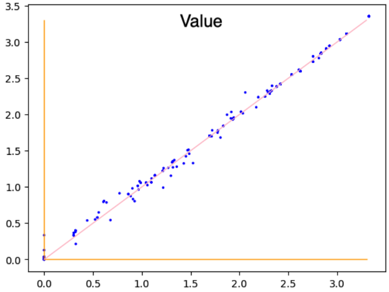
Our goal is to minimize the number of active nodes below the given node of the proof tree, depending on the choice of edge to cut upon.

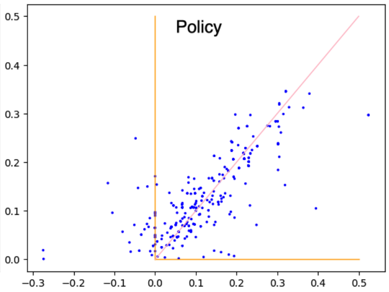
This number of active nodes is going to depend, in turn, on the strategy that will later be used to treat the proof after that node.

The value-policy approach is well adapted to this situation. Here we have a canonical reward structure using the prediction of the remaining proof size. It is important to get as accurate as possible an estimation of the underlying true values of these quantities.

Architecture: value and policy accuracies

Compare the relative accuracies of the value and policy networks on their training data:





Notice that it is much easier to train for high accuracy with a scalar output, whereas the machine will find it more difficult to differentiate between the effects of choice of different edges.

The value network provides a useful way of generating the training data for the policy network.

Training procedure

We adopt a step-by-step recursive training philosophy.

A training data element consists of an active polygon[1] P. Training is done using batches of these. All computations are done in batches using PYTORCH tensor operations.

To train the policy network we choose at random an edge e of P on which to make a cut. Then the value network is evaluated on the resulting polygons P' and the results are combined to get a computed value for (P, e). This value is compared with the predicted policy value for (P, e) and the policy network is trained with Adam gradient descent on the batched loss (combined L1 and MSE) between the computed and predicted values.

[1]When there is an edge that has just one associated triangle we can automatically proceed forward on that triangle, so active polygons are ones where these steps have all been processed.

Training procedure

To train the value network at P we output the policy array and choose the edge e_0 with smallest policy value, then make the cut at e_0, calculate the value of P using the value network recursively on the resulting polygons P', and combine these together, adding 1 for P itself, to get the value computation for P.

Again, the value network provides a predicted value for P and we train on the loss between these two.

Having computed the full policy array for P, we can modify the training of the policy network to include the minimal edge e_0 and add a component of the policy loss function to try to rule out having the value computation for the random edge e be smaller than then predicted policy minimum at e_0.

Training procedure: recursivity

The values on a polygon P are obtained recursively using the networks as intermediaries, because the training is done for just a single proof step at each stage. The proof steps are not concatenated together at the training stage.

(However, they will be concatenated at the stage of creating the training data pool.)

We avoid doing something like taking the minimal number of proof steps computed separately for small cases such as 3×3 and using those directly to train the networks.

This makes it somewhat remarkable that the neural networks would be able to predict values accurately enough to choose the good cuts at each proof step.

Recursive training for other quantities

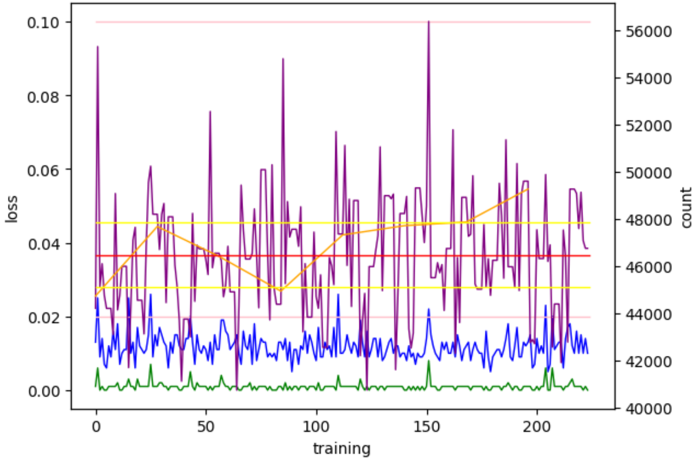
The same idea of recursively training a network to predict some quantities can also be applied to other quantities. The main one that comes to mind is the number of triangulations itself.

We can set up a network (pretty much the same as the value network) to try to predict the number of triangulations of a polygon. Given P and an edge e, making the cut at e yields a collection of new polygons P' and the number of triangulations of P can be calculated by a combination of the results for the P'. We can therefore apply the same training method as described above to train this network.

It will help to have a good policy network to make an optimized choice of edge e at each stage.

By the way, we take the $\log_{10}$ of all numerical quantities before trying to approximate them with a neural network.

Recursive training for other quantities



An older attempt to approximate the value of 46456 triangulations of the 3×3 square.

Recursive training for other quantities

It is perhaps not unexpected that it is not very easy to obtain a precise numerical value in such a recursive way. Experiments so far have not given perfect results, and the behavior seems to depend a lot on all the hyperparameter choices involved.

This is going to be a little bit like trying to control the top block of a stack of blocks by holding it at the bottom, when the interfaces between the blocks are not completely flat.

We indeed observe oscillations around the expected value and the networks do not seem to "like" to converge nicely to the expected value.

This is the subject of ongoing work.

Potential applications could include the approximation of other quantities of interest for Mathematical Physics. The basic criterion is that the quantity for P should be computable, after cutting at an edge e, from the corresponding quantities for the resulting polygons P'.

Neural network design

A polygon is a cyclically ordered set of edges. There is no preferred starting point. We would like to make the computations as invariant as possible, within reason. The rationale includes avoiding the phenomenon of “memorization” that would severely reduce the potential for scalability and generalizability.

The famous **attention mechanism** that plays a key role in “Transformers” and hence in LLM's, possesses good properties in this situation. We can indeed think of a polygon as being like a phrase with the edges being the words.

The computations should also be invariant under translations on the lattice. We therefore calculate a barycenter of the polygon to use as origin in order to create a positional encoding.

Ideally it would be good to include invariance under the dihedral group too, but that looks to be out of reach at the moment.

Neural network design

The networks therefore manipulate vectors of features, one for each edge, where the number of elements is the perimeter of the polygon, hence not the same from one instance to another.

In the attention layers, each edge interrogates all the other ones with the query-key-value process.

The `key_padding_mask` allows to take into account the differing number of active elements in the vectors for different polygons in the batch.

We can also use fully connected feedforward linear layers acting on the edges individually.

Another crucial type of layer is a convolutional layer on the initial lattice. In order to preserve translation invariance, we pad this and then restrict to only the boundary or interior vertices of the polygon. This allows the machine to develop a global geometric picture of the polygon in the plane rather than having to integrate out the information from the collection of edges.

Neural network design

Stacking these all together in the order

- convolution layers
- attention layers
- linear feedforward layers

yields the design for the policy network. The output is a vector with one element for each edge.

For the value network, an additional step is needed to pass from a vector to a scalar.

Neural network design

For this, we use an attention layer where the query values are a vector, of fixed length, of fourier values. This is queried against the vector features of the previous layers in the key and value spots.

The vector of fourier values is then reshaped into a single scalar feature that may then be passed to the feedforward linear layers.

The resulting design of the value network is:

- convolution layers
- attention layers
- consolidation from vector to scalar using an attention layer with fixed query
- linear feedforward layers

Defining a good collection of parameters: sizes and numbers of layers, is still a question in progress. I do not have enough computing capacity to do any kind of exhaustive search, so the goal for the moment is to try to develop a good intuition for what works best.

Triangulations — conclusion

Looking at a dynamic programming problem for the classification of triangulations of lattice planar polygons, we can set up a machine learning approach to choosing the locations of cuts at each node of the classification proof.

This provides the opportunity to test the design of neural networks for the analysis of geometrical data, the processes needed to produce balanced sets of training data, and the general architecture for this reinforcement learning question.

Although the training time on a simple example is long, the machine can get very close to finding a proof of minimal length.

This suggests that it should be possible to obtain significant gains of time in classification proofs at a larger scale. We will be looking for applications to the Mathematical Physics setting.

The definition of metrics to measure the progress and hence to define an appropriate reinforcement learning setup should be relevant when we try to approach more general mathematical proofs.

Next : proofs in set theory

The current project most closely aligned with the Malinca workflow is to adapt these techniques to the case of proofs in set theory.

One of the first desiderata is to have a rudimentary proof-checking program written in Pytorch. There are a few reasons for this:

- Speed: at a sub-expert level of programming, a language such as Pytorch gives us access to speeds approaching the top speed of loop-style programming in C.
- Parallelization: a language that supports direct utilisation on GPU's will be better adapted to scaling up the size of things.
- Batches: if we want to treat big amounts of training data, it will be useful to be able to do computation in batches.
- ML interface: it will be convenient and faster to pass information back and forth with instances of neural networks if the underlying logic is written in the same language.

Next : proofs in set theory

For these reasons, the next steps in the process will involve programming a proof formalization method in Pytorch. This would probably be loosely based on the ideas going into automated theorem provers such as VAMPIRE.

Subsequent steps will include the collection of a sufficient number of relatively small examples to use as training examples.

We have seen above that even a single instance (for example our triangulation question just for the 3x3 square) can generate large quantities of reinforcement learning training data, so we do not necessarily need to look for large collections of “math problems”. The small steps inherent in a basic development of ZF set theory should be sufficient.

Then: neural network architecture, training data pools, ...