Abstract

Providing examples is one of the most common ways for end-users to interact with program synthesizers. However, program synthesis systems assume that examples consistent with the program are chosen at random, and do not exploit the fact that users choose examples pragmatically. Prior work [1] modeled program synthesis as pragmatic communication, but required an inefficient enumeration of the entire program space. In this paper, we show that it is possible to build a program synthesizer that is both pragmatic and efficient by approximating the joint distribution of programs with a product of independent factors, and performing pragmatic inference on each factor separately. This factored distribution approximates the exact joint distribution well when the examples are given pragmatically, and is compatible with a basic neuro-symbolic program synthesis algorithm. Surprisingly, we find that the synthesizer assuming a factored approximation performs better than a synthesizer assuming an exact joint distribution when evaluated on natural human inputs. This suggests that humans may be assuming a factored distribution while communicating programs.

1 Introduction

Program synthesizers are systems that take a specification of user intent as input, and synthesize a program in a domain-specific language (DSL) that satisfies the specification. Providing examples is the standard form of specification for end users of state-of-the-art program synthesizers, as examples are both intuitive for humans to provide and explicitly checkable by machines. However, program synthesizers typically assume that examples are chosen randomly [2, 3, 4, 5], and don’t leverage the fact that humans choose examples pragmatically to convey their intent [6]. As a consequence, existing synthesizers are both inefficient – having to reason over a complex space of programs conditioned on random specifications, and unintuitive – not treating the given specification pragmatically.

Prior work [1] posed program synthesis as a reference game, allowing for the application of the Rational Speech Acts (RSA) model [7] towards pragmatic program synthesis. They have shown that by treating user given examples pragmatically, one can build synthesizers that are more intuitive to use and require fewer examples when compared to the non-pragmatic ones. However, the formulation in [1] considers each program as an atom, and requires multiple enumerations over the entire program space. This does not scale to combinatorially complex program spaces. Thus, one is left with a dilemma: One can either make a synthesizer pragmatic at the cost of extreme inefficiency, or make it efficient and scalable to a large space of programs at the cost of it being non-pragmatic.

We take a step towards making a program synthesizer that is both pragmatic and efficient by factorizing the distribution of programs as a set of independent production rules as prescribed by the DSL’s grammar. This factorization completely disregards the complicated correlative structures between these production rules necessary to faithfully model the space of programs given the specification.
Indeed, under a literal specification that doesn’t actively constrain the number of consistent programs, this naive factorization will generate programs that fail to satisfy the specification nearly every time – it is a bad approximation. However, under a pragmatic specification that selects a smaller number of consistent programs and is thus more informative, the factorization preserves the sparsity of the joint distribution, and performs on par with a literal listener that explicitly models the exact joint distribution at a fraction of the inference cost. On the human speaker data of (1), the factored literal listener performs even better than its exact counterpart, suggesting that humans may intuitively be assuming a factored distribution while communicating to the synthesizers.

The factored representation also makes it efficient to perform the recursive reasoning required for RSA algorithm, as each factor is a distribution over a small number of concepts. We find that the factored pragmatic synthesizer achieves reasonably good performance when compared to its exact counterpart. More importantly, our factorized approach allows for the introduction of learned (i.e. neural symbolic) program synthesizers, which makes it applicable to large program spaces.

2 Efficient Pragmatic Synthesis

As in (1), we model program synthesis as a reference game between two agents – a speaker $S$ and a listener $L$. The speaker chooses a specification – a set of examples $D = u_1,u_2,\ldots,u_n$ – to communicate a program $h$ to the listener. In keeping with the literature on the Rational Speech Acts model, we also refer to an example as an utterance. The communication is successful if the listener is able to infer, or synthesize, the correct program given the speaker’s utterances.

2.1 Exact Pragmatic Program Synthesis

In this communication game setting, the task of synthesis is to model the listener distribution $P_L(h|D)$ of programs given utterances. To model a pragmatic listener, (1) propose using the Rational Speech Acts (RSA) model (2) to recursively reason about a speaker generating utterances according to a speaker distribution $P_S(D|h)$ of utterances given a program.

The literal listener $L_0$ reasons about the lexicon $l(h, D)$. The lexicon function takes the value 1 if the program $h$ is consistent with a specification $D$, and 0 otherwise. The literal listener is defined in Equation (1). The pragmatic speaker $S_1$ reasons about a literal listener. Since modelling a distribution over all specifications is intractable, (1) factorize the speaker distribution over specifications autoregressively into a product of distributions over utterances (eqs. (2) and (3)). The pragmatic listener $L_1$ reasons about $S_1$ to pragmatically synthesize programs (eq. (4)).

Under this formulation, the distribution $P(h|D)$ is over the space of all programs, making pragmatically considering all possible alternatives intractable outside of simple domains.

2.2 Efficient Synthesis with a Mean-field Approximation

Instead of viewing the program as an atomic referent, we view a program as a finite sequence of derivations in the grammar of the DSL. The grammar defines a set of $K$ non-terminals, and a number of rules $N_i \rightarrow \alpha_1^i, \ldots, N_i \rightarrow \alpha_n^i$ that expand $N_i$. Each step of the derivation selects a rule $N_i \rightarrow \alpha_j^i$, which expands a non-terminal $N_i$. We let $R_i$ denote the rule $N_i \rightarrow \alpha_j^i$ that expands the non-terminal $N_i$ in a program. When the production rules are not mutually recursive, the program can be simply represented as a finite sequence of production values: $[R_1, R_2, \ldots, R_K]$. Figure (1) shows the specific DSL we use in our experiments and Figure (2) shows two sample programs from the DSL, represented as a finite sequence of productions.

Under this representation of the program, we can use a mean-field approximation (8, Ch. 10) to naively factorize the distribution of programs $P(h|D)$ as

$$Q(h|D) = Q^1(R_1|D)Q^2(R_2|D)\cdots Q^K(R_K|D)$$
We consider 6 different listener models – literal and pragmatic variants of 3 different types of models. These are the joint literal listener $L_{\text{J}}$, and the joint pragmatic listener $J_{\text{J}}$, since they model the full joint distribution from eqs. (1) and (4) respectively. The second type is based on the factorized

**3 Experiments**

As an exploratory study, we evaluate our approach on a simple layout domain [1], where exact inference over the space of programs is possible. In this domain, a program is a pattern on a grid formed from a set of objects. These objects may be a colourless pebble, or a chicken or pig that may be red, green or blue. Each utterance reveals the object at one square on the grid, and the speaker has to communicate the pattern by revealing squares on the grid. The pattern is formed according to rules specified in the domain-specific language in Figure 1. The specification is a sequence of objects, specified by the coordinates of the object on the grid, and its shape and colour.

![Figure 1: Grammar of the DSL](image)

**3.1 The Models**

We consider 6 different listener models – literal and pragmatic variants of 3 different types of models. The first type is the models from [1], which enumerate the entire program space during inference. This approximation allows us to reason over each non-terminal independently. Appendix A shows that minimizing the forward KL distance between $P(h|D)$ and $Q(h|D)$ amounts to performing supervised learning on the marginal distribution of $P$ on each factors $Q^i$ separately.

Pragmatic synthesis under the mean-field approximation amounts to carrying out the recursive computation for each productions independently (tractable as each $Q^i$ operates over a small number of productions), similar to Equations (1) to (4). This is shown Equations (6) to (8). $Q_{S_1}^{i,utt}$ is defined autoregressively using $Q_{S_1}^{i,utt}$ as in $P^{spec}_{S_1}$.

The lexicon $l^i$ for the $i^{th}$ rule (eq. (5)) represents the fraction of all programs that satisfy the specification where the $i^{th}$ non-terminal is expanded using the rule $R_i$. We can use this notion of the lexicon to compute the factorized literal listener as a marginal over all other factors, which is the best approximation under the factorization. Alternatively, the distribution $Q_{L_0}^{i,utt}$ can be parametrized using a neural network trained to approximate the literal listener distribution. Given $Q$, we enumerate programs in decreasing order of probability until we find a consistent program, or run out of the search budget (Appendix B).

\[
Q_i(R_i, D) = \frac{\sum_{h, R_i \in h} l^i(h, D)}{\sum_h l^i(h, D)} \quad \text{(5)}
\]

\[
Q_{L_0}^{i}(R_i|D) = \frac{\sum_{R_i} l^i(R_i, D)}{\sum_{R_i} l^i(R_i, D)} \quad \text{(6)}
\]

\[
Q_{S_1}^{i,utt}(u_j|R_i, u_j^{i-1}) = \frac{\sum_{u_j} Q_{L_0}^i(R_i|u_j^{i-1}, u_j)}{\sum_{u_j} Q_{L_0}^i(R_i|u_j^{i-1}, u_j)} \quad \text{(7)}
\]

\[
Q_{S_1}^{i,utt}(D|R_i) = \frac{\sum_{R_i} Q_{S_1}^{i,utt}(D|R_i)}{\sum_{R_i} Q_{S_1}^{i,utt}(D|R_i)} \quad \text{(8)}
\]
Figure 2: Two patterns in our layout domain and their corresponding programs, represented as a sequence of production rules: \[ \text{Program, Shape, Left, Right, Top, Bottom, Thickness, O, I, Colour, A1, A2} \]. The symbol \( \cdot \) indicates rules which only have 1 choice of expansion (Program, Shape, and Colour). The rules where these two programs differ are marked with a box.

Listener distributions from eqs. (6) and (8). In this type, the literal listener distributions are obtained by enumerating over the space of programs. These are termed the factorized literal listener \( L_{F0} \) and the factorized pragmatic listener \( L_{F1} \). The third type of model is also based on the factorized listener distributions, but the literal listener distribution is approximated using a neural network (Appendix D). These are termed the neural literal listener \( L_{N0} \) and the neural pragmatic listener \( L_{N1} \).

3.2 The Data

We use speaker models to generate specifications from 1000 randomly chosen programs from the DSL. We use a literal speaker \( S_{M0} \) to randomly generate utterances that are true of the program, and a pragmatic speaker \( S_{M1} \) to generate utterances under the joint speaker distribution in eq. (2). These are idealized cases of a perfectly literal and pragmatic (according to RSA) listeners respectively.

We leverage human speaker data from \( \Pi \), collected by interactions between Human \( \rightarrow L_{J0} \) and Human \( \rightarrow L_{J1} \). We denote a human speaker interacting with \( L_{J0} \) as \( S_{H0} \), and a human speaker interacting with \( L_{J1} \) as \( S_{H1} \). Note that even when interacting with \( L_{J0} \), a human speaker still behaves pragmatically (i.e. \( S_{H0} \neq S_{M0} \)). Details of how the data are processed are presented in Appendix E.

Figure 3: Comparison of different listeners (represented using different line types), literal: \( L_{J0}, L_{F0}, L_{N0} \) and pragmatic \( L_{J1}, L_{F1}, L_{N1} \), interacting with different speakers (represented using different line colors) \( S_{M0}, S_{H0}, S_{H1} \) and \( S_{M1} \). X-axis shows number of utterances presented (i.e. user efforts) and Y-axis shows fraction of programs inferred correctly (i.e. communication accuracy).
Figure 4: Heatmaps showing the distribution over the rules for two different non-terminals in the DSL – \texttt{Left} and \texttt{Right} – that specify the left and right edges of the pattern respectively, under the specification shown on the left. Each case shows the distribution produced by one type of listener (literal \texttt{L_{J0}}/\texttt{L_{F0}}, or pragmatic \texttt{L_{J1}}/\texttt{L_{F1}}) under a specification produced by a speaker model. In each case, the heatmap on the left shows the joint distribution over these two productions under the a joint listener model. The heatmap on the right shows the joint distribution over these two productions under a factorized model, with the distribution of each factor shown as a histogram. The rules that satisfy the program are indicated with a star.

3.3 Results

In Figure 3a, we note that an idealized pragmatic speaker \texttt{S_{M1}} allows all the literal listener models \texttt{L_{J0}}, \texttt{L_{F0}}, and \texttt{L_{N0}} to perform well. This suggests that under pragmatic specifications from \texttt{S_{M1}}, the factorized distribution of \texttt{L_{F0}} approximates the exact joint distribution \texttt{L_{J0}} well, due to informative specification concentrating the probability mass in a smaller region that’s amendable to a factorized model. We illustrate this effect on a concrete instance in Figure 4. This affirms our intuition that a pragmatic specification should make inference easier, and allows for an extremely simple neuro-symbolic synthesizer. As expected, performance on uninformative specifications from \texttt{S_{M0}} is poor for both literal and pragmatic models.

Surprisingly, the factored literal listener \texttt{L_{F0}} performs better than the exact model \texttt{L_{J0}} when given human generated specifications of \texttt{S_{H0}}. This is surprising as \texttt{S_{H0}} is expressly collected in 1 from the interactive pair \texttt{S_{H0} \rightarrow L_{J0}}. This suggests that humans may intuitively be assuming a factored distribution while communicating with the synthesizer, for instance, choosing examples to communicate a particular coordinates of one of the edges of the pattern.

For the pragmatic listeners in Figure 3b, we note that the factored model \texttt{L_{F1}} approximates the exact model \texttt{L_{J1}} well when given idealized specifications from \texttt{S_{M1}}, i.e. the communicative pair \texttt{S_{M1} \rightarrow L_{F1}} performs as well as \texttt{S_{M1} \rightarrow L_{J1}}. However, using human-generated specifications \texttt{S_{H1}}, we find that the pair \texttt{S_{H1} \rightarrow L_{F1}} performs worse than \texttt{S_{H1} \rightarrow L_{J1}}. We suspect this is due to an “interaction gap” where data from \texttt{S_{H1}} was collected by 1 during interaction between the pair \texttt{S_{H1} \rightarrow L_{J1}}. The human speaker had the opportunity to adjust their strategy and adapt specifically to \texttt{L_{J1}}, but could not dynamically adapt to \texttt{L_{F1}} in our experiments.

We also note that the neural model \texttt{L_{N1}} performs worse when compared to \texttt{L_{J1}} and \texttt{L_{F1}}. We suspect this is due to approximation errors of the neural model accumulate when different factors are combined (via multiplication), and during RSA’s recursive reasoning process. This results in larger gaps between \texttt{L_{N1}} and \texttt{L_{F1}} than between \texttt{L_{N0}} and \texttt{L_{F0}}.
4 Discussion and Future Directions

We find that a synthesizer using a naive factorized distribution over programs performs well, but only when given informative, pragmatic specifications such as those a generated by a human. One explanation is that informative specifications concentrating the probability mass over a smaller number of programs, as illustrated in Figure 4. Another explanation is that our DSL factors the hypothesis space similarly to how a human would factor it. This results in a communication scheme that focuses on conveying each factor independently, without significant loss of information. Formally characterizing this notion of “concentrating the probability mass” (perhaps in an information theoretic language) and conducting user studies exploring how humans conceptualize and reason about large, compositional space of programs are exciting directions for future work.

We also observe that our current neural pragmatic model $L_{N_1}$ performs poorly. We postulate that one can obtain better results by (i) having a better model and training schemes for approximating eq. (6) and (ii) the opportunity to interact directly with users.

References

[1] Yewen Pu, Kevin Ellis, Marta Kryven, Josh Tenenbaum, and Armando Solar-Lezama. Program synthesis with pragmatic communication. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, Advances in Neural Information Processing Systems, volume 33, pages 13249–13259. Curran Associates, Inc., 2020.

[2] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, editors, Advances in Neural Information Processing Systems, volume 32. Curran Associates, Inc., 2019.

[3] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In Proceedings of the 34th International Conference on Machine Learning - Volume 70, ICML’17, page 990–998. JMLR.org, 2017.

[4] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. SIGPLAN Not., 46(1):317–330, January 2011.

[5] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In John Paul Shen and Margaret Martonosi, editors, Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006, pages 404–415. ACM, 2006.

[6] Patrick Shafto, Noah D. Goodman, and Thomas L. Griffiths. A rational account of pedagogical reasoning: Teaching by, and learning from, examples. Cognitive Psychology, 71:55–89, 2014.

[7] Michael C Frank and Noah D Goodman. Predicting pragmatic reasoning in language games. Science, 336(6084):998–998, 2012.

[8] Christopher M. Bishop. Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag, Berlin, Heidelberg, 2006.

A Factorized Approximation of Joint Listener Distribution

For simplicity, let us consider a setting where $K = 2$. So, we have the joint distribution $P(h|D)$ approximated by $Q(h|D) = Q_1(R_1|D)Q_2(R_2|D)$. 
We would like to minimize the quantity $KL(P(R_1 R_2 | D) || Q(R_1 R_2 | D))$. 

$$\min_{Q} KL(P(R_1 R_2 | D) || Q(R_1 R_2 | D)) = \min_{Q^1, Q^2} KL(P(R_1 R_2 | D) || Q^1(R_1 | D) Q^2(R_2 | D))$$

$$= \min_{Q^1, Q^2} \sum_{R_1, R_2} P(R_1 R_2 | D) \log \left( \frac{P(R_1 R_2 | D)}{Q^1(R_1 | D) Q^2(R_2 | D)} \right)$$

$$= \min_{Q^1, Q^2} \sum_{R_1, R_2} P(R_1 R_2 | D) \log P(R_1 R_2 | D) - \sum_{R_1, R_2} P(R_1 R_2 | D) \log (Q^1(R_1 | D) Q^2(R_2 | D))$$

$$= \max_{Q^1} - \sum_{R_1, R_2} P(R_1 R_2 | D) \log (Q^1(R_1 | D) Q^2(R_2 | D))$$

$$= \max_{Q^1} \sum_{R_1, R_2} P(R_1 R_2 | D) \log Q^1(R_1 | D) + \max_{Q^2} \sum_{R_1, R_2} P(R_1 R_2 | D) \log Q^2(R_2 | D)$$

We can maximize each term of the sum separately. Without loss of generalization, let’s focus on $Q^1$.

$$\max_{Q^1} \sum_{R_1, R_2} P(R_1 R_2 | D) \log Q^1(R_1 | D)$$

$$= \max_{Q^1} \sum_{R_1} \sum_{R_2} P(R_1 R_2 | D) \log Q^1(R_1 | D)$$

$$= \max_{Q^1} \sum_{R_1} \log Q^1(R_1 | D) \sum_{R_2} P(R_1 R_2 | D)$$

$$= \max_{Q^1} \sum_{R_1} P(R_1 | D) \log Q^1(R_1 | D)$$

$$= \min_{Q^1} - \sum_{R_1} P(R_1 | D) \log Q^1(R_1 | D)$$

We see that minimizing the KL divergence can be reduced to minimizing the cross-entropy loss between the factor of the mean-field approximation corresponding to a non-terminal symbol and the marginal of the joint distribution over all other non-terminals.

To train a learned model, we can sample programs and corresponding specifications and train with this objective.

$$\min_{Q^1} \mathbb{E}_{r_1 \sim P(R_1 | D)} \left[ - \log Q^1(R_1 | D) \right]$$

**B Search algorithm**

Algorithm describes the search algorithm. We use a best-first search algorithm, where we start with the highest probability program by considering the highest probability rule for each non-terminal. We then iterate over programs in decreasing order of probability. At each step, we replace one rule in the program with the next best alternative for that rule, and add it to a priority queue. We terminate the search when we find a consistent program, or exceed the search budget. For our experiments, we use a search budget of 50.

**C Data processing**

**C.1 Speaker models**

We use the speaker models $S_{M_0}$ and $S_{M_1}$ to generate specifications according to an idealized literal and pragmatic speaker distribution respectively. The literal speaker model $S_{M_0}$ selects utterances
We train a 2-hidden layer MLP, with each hidden layer having 256 units, as a neural literal listener. We consider the utterances as a sequence, in the other they were provided by the user. We process the
\[ A \]
A common model for each production is trained for all productions. The model produces a
\[ B \]
listener model is presented with incrementally larger prefixes of this sequence of utterances, and
\[ C \]
the synthesis experiments are conducted in a similar manner to the experiments with data from
\[ D \]
Algorithm 1

|Require: Specification \( D \), distributions over rules \( Q^i(R_i|D) \), search budget \( B \) |
|Ensure: Program \( (R_1, \ldots, R_K) \) which satisfies \( D \) |
|for \( i = 1 \) to \( K \) do |
|Sort rules \( R_i^j \) in decreasing order of \( Q^i(R_i^j|D) \) |
|end for |
|Searched set \( S \leftarrow \{\} \) |
|Priority queue \( P \leftarrow [\] \) |
|enqueue(\( P, \langle R_1^1, \ldots, R_K^K \rangle \)) \quad \triangleright \text{Max priority queue with score } p |
|while \( |S| < B \) do |
|\( \langle R_1^1, \ldots, R_K^K \rangle \leftarrow \text{dequeue}(L) \) |
|if \( \langle R_1^1, \ldots, R_K^K \rangle \in S \) then |
|continue |
|end if |
|if \( \langle R_1^1, \ldots, R_K^K \rangle \) is consistent with \( D \) then |
|return \( \langle R_1^1, \ldots, R_K^K \rangle \) |
|end if |
|\( S \leftarrow S \cup \{\langle R_1^1, \ldots, R_K^K \rangle\} \) |
|for \( i = 1 \) to \( K \) do |
|if replacing \( R_i^j \) with \( R_i^{j+1} \) results in a valid program then |
|enqueue(\( P, \langle R_1^1, \ldots, R_i^{j+1}, R_i^{j+1}, R_{i+1}, \ldots, R_K^K \rangle \)) \quad \triangleright p = \sum_i \log Q^i(R_i^j|D) |
|end if |
|end for |
|end while |

that are true of the program with uniform probability. The pragmatic speaker model \( M_1 \) generates specifications autoregressively, at each step choosing the utterance with the highest probability according to Equation (3). For each program, we generate a specification of size 15 using each of the models \( S_{M_0} \) and \( S_{M_1} \) as a sequence of utterances \( u_1, u_2, \ldots, u_N \).

A listener model is presented with incrementally larger prefixes of this sequence of utterances, and the synthesis is terminated if the intended program is correctly identified by the listener model.

C.2 User interaction data

For each model \( L_{J_0} \) and \( L_{J_1} \), we have a set of trials collected by [1]. We term a user interacting with \( L_{J_0} \) as \( S_{H_0} \) and a user interacting with \( L_{J_1} \) as \( S_{H_1} \). Each trial presents the user with a program in the DSL, and the user is tasked with communicating the program to the model. The user reveals a sequence of utterances \( u_1, u_2, \ldots, u_N \).

We consider the utterances as a sequence, in the other they were provided by the user. We process the specifications to remove duplicate utterances within a trial, and retain only the first occurrence of the utterance. We have a total of 371 trials for \( S_{H_0} \) and 386 trials for \( S_{H_1} \).

The synthesis experiments are conducted in a similar manner to the experiments with data from speaker models.

D Neural models

We train a 2-hidden layer MLP, with each hidden layer having 256 units, as a neural literal listener \( N_0 \). The input to this model is a \( 7 \times 7 \times 6 \) tensor \( T \). For each element of the specification \((x, y, s, c)\), denoting the \( x \)-coordinate, \( y \)-coordinate, shape, and color (shape and color are mapped to an index between 0 and 2) respectively is encoded by setting \( T[x, y, s] = 1 \) and \( T[x, y, 3 + c] = 1 \). All other elements of the input are set to 0.

A common model for each production is trained for all productions. The model produces a \( 12 \times 7 \) matrix \( Q \), where \( Q_{ij} \) represents the probability that a rule \( R_j \) is chosen for the \( i \)th factor. The loss
for each factor is computed as described in Appendix A and summed to obtain the total loss for a sample.

The model is trained by sampling a batch of programs from a set of 10000 programs, sampling a specification using the literal speaker model \( S_{M_0} \) for each program in the batch, and computing the loss between the predicted distribution for each factor and the ground truth value for factor.

The model is trained with a batch size of 8 and specifications comprising of 2 to 25 elements for 150,000 steps.