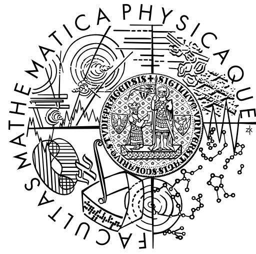


FACULTY OF MATHEMATICS AND PHYSICS
Charles University

MASTER THESIS

Martin Vastl

Deep Learning for Symbolic Regression

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Martin Pilát, Ph.D.
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2022

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I want to thank my supervisor Mgr. Martin Pilát, Ph.D., for all his support. Furthermore, I would like to thank all my friends and my girlfriend who helped me get through the whole master's degree. Many thanks also go to my parents, who supported me through my studies.

Title: Deep Learning for Symbolic Regression

Author: Martin Vastl

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., department

Abstract: Symbolic regression is task of finding mathematical equation based on the observed data. Historically, genetic programming was the main tool to tackle the symbolic regression, however, recently, new neural network based approaches emerged. In this work, we propose transformer based approach which predicts the expression as a whole without the need of finding the expression coefficients in post-processing step. We also introduce gradient local search to further improve the expression coefficients. We compare our solution to previous approaches on several benchmarks and demonstrate, that our solution is comparable in terms of model performance while outperforming them in terms of speed of prediction in average case.

Keywords: deep learning, symbolic regression, transformer

Contents

Introduction **3**

1 Background **5**
1.1 Symbolic regression . . . 5
1.2 Function representation . . . 5
1.3 Encoder-decoder architecture . . . 6
1.3.1 Recurrent neural networks . . . 6
1.3.2 Transformer . . . 7
1.3.3 Decoding . . . 10
1.4 Deep learning with sets as input . . . 12
1.4.1 Deep sets . . . 12
1.4.2 Set-Transformer . . . 13

2 Related work **16**
2.1 Genetic programming . . . 16
2.1.1 Initialization . . . 16
2.1.2 Fitness evaluation . . . 16
2.1.3 Selection, Mutation, Crossover . . . 16
2.2 Equation Learner . . . 17
2.3 Methods based on reinforcement learning . . . 18
2.3.1 Training . . . 18
2.3.2 Hybrid approach . . . 19
2.4 Transformer based approaches . . . 20
2.4.1 Data generation . . . 20
2.4.2 Coefficient handling . . . 21
2.4.3 Symbolic GPT . . . 22
2.4.4 Neural Symbolic Regression that Scales . . . 23
2.4.5 Deep Symbolic Regression for Recurrent Sequences . . . 23

3 Method **25**
3.1 Dataset . . . 25
3.2 Architecture . . . 26
3.3 Function representation . . . 29
3.4 Training . . . 30
3.5 Inference . . . 31

4 Experiments **32**
4.1 Metrics . . . 32
4.1.1 Classification metrics . . . 32
4.1.2 Coefficient metrics . . . 32
4.1.3 Hybrid metrics . . . 33
4.2 Training . . . 34
4.3 Dataset . . . 35
4.4 Local search . . . 36
4.5 Evaluation benchmarks . . . 36

5 Results and discussion **38**
5.1 Dataset hyperparameters 38
5.1.1 Effect of function representation 38
5.1.2 Effect of number of sampled points 39
5.1.3 Effect of number of equations 40
5.2 Architecture considerations 40
5.2.1 Effect of the embedding merge operation 41
5.2.2 Effect of the model size 41
5.2.3 Effect of the injected noise 42
5.3 Final model 42
5.3.1 Greedy and Top-K sampling results 42
5.3.2 Local optimization 43
5.4 More dimensions 44
5.5 Benchmark results 45
5.5.1 Performance of the 2D model 48
5.6 Discovering mathematical formulas 48
5.7 Generalization 49
5.7.1 Extrapolation ability 49
5.7.2 Effect of number of sampled points 49
5.7.3 Effect of sampling range 50
5.8 Ablation study 50
5.9 Acknowledgement 51

Conclusion **52**

Bibliography **53**

List of Figures **59**

List of Tables **60**

List of Algorithms **61**

List of Abbreviations **62**

A Attachments **63**
A.1 Dataset generator unnormalized probabilities 63
A.2 Model vocabulary 65
A.3 Examples of generated functions 66
A.4 Benchmark functions 67
A.5 Final hyperparameters 69
A.6 Effect of number of sampled equations 70
A.7 Full predictions 70
A.8 Benchmark results without local optimization 72
A.9 Comparison of 1D and 2D model 74
A.10 Effect of number of sampled points during inference 75
A.11 Model ability to perform outside of sampling range 77

Introduction

In the last decade, neural networks achieved impressive results in all areas of science, ranging from machine translation [Vaswani et al., 2017] and language modeling [Brown et al., 2020] to image classification [Deng et al., 2009] or protein folding [Senior et al., 2020]. These achievements were possible due to new architectures and a large amount of available data. Thanks to new technologies, we can communicate with foreign people in their native language, our smart devices can answer simple questions, and we are able to search through our photos just by using keywords. Deep learning also influences other science fields, such as physics, where they used deep reinforcement learning to control Tokamak [Degrave et al., 2022], or biochemistry, where they used deep learning to tackle protein folding with substantial precision [Senior et al., 2020].

One of the most natural questions that appear in other science areas is what formula explains measured inputs and outputs. Ideally, we would want to measure the process and then pass these data points into some clever algorithm, which would then output the equation that best fits the data. Such problem of finding underlying equation from inputs and outputs is called symbolic regression or equation discovery and was first proposed by Koza [1992]. The searched equation is usually made of variables, constants, and elementary functions, but it can even contain recurrence relations d'Ascoli et al. [2022] or be in the form differential equation. Symbolic regression was used by Matchev et al. [2021] to make a mass prediction of galaxy clusters more accurate. Wadekar et al. [2022] used symbolic regression to derive analytical expressions describing the relationship between the modulated stellar spectrum $M(\lambda)$ and the input atmospheric parameters. Besides these applications in natural sciences, symbolic regression was used to discover policy and value functions for reinforcement learning [Hein et al., 2017, Kubalík et al., 2019]. Symbolic regression can also be used as a tool to learn to represent classical machine datasets. Wilstrup and Kasak [2021] used symbolic regression to find the underlying equation for each of the Penn Machine Learning Benchmarks [Le et al., 2020] and outperformed several statistical machine learning approaches.

Historically, symbolic regression was approached by means of genetic programming firstly proposed by Koza [1992]. Since then, many authors have built on its foundations and it is still the prevailing method in symbolic regression [Schmidt and Lipson, 2009, Gomes et al., 2019]. Genetic programming is a bio-inspired approach that offers a natural way of finding the formula. It works by evolving expressions encoded as a tree using selection, crossover, and mutation. This approach, however, has several disadvantages. It can be slow, and the found formulas tend to increase in complexity without an additional performance. Since then, several various approaches have been proposed. Brence et al. [2020] use a probabilistic context-free grammar and the Monte-Carlo algorithm for grammar-based equation discovery to recover equations from the Feynman dataset [Udrescu and Tegmark, 2020]. Recently Martius and Lampert [2016], Sahoo et al. [2018], Werner et al. [2021] used neural networks by training them on function inputs and predicting the correct output values. However, instead of using one non-linear activation function through the whole network, they use elementary functions as activation functions and read the weights after the training to get the final

formula. The disadvantage of this approach is that the number of layers restricts the complexity of the final equation and that there is no straightforward way to handle functions that do not have real numbers as their domain. Another neural-based approach is proposed by Petersen [2019], Mundhenk et al. [2021] where they have used RNN to predict the function. Then they have used reinforcement learning to train the network, and the coefficients are found using global optimization. The disadvantage of these previously mentioned approaches is that they are always trained from scratch, which can be slow and impractical for some applications. To solve this issue Valipour et al. [2021], Biggio et al. [2021] propose an approach where they pre-train transformer [Vaswani et al., 2017] or transformer encoder on points and ground-truth equations to predict expressions without coefficients and find the coefficients in the post-processing step using global optimization. d'Ascoli et al. [2022] extends the transformer-based approaches to recurrent equations and predicts the coefficients jointly with the symbolic representation.

To solve the issues of previous methods, we propose a novel approach that extends the work of d'Ascoli et al. [2022] and Biggio et al. [2021] with the following contributions:

- Effective utilization of powerful and scalable transformer [Vaswani et al., 2017] architecture for the symbolic regression.
- Use pre-training on a large number of equations to bias the search.
- Jointly predicts both the symbolic representation and the coefficients, which makes the decoder more informed.
- We utilize a local gradient search to further improve our results.
- We empirically show, that our model is competitive with current state-of-the-art results while outperforming them in the required time to find the expression.
- We propose a new way of coefficient representation during the training and inference, which empirically helps the model.

1. Background

This chapter introduces the reader to the most critical concepts of symbolic regression and deep learning that are later used in the rest of this thesis.

1.1 Symbolic regression

Formally we can define the symbolic regression as: "Given set of n input-output pairs $\{(x_i, y_i)\}_{i=1}^n \subseteq X \times Y$, where $x_i = (x_{i_1}, \ldots, x_{i_k})$ and $y_i \in \mathbb{R}^k$, we want to find an equation e and corresponding function f_e, such that $y \approx f_e(x)$." [Biggio et al., 2021]. In general setting, the function f can be recurrent function as used by d'Ascoli et al. [2022] or even differential equation. Symbolic regression is typical example of machine learning on sets, since in general, the input to the model has no natural order. Due to this limitation, we need to use special architectures or tools to work with them.

Practically speaking, we are trying to find a function definition that for given inputs generates given outputs. Usually, we are constrained on some closed interval $x_{i_k} \in [a, b]$ and therefore, there exist many such functions which meet this condition. An example of such set with input-output pairs sampled from $[0, 5]$ could be points $\{(1, 1), (2, 2), (3.5, 3.5), (4.1, 4.1)\}$ and desired function $f(x) = x$. The goal of symbolic regression is to find the function $f(x) = |x|$, but the program could also find function $f(x) = \sqrt{x^2}$, which would have zero error on the interval $[0, 5]$ or functions $\sin(x)$ and $\cos(x - \frac{\pi}{2})$, which would be equally good. However, usually, we are looking for a function that is in some sense the simplest one, e. g. in the number of operators.

The complexity of symbolic regression spans mainly from the vast amount of possible functions that can be generated. As shown in Lample and Charton [2019], the amount of possible functions grows exponentially with the number of operators.

1.2 Function representation

In the future, we will need a way to represent functions as a sequence of symbols so they can be predicted using a neural network. We will therefore look closely on what are the possibilities. In general, an expression can be considered a tree, where internal nodes are operators (addition, sin, . . .) and leaves (variables and numbers) are operands. Each of these operators has an arity that tells us how many arguments it expects (how many children the node has). For example, abs, denoting absolute value, has an arity of one since it expects only one argument, on the other hand, operator pow, denoting power, has an arity of two. All the elementary functions in mathematics are at maximum binary, and therefore the expression forms a unary-binary tree. Note, that the trees of semantically equivalent expressions can be different due to commutativity or associativity. E.g. function $x + y = y + x$, $x \cdot y = y \cdot x$ or even $\sin x = \sin(x + 2\pi)$, whose tree of the left expression is different then the tree of the right expression. Since each expression can be represented as a tree, we have three different ways to traverse

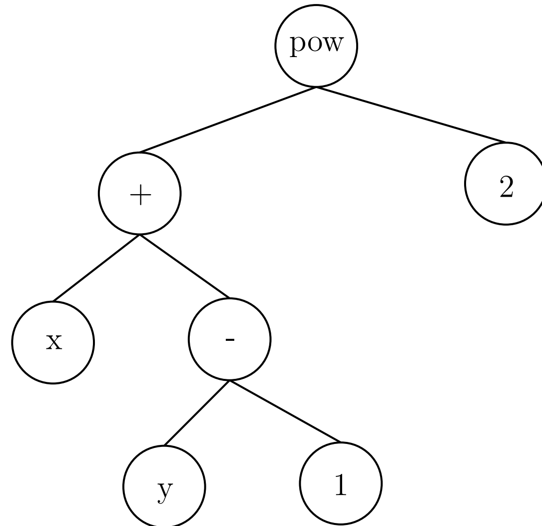


Figure 1.1: Example of an expression tree.

it.

- Preorder (prefix) representation
- Inorder (infix) representation means
- Postorder (postfix) representation

For example, if we have an expression $(x + y - 1)^2$ and its tree is the same as in Figure 1.1. The inorder representation would be the same, and the preorder would be as ”pow + x - y 1 2” and postorder ”x y + 1 - 2 pow”. The advantage of using preorder or postorder representation is that we do not need parentheses and that the expressions can be easily evaluated [Gabbrielli and Martini, 2010].

1.3 Encoder-decoder architecture

Encoder-decoder is a general type of neural network architecture for handling sequence to sequence problems. As the name suggests, the architecture consists of an encoder and a decoder. The function of the encoder is to take an input sequence and convert it into a dense representation. This representation can be thought of as a representation of the whole input sequence. The decoder then takes this representation and generates the output sequence [Arumugam and Shanmugamani, 2018].

The encoder-decoder architecture is mainly known from NLP, where it is used, for example, in the case of machine translation [Vaswani et al., 2017] or text summarization [Khandelwal et al., 2019]. In the setting of symbolic regression, the encoder takes a set of points, and the decoder outputs a sequence of tokens representing the function given the dense representation and previous outputs.

1.3.1 Recurrent neural networks

Recurrent neural networks (RNNs) are a family of neural networks for processing sequential data. The idea behind RNN is parameter sharing, making it possible to

extend and apply the model to examples of different lengths and generalize across them. Such sharing is essential when a specific piece of information can occur at multiple positions within the sequence. For instance, in the case of intent classification, the model should have the same output for "I want to travel to Germany next year." and "Next year, I want to travel to Germany." [Goodfellow et al., 2016].

The usual way to think about RNNs is as unfolding computation graph. Goodfellow et al. [2016] defines RNN as a function f parameterized with Θ which takes previous state $h \in R^m$ and input $x = (x_1, x_2, ..., x_k)$:

$$h^{(t)} = f(h^{(t-1)}, x_t; \Theta), \tag{1.1}$$

where $h^{(t)}$ is called the hidden state. Lets assume that we are at time step $t = 3$, then we can unfold the previous equation and obtain:

$$h^{(3)} = f(h^{(2)}, x_2; \Theta) = f(f(h^{(1)}, x_1; \Theta), x_2; \Theta) \tag{1.2}$$

Note that the same parameters Θ are shared across all timesteps Goodfellow et al. [2016].

1.3.2 Transformer

The Transformer is encoder-decoder network architecture, where the encoder maps input sequence of symbols $(x_1, \dots, x_n)$ to a sequence of continuous representations $z = (z_1, \dots, z_n)$, then given z, the decoder generates output sequence $(y_1, y_2, \dots, y_m)$ one token at the time [Vaswani et al., 2017]. The probability of next token y_k is given as $\mathrm{P}(y_k | y_{k-1}, y_{k-2}, \dots, y_0, z)$. The goal of the transformer model was to overcome some of the issues that the previous architectures had. One of the main problems that RNN has is its inability to properly parallelize the training due to the need of the previous hidden state to compute the next state. The second issue is that the RNNs at step n need to remember k previous steps to retrieve information at position $n - k$ [Vaswani et al., 2017]. These liabilities are not present in Transformer due to the Multihead attention mechanism, which on the other side, has its issues, such as its computational cost, which is quadratic with the length of the sequence [Vaswani et al., 2017]. This issue will, however be addressed later. For the complete architecture of the transformer model, see Figure 1.2.

Attention

An attention function can be described as the mapping of a query and a set of key-value pairs to an output. This output is then computed as a weighted sum of the values, where the weight represents the compatibility of the query and corresponding key [Vaswani et al., 2017].

In case of transformer, the "Scaled Dot-Product Attention" is used. Let us denote concatenated queries as $Q \in \mathbb{R}^{n \times d_q}$, concatenated keys $K \in \mathbb{R}^{n \times d_q}$ and concatenated values $V \in \mathbb{R}^{n \times d_v}$. The attention matrix is then computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V, \tag{1.3}$$

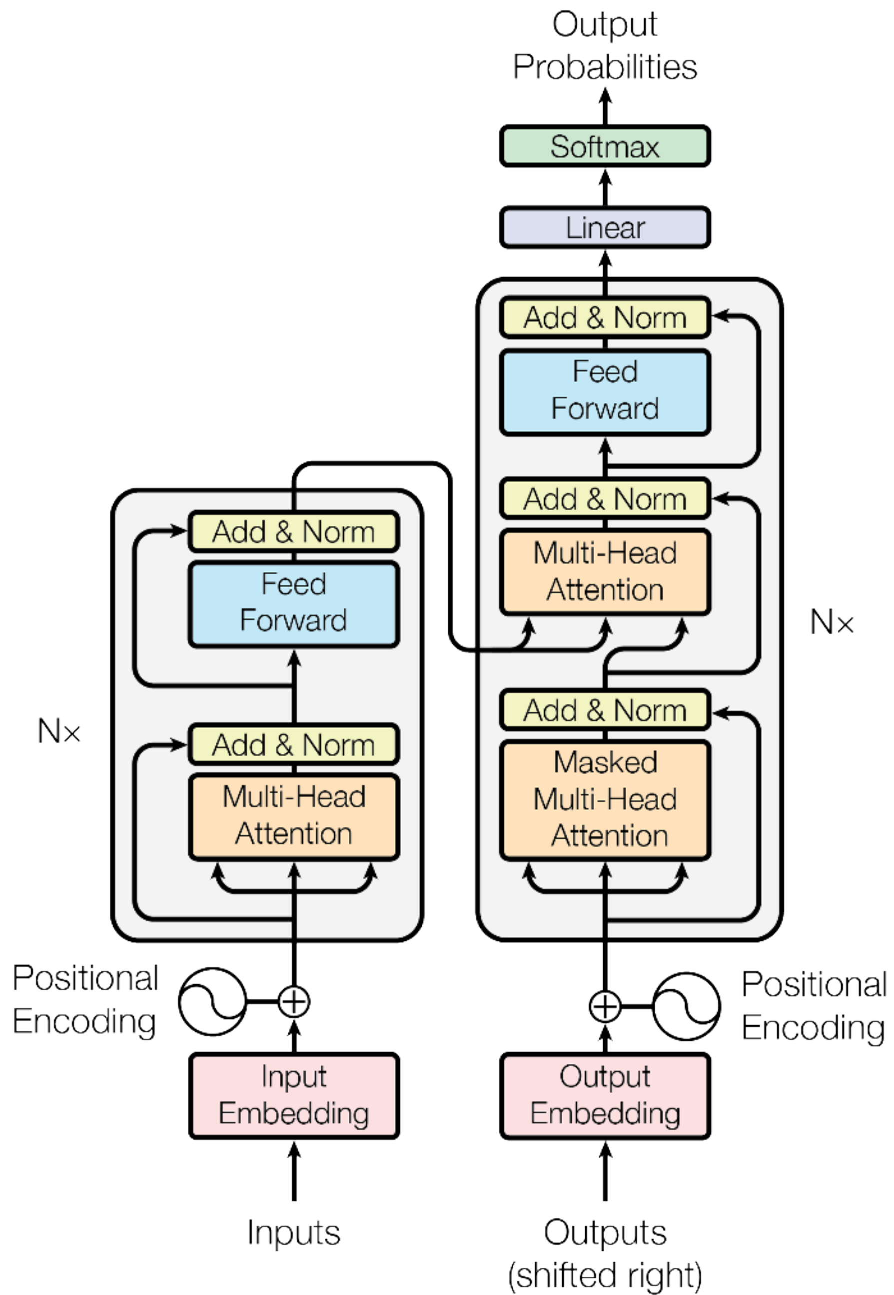


Figure 1.2: Transformer model architecture. Source: Vaswani et al. [2017]

where d_k is the dimension of the model [Vaswani et al., 2017].

This computation is identical to the dot-product attention except for the $\frac{1}{\sqrt{d_k}}$ factor. The Vaswani et al. [2017] notes, that the scaling factor is used because of the large values of dot product, which push the softmax into regions where the gradient is small. To illustrate the issue, Vaswani et al. [2017] uses this example. Assume that components of q and k are independent random variables with mean 0 and variance 1, then their dot product $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$.

$$\begin{aligned}\operatorname{var}(q \cdot k) &= \operatorname{var}(\sum_{i=1}^{d_k} q_i k_i) \overset{indep.}{=} \sum_{i=1}^{d_k} \operatorname{var}(q_i k_i) = \\ &= \sum_{i=1}^{d_k} (\operatorname{var}(q_i) \operatorname{var}(k_i) + \operatorname{var}(q_i)(\mathrm{E}(k_i))^2 + \operatorname{var}(k_i)(\mathrm{E}(q_i))^2 = \\ &= \sum_{i=1}^{d_k} \operatorname{var}(q_i) \operatorname{var}(k_i) = d_k\end{aligned} \tag{1.4}$$

Instead of performing a single attention function with d_{model}-dimensional keys, values, and queries, Vaswani et al. [2017] found beneficial to project the queries, keys, and values h times with learned linear projections. Then on each of these projections, we can compute the attention and concatenate the output values, which are projected once again. Multi-Head attention can therefore be defined as:

$$\begin{aligned}\mathrm{MultiHead}(Q, K, V) &= \mathrm{Concat}(\mathrm{head}_1, \ldots \mathrm{head}_h) W^O, \\ \mathrm{head}_i &= \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V),\end{aligned} \tag{1.5}$$

where $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$. The computation of the Multi-Head Attention, can be seen on Figure 1.3.

One of the issues with the Multi-Head attention is its time and memory complexity. Since we need to compute matrix multiplication, the time complexity is $O(n^2 \cdot d)$, where n is sequence length, and d is the dimension of the representation. The memory complexity is also quadratic $O(n^2)$ since we need to store these matrices during the computation [Vaswani et al., 2017]. There were several attempts to lower the time and memory complexity of Multi-Head attention, such as Wang et al. [2020], Kitaev et al. [2020], but we will later describe a different approach that reduces the computational complexity and therefore is more suitable for models with larger inputs.

Positional encoding

Since there are no recurrences and no convolutions in the transformer, the model does not have information about the relative or absolute position of the tokens in the sequence. To tackle this issue, Vaswani et al. [2017] introduces "positional encoding". The positional encodings have the same dimension as d_{model} and are summed with the token representation. There are two possible choices of positional encodings. The first one is learned, which is more flexible encoding, however, it adds parameters to the model. The second option is to use fixed embeddings, which are given by some equation. Vaswani et al. [2017] propose these embeddings as:

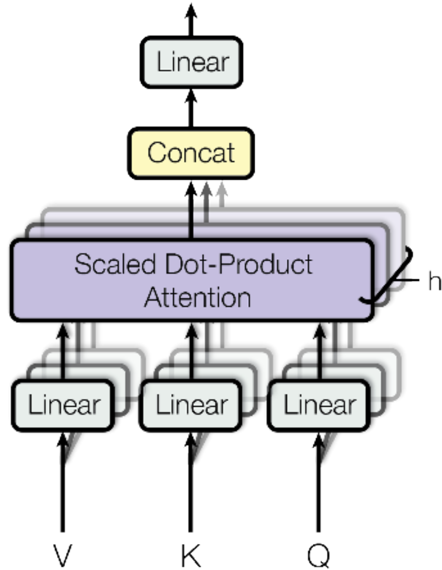


Figure 1.3: Computation of Multi-Head attention. Source: Vaswani et al. [2017]
.

$$\mathrm{PE}(\mathrm{pos}, 2i) = \sin\left(\frac{\mathrm{pos}}{10000^{2i/d_{model}}}\right) \tag{1.6}$$

$$\mathrm{PE}(\mathrm{pos}, 2i+1) = \cos\left(\frac{\mathrm{pos}}{10000^{2i/d_{model}}}\right) \tag{1.7}$$

1.3.3 Decoding

Since encoder-decoder-based models usually work by predicting probability distribution over the next token once for each time step, we need a way to decode the most probable sequence. The most straightforward way is to take the most probable token at each time step. This method of decoding is called greedy search [Kamath et al., 2019]. The issue with greedy decoding is it may not yield the most probable sequence overall, since we are not considering any other tokens during the decoding. To solve this issue, we can introduce beam search. The idea behind beam search with beam width k is to keep the k most probable sequences we have encountered before the current time step. This allows us to decode sequences with a small probability in the beginning but high in the end. The exact algorithm can be seen in the Algorithm 1.

Another way how to select the most probable sequence is to use random sampling. The idea behind random sampling is that in each time step, we sample from the predicted distribution. Furthermore, there are several extensions of this algorithm, namely Temperature sampling [Ackley et al., 1985], in which we alter the distribution by defining the new probability as:

$$f(p;\tau)_i = \frac{p_i^{\frac{1}{\tau}}}{\sum_j p_j^{\frac{1}{\tau}}}, \tag{1.8}$$

Data: $\hat{\mathbf{y}}$, beam_width, T
Result: y with highest $P(y)$
$R_0 \leftarrow \{(\langle \text{SOS} \rangle)\}$;
$P_0 \leftarrow \{0\}$;
for *t in 1 to T* **do**
  for *h in* R_{t-1} **do**
    for $\hat{y} \in Y$ **do**
      $\hat{\mathbf{y}} = (y_1^h, \dots, y_{t-1}^h, \hat{y})$;
      $R_t += \hat{\mathbf{y}}$;
      $P_t += \log_2 P(\hat{\mathbf{y}})$;
    end
    $R_t \leftarrow$ select beam_width beams from R_t according to P_t;
    $P_t \leftarrow$ select highest beam_width items from P_t;
  end
end
return The most probable R_T based on P_T.

Algorithm 1: Beam search. Source: [Kamath et al., 2019]

Data: $\hat{\mathbf{y}}$, n_beams, T
Result: y with highest $P(y)$
$R \leftarrow [(\langle \text{SOS} \rangle)] \times$n_beams;
$P \leftarrow [0] \times$n_beams;
for *t in 1 to T* **do**
  preds, probs $\leftarrow$ predict(R);
  next_tokens, probs $\leftarrow$ sample(preds, probs);
  $P += \log_2(\text{probs})$;
  $R \leftarrow$ concatenate(R, next_tokens);
end
return The most probable R based on P.

Algorithm 2: Random sampling.

where the lower the temperature τ is, the less diverse the results. Lately, there were proposed several extensions of random sampling (popular mostly in NLP). Namely Top-K sampling [Fan et al., 2018], where only the top k most probable tokens are considered for sampling (while redistributing the previous probability mass), or Top-P sampling [Holtzman et al., 2019], also known as nucleus sampling, where we sample from the smallest set of tokens with cumulative probability larger than some threshold p (while also redistributing the probability mass in the end). The pseudocode of random sampling can be seen in the Algorithm 2, where predict($\cdot$) function predicts the probability distribution of the next token for each of the inputs, sample($\cdot,\cdot$) is one of the methods of sampling (random, temperature, Top-K, or Top-P), and concatenate($\cdot,\cdot$) function concatenates the sampled tokens with previous tokens.

1.4 Deep learning with sets as input

Sets are a natural way of representing many machine learning problems. These problems usually consist of some kind of input data that has no natural order. For example, such problems are multi-instance problems, where the input data are sets of examples [Chevaleyre and Zucker, 2001]. One of the examples of a multi-instance problem is classifying aromatic molecules according to whether or not they are "musky" [Dietterich et al., 1997]. Another typical problem is set classification, e.g. 3D shape classification [Wu et al., 2014a] or set to sequence problems such as symbolic regression. There are generally two conditions that the method should satisfy to solve these problems. The first one is permutation invariance:

Property 1. A function $f : 2^{\mathcal{X}} \to Y$, acting on sets must be permutation invariant to the order of objects in the set, i.e. for any permutation π:

$$f(\{x_1, x_2, \ldots, x_n\}) = f(\{x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)}\}), \tag{1.9}$$

which means that the output of the function f is same for $\{1, 2, 3\}$ and $\{2, 3, 1\}$ [Zaheer et al., 2017]. Note that $2^{\mathcal{X}}$ denotes powerset. However, a more common setting is when each set element has an associated label, the objective would be to learn a permutation equivariant function.

Property 2. A function $f : \mathbb{X}^M \to \mathbb{Y}^M$ acting on sets must be permutation equivariant, when function upon permutation of input instances permutes output labels, i.e. for any permutation π:

$$f([x_{\pi(1)}, x_{\pi(2)}, \ldots x_{\pi(n)}]) = [f_{\pi(1)}(x), f_{\pi(2)}(x), \ldots, f_{\pi(n)}(x)], \tag{1.10}$$

which just means that if the input to function f is $[1, 2, 3]$ and the output is $[2, 4, 6]$, then for input $[2, 3, 1]$ the output should be $[4, 6, 2]$ [Zaheer et al., 2017].

The second desirable condition is the ability to handle variable-size inputs since the size of the input sets can vary in size [Lee et al., 2018]. In recent years, many methods were developed that meet these conditions, and some of them will be described in the following sections.

1.4.1 Deep sets

Deep sets is a general framework for solving problems containing sets as inputs using deep learning. Zaheer et al. [2017] prove that:

Theorem 1. A function $f(X)$ operating on a set X having elements from countable universe is a valid set function, i. e. invariant to the permutation of instances in X, iff it can be decomposed in the form $\rho(\sum_{x \in X} \phi(x))$ for suitable transformation ρ and ϕ.

Next, they analyze the equivariant case when $\mathbb{X} = \mathbb{Y} = \mathbb{R}$ and $f_\Theta(x) = \sigma(\Theta X)$ is neural network parameterized by Θ and σ is non-linear function.

Theorem 2. The function $f_\Theta : \mathbb{R}^M \to \mathbb{R}^M$ (as defined above) is permutation equivariant iff all the off-diagonal elements of Θ are tied together, and all the diagonal elements are equal as well. That is

$$\Theta = \lambda \mathbf{I} + \gamma(\mathbf{1}\mathbf{1}^T), \tag{1.11}$$

where $\lambda, \gamma \in \mathbb{R}$, $\mathbf{1} = [1, \ldots, 1]^T \in \mathbb{R}^M$ and $\mathbf{I} \in \mathbb{R}^{M \times M}$ is identity matrix [Zaheer et al., 2017].

Based on the Theorem 2, the function is permutation equivariant iff $f_\Theta(x) = \sigma(\lambda \mathbf{I}x + \gamma(\mathbf{1}\mathbf{1}^T)x)$. This is simply combination of its input $\mathbf{I}x$ and sum of the input values $(\mathbf{1}\mathbf{1}^T)x$. Since the summation is permutation invariant, the layer is also permutation invariant. Zaheer et al. [2017] also further manipulate the operations and parameters to get other variant, e.g.:

$$f(x) = \sigma(\lambda \mathbf{I}x + \gamma \cdot \text{maxpool}(x)\mathbf{1}) \tag{1.12}$$

This function can be interpreted as a combination of feed-forward layers with pooling operation in the end. Zaheer et al. [2017] calls this neural network as DeepSets and applies it to several problems. The general framework that they propose is to first pass each input element x_i through the same, possibly multi-layer, feed-forward neural network to get the embedded representation $\phi(x_i)$. The second step is to use set invariant operation pool$(\cdot)$ (sum, mean, max, ...) over the feature dimensions to get set representation and then pass it through several feed-forward layers ρ [Jurewicz and Strømberg-Derczynski, 2021].

$$\text{DeepSets}(\{x_1, x_2, \ldots, x_n\}) = \rho(\text{pool}(\{\phi(x_1), \phi(x_2), \ldots, \phi(x_n)\}) \tag{1.13}$$

Even though this method introduces a simple way to handle sets in deep learning and proof of the sufficiency of this method, several concerns were raised. Jurewicz and Strømberg-Derczynski [2021] argues that summation prevents the model from learning pair-wise and higher-order interactions between the elements of the set. Wagstaff et al. [2019] notes that the proof of Property 1 presented in Zaheer et al. [2017] considers only functions on countable domains, which could in practice limit the practical value of the work.

1.4.2 Set-Transformer

Set-Transformer is transformer architecture, which can process input data which consists of sets. There were two goals when proposing this architecture:

- Use powerful transformer architecture to handle sets.
- Since sets can be quite large and Multi-Head Attention scales with $O(n^2)$ then to lower the time complexity without much loss of power.

Lee et al. [2018] builds upon the foundation of Zaheer et al. [2017] and notices that Equation 1.13 can be deconstructed into two parts. The first is an encoder $\phi(\cdot)$ which acts on each element of a set and decoder $\rho(\text{pool}(\cdot))$ which aggregates these encoded features and produces desired output. To achieve these goals above, Lee et al. [2018] introduces new layers.

Permutation Equivariant (Induced) Set Attention Block

Most previous methods used to project each instance of the set independently of each other, which forbid them to model interactions among the instances. Lee et al. [2018] therefore uses Multi-Head attention [Vaswani et al., 2017] from the

transformer to encode the whole set. This allows the Set Transformer to compute pairwise as well as high-order interactions among the instances. Lee et al. [2018] defines Multi-head Attention block (MAB) as:

Definition 1 (Multi-head Attention Block). Let $X, Y \in \mathbb{R}^{n \times d}$ represents two sets of d-dimensional vectors of cardinality n. Then Multi-head Attention Block is defined as:

$$\mathrm{MAB}(X, Y) = \mathrm{LayerNorm}(H + \mathrm{rFF}(H)), \tag{1.14}$$

where $H = \mathrm{LayerNorm}(X + \mathrm{MultiHead}(X, Y, Y))$.

LayerNorm is Layer Normalization as defined by Ba et al. [2016], MultiHead is Multi-Head Attention as defined in Equation 1.5 and rFF indicates row-wise feed-forward layer, which processes each instance independently and identically. After the definition of MAB, we can define the Set Attention block (SAB) [Lee et al., 2018].

Definition 2 (Set Attention Block). Let $X \in \mathbb{R}^{d \times n}$ be a matrix representing the set of d-dimensional vectors and the cardinality of the set is n, then SAB is defined as

$$\mathrm{SAB} := \mathrm{MAB}(X, X). \tag{1.15}$$

The SAB block takes a set as input and performs a self-attention between the elements in the set, resulting in a set of the same size. We can then stack these blocks to create an encoder.

One of the potential issues of the SAB layer is the quadratic time complexity $O(n^2)$ due to the use of Multi-Head Attention. This becomes costly for large sets e. g. for 3D point clouds, where the number of points in the set can be in thousands [Wu et al., 2014b], which becomes computationally infeasible. To tackle the issue, Lee et al. [2018] introduces Induced Set Attention Block.

Definition 3 (Induced Set Attention Block). Let $X \in \mathbb{R}^{n \times d}$ be a set and $I \in \mathbb{R}^{m \times d}$ is m d-dimensional matrix of learnable parameters called inducing points, then ISAB is defined as

$$\begin{aligned} \mathrm{ISAB}_m(X) &= \mathrm{MAB}(X, H), \\ H &= \mathrm{MAB}(I, X). \end{aligned} \tag{1.16}$$

The ISAB first transforms inducing points I into H by attending to the input set. Then this resulting set is again attended to the input set X, which produces a set of n elements. This has an effect, that the attention is computed between the set of size m and n, which lowers the time complexity of ISAB_m layer to $O(n \cdot m)$, where m is a hyperparameter. Note that both SAB and ISAB are permutation equivariant due to the usage of Multi-Head Attention, rFF, and LayerNorm, which are all permutation equivariant.

Pooling by Multi-Head attention

A usual way to aggregate representations of each instance of the input set was to use pooling operations such as a feature dimension-wise sum. However, one of the concerns is that the high order interactions between individual instances are lost [Jurewicz and Strømberg-Derczynski, 2021]. Lee et al. [2018] therefore

proposes a different kind of pooling which is called Pooling by Multi-Head attention (PMA). The purpose of this method is to aggregate feature vectors by applying Multi-Head attention with a set of k learnable vectors $S \in \mathbb{R}^{k \times d}$ called seed vectors.

Definition 4 (PMA_k). Let $Z \in \mathbb{R}^{n \times d}$ be set of features from an encoder and $S \in \mathbb{R}^{k \times d}$ are learnable parameters. Then:

$$\text{PMA}_k(Z) = \text{MAB}(S, \text{rFF}(Z)). \tag{1.17}$$

Note that the output of PMA_k is a set of k items, which is another hyper-parameter. The value of k is dependent on the task, however Lee et al. [2018] usually use $k = 1$. Different values of $k \neq 1$ can be used, for example, in the case of clustering, where the k is dependent on the number of clusters. Lee et al. [2018] also further models the interactions between the k outputs by applying SAB afterward.

$$H = \text{SAB}(\text{PMA}_k(Z)), \tag{1.18}$$

which they show empirically to work better.

Final architecture

Using the layers above, Lee et al. [2018] defines the final model architecture as:

$$\begin{aligned} \text{Encoder}(X) &= \text{SAB}(\text{SAB}(X)) \\ \text{Decoder}(Z) &= \text{rFF}(\text{SAB}(\text{PMA}_k(Z))). \end{aligned} \tag{1.19}$$

Is it also possible (depending on the underlining dataset and task) to change the SAB layers to ISAB_k to lower the computational cost. Lee et al. [2018] also proves that the whole transformer is permutation equivariant since all the building blocks are permutation equivariant.

2. Related work

The overall goal of this chapter is to describe related work. In history, the most used methods were based on genetic programming, such as Koza [1992], which is since then one of the main branches of the symbolic regression. However, with the increasing popularity of neural networks, new approaches emerged in recent years. Most of these methods are based on recurrent neural networks [Petersen, 2019, Mundhenk et al., 2021] or transformer architecture [Valipour et al., 2021, Biggio et al., 2021, d'Ascoli et al., 2022], however, several novel approaches, such as Equation Learner, [Martius and Lampert, 2016, Werner et al., 2021, Sahoo et al., 2018] were developed.

2.1 Genetic programming

A natural way of solving symbolic regression is by using Genetic programming. The way how Koza [1992] solve the symbolic regression is to encode the expression as a tree. First, they create a set of possible operands and operators, then generate a population of random functions, and start the evolution process involving selection, crossover, and mutation. The fitness function is defined as the mean squared error between the function and the correct value from input points. Poli et al. [2008] describes the usage of genetic programming for symbolic regression in these steps.

2.1.1 Initialization

The evolution starts with creating k random expressions called population. Each of these individuals is a binary-unary tree representing expression. The example population can be seen in Figure 2.1. The first tree represents $x + y - 1 + 2 = x + y + 1$, the second tree represents $x + 1 + 2 = x + 3$ and last one is $1 - 2 = -1$. The Poli et al. [2008] does not mention any simplification procedure, but it could be possibly used.

2.1.2 Fitness evaluation

As a fitness function, Poli et al. [2008] uses the sum of absolute errors, however, different fitness functions such as RMSE [Rad et al., 2018] or R^2 [Glantz and Slinker, 2000] are possible. Chicco et al. [2021] notes that R^2 is more informative than MSE or MAE for regression analysis, and therefore R^2 should also be a good choice for the fitness function.

2.1.3 Selection, Mutation, Crossover

An important point, which Poli et al. [2008] raises, is to use selection that is not greedy, meaning that even the individuals who have large fitness value (in case of minimization) are selected with some probability. These types of selections are called Fitness proportionate selections. An example of such selection is roulette

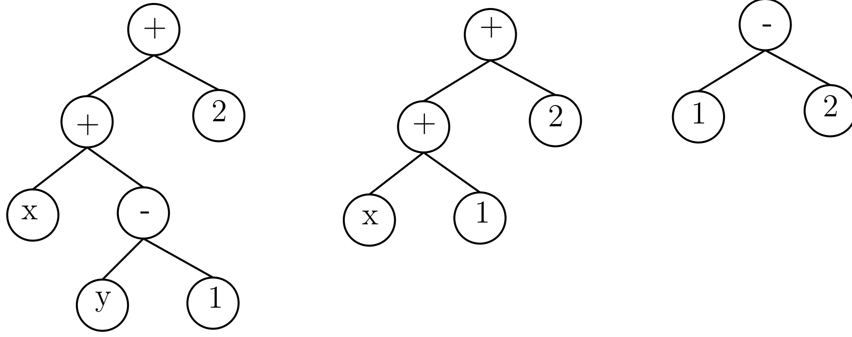


Figure 2.1: Initial population

wheel selection [Blickle and Thiele, 1996]. The reason for selecting a sub-optimal solution is to escape local minima.

Next operation that Poli et al. [2008] uses is mutation. During mutation, random individuals are selected, and the subtree of a random node in the tree is replaced with a random tree generated in the same manner as the individual was created.

The final operation is the crossover, in which individuals are selected to create offspring. First, two random individuals are selected, and an arbitrary node is chosen in each tree. Then the algorithm swaps the subtrees spanning from these selected nodes to create two new trees. The algorithm is stopped once the fitness function is small enough (in the case of minimization).

2.2 Equation Learner

An interesting approach is presented by Martius and Lampert [2016]. Instead of training the neural network to predict the desired function as its output, the authors take a different approach. Lets $\{(x_i, y_i)\}_{i=1}^{n} \subseteq X \times Y$, where $x_i = (x_{i_1}, \dots, x_{i_k})$ and $y_i \in \mathbb{R}$ be one input instance. They train a feed-forward neural network to predict y_i based on the inputs x_i. However, instead of using typical non-linear activation functions such as ReLU [Agarap, 2018], they use elementary functions (sin, cos, identity, multiplication, ...) as activation functions. Then they stack L of these layers to create a neural network with parameters $\theta = \{W^{(1)}, \dots, W^{(L)}, b^{(1)} \dots, b^{(L)}\}$. To get the final function, Martius and Lampert [2016] wait up until the network converges, and then they read the weights from the network and its activations. However, this representation would be dense since most network weights would be non-zero. Therefore to get the sparse function representation, they use a L1 penalty on the model weights, which pushes them to 0. Therefore, the final loss function L consists of MSE between the predictions $\phi(x_i)$ and ground truth y_i and L1 penalty on the network weights (excluding biases).

$$L(D) = \frac{1}{N} \sum_{i=1}^{|D|} \|\phi(x_i) - y_i\|^2 + \lambda \sum_{l=1}^{L} \|W^{(l)}\|_1 \tag{2.1}$$

The benefit of using this method is that we can select different kinds of networks based on each input type, for example, if we have some insight into how

complex the function is, we can use a different number of layers or select different activations in each layer. However, this approach also has some disadvantages. First, the network needs to be trained from scratch for each equation, which can take some time. Second, it is possible that the final function representation can get dense due to the number of weights in the network (many of them will be probably close to zero), and lastly, there are some issues with functions domains, e.g., there do not exist clear way how to handle $\ln \cdot$ or $\sqrt{\cdot}$ for negative values. This model would also have a significant problem in retrieving exact exponents since, once again, $x^a, a \in \mathbb{R}$ can generate complex values.

However, recently there were several approaches which improves the original Equation learner idea [Werner et al., 2021, Sahoo et al., 2018] by fixing the issue with the function domains. For example Werner et al. [2021] introduces solution to domain problem of $\ln \cdot$ or $\sqrt{\cdot}$ which involves shifting input $\hat{z} = z + \alpha$ by learnable positive relaxation parameter $\alpha = \log\left(1 + e^{\hat{\alpha}}\right) > 0$

$$\hat{f}(\hat{z}) = \begin{cases} f(z+\alpha), & \text{for } z > a, \\ 0, & \text{otherwise} \end{cases}$$

2.3 Methods based on reinforcement learning

In recent years, new neural network-based approaches emerged. The idea is to leverage the knowledge and power of neural networks to predict the expression. One of the first methods was introduced by Petersen [2019]. The idea is to train Recurrent Neural Network for each of the input equations. The goal of the network then predicts the given equation. To calculate the loss between the ground truth function and the function predicted by the RNN, they use Normalized root mean square error between the predicted values from the predicted function and the groundtruth function. NRMSE is defined as $\text{NRMSE} = \frac{1}{\sigma_y}\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{f}(X_i))^2}$, where σ_y is standard deviation of y, y_i is the ground truth, $\hat{f}$ is the predicted function and X_i is one the function inputs. This loss is however not differentiable due to usage of predicted $\hat{f}$, therefore Petersen [2019] uses reinforcement learning to train the network with reward function defined as $R(\tau) = \frac{1}{1+\text{RMSE}}$. The process of generating the same expression tree as in the Figure 1.1 can be seen in the Figure 2.2. To exploit the problem's hierarchical nature, the author uses the parent in the expression tree and its sibling if its parent binary operation as the input to the network. This helps the model since it does not need to remember these important nodes.

2.3.1 Training

The training is then done using NRMSE as the reward function using the Risk-seeking policy gradient. The reason for using the Risk-seeking policy gradient is that the standard REINFORCE policy gradient [Williams, 1992]:

$$\nabla_\theta J_{\text{std}}(\theta) = \nabla_\theta \mathbb{E}_{\tau\sim p(\tau|\theta)}[R(\tau)] = \mathbb{E}_{\tau\sim p(\tau|\theta)}[R(\tau)\nabla_\theta \log p(\tau|\theta)], \quad (2.2)$$

is defined as an expectation that is often desired for control problems, where one seeks to optimize the average performance of the policy [Petersen, 2019]. However,

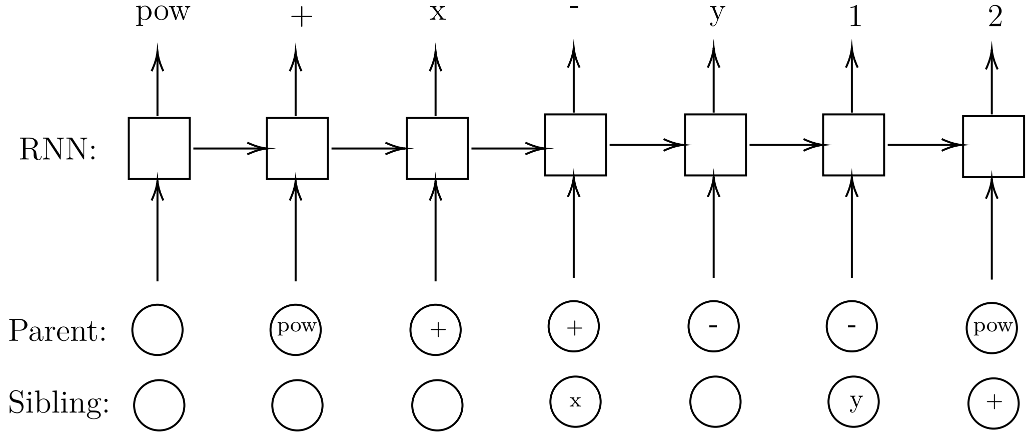


Figure 2.2: The architecture of deep symbolic regression. Inspired by Petersen [2019]

.

in the case of symbolic regression or program synthesis, the final performance is measured by a single sample found during training. For such cases $J_{\text{std}(\theta)}$ is not an appropriate objective. Therefore Petersen [2019] proposes a new objective, which focuses on maximizing best-case performance. First, they define $R_\epsilon(\theta)$ as the $(1-\epsilon)$-quantile of the distribution of rewards under the current policy. This is, however, not tractable, and therefore they propose a new learning objective $J_{\text{risk}}(\theta;\epsilon)$, parametrized by ϵ

$$J_{\text{risk}}(\theta;\epsilon) = \mathbb{E}_{\tau\sim p(\tau|\theta)}[R(\tau)|R(\tau) \geq R_\epsilon(\theta)]. \tag{2.3}$$

This learning objective aims to increase best-case performance at the expense of lower worst-case and average performance. During the training Petersen [2019] also outputs C as part of the symbolic output, which indicates coefficient. Since the model does not output the exact values of these coefficients, they need to be found in another way. Petersen [2019] uses global optimization BFGS [Fletcher, 1987] to find them, which in effect slows down the training. They report that finding an equation from the Nguyen benchmark [Uy et al., 2011] (which will be shown later) takes 1920.7 ± 342.9 s without early stopping and 483 ± 641.9 s with early stopping on Nvidia P100.

Petersen [2019] also shows an easy way how to constrain the search space, which can be easily used in any sequence to sequence model which outputs probability distribution. During the inference, it is possible to zero the probabilities of an unwanted sequence of tokens. E.g. we do not want to have an operator followed by its inverse $(\log(\exp(x)))$ or nested trigonometric function $(\sin(1+\cos(x)))$, which is not very common in real-world problems.

2.3.2 Hybrid approach

An interesting approach that has been proposed recently is to use Recurrent neural networks and Genetic programming jointly. Mundhenk et al. [2021] uses a method that first generates N samples from the RNN and then uses these samples to initialize the population for the genetic programming algorithm. Then it runs the genetic algorithm several times and selects the best samples from both RNN

and the genetic algorithm. Lastly, it uses these found examples to calculate the reward function to train the RNN and continues with the first step, which once again generates N samples from RNN. This algorithm continues up until the prediction is good enough. This approach has empirically better results in comparison to Petersen [2019], however it can be slower in some cases.

2.4 Transformer based approaches

Since the invention of the Multi-Head Attention mechanism [Vaswani et al., 2017], the transformer architecture has been used across all the machine learning subfields. Symbolic regression is no exception in this regard, and therefore we will describe approaches that use transformed-based architectures.

A critical distinction from previous methods is that the transformer-based approaches are trained in a supervised manner, so they need to receive pairs of points and expressions. In contrast, Genetic programming and Reinforcement based methods focus on guiding the search, and therefore no training data are necessary.

The full workflow usually looks like this. The first step is to generate data. There are several approaches how to generate random expressions, which will be described later, however, for now, let's assume that we have generated k random expressions. Next, we need to sample points from these equations. Some authors choose to sample n points uniformly such as Biggio et al. [2021] or equidistantly such as d'Ascoli et al. [2022]. The next step is to encode the generated expression as a sequence of tokens. In most cases, each method has its unique way of handling the coefficients. The different ways of encoding and handling the coefficients will be described later. Finally, the model is trained using cross-entropy loss and evaluated.

2.4.1 Data generation

There are several ways how to generate random expressions, such as modeling an expression as output from probabilistic context-free grammar [Brence et al., 2020] or generating a random expression tree [Valipour et al., 2021, Lample and Charton, 2019]. Each of these methods starts with defining the operators (e.g., sin, cos, ...), which are fixed before the training, and the operands (variables, possible integers, and floats) and assigning them some probability. Usually, the randomly generated expressions are not in the most simple form (e.g., $1 + 1 - 1$ can be generated), and therefore most methods use postprocessing to simplify the generated expression using SymPy [Meurer et al., 2017]. This, however prolongs the time to generate one expression since the simplification can be costly. Interestingly d'Ascoli et al. [2022] empirically showed that skipping the expression simplification does increase the training error, but the error on the validation set stays the same.

The most straightforward way how to generate a random expression tree is to recursively generate nodes while deciding what kind of node it is by randomly choosing the arity of the node. If it is terminal, therefore containing constant or variable, or if it is an operator such as sin or addition. The problem with this approach is that it favors deep trees, and therefore not all trees are equiprobable

[Lample and Charton, 2019]. To solve this issue Lample and Charton [2019] derives an algorithm to generate a random unary-binary tree with n internal nodes. First, they generate the tree, filling in the operators and leaves. The pseudocode can be seen in Algorithm 3, where $L(e, n) = (k, a)$ is the probability that the next internal node is in position k with arity a.

```
Start with an empty node, set e = 1;
while n > 0 do
    Sample a position k and arity a from L(e, n) (if a = 1 the next
      internal node is unary);
    Sample the k next empty nodes as leaves;
    if a = 1 then
        Sample a unary operator;
        Create one empty child;
        Set e = e − k;
    end
    else
        Sample a binary operator;
        Create two empty children;
        Set e = e − k + 1;
    end
    Set n = n − 1;
end
```

Algorithm 3: Random unary-binary tree generation. Source: Lample and Charton [2019]

A different approach is taken by Valipour et al. [2021], where they parametrize the algorithm by the depth of the tree instead of a number of internal nodes. This allows them to constrain the complexity of the expression since the complexity is dependent on the depth of the tree. First, they generate a perfectly balanced tree and fill the operations into the tree. If the operation is binary, then both of its children are considered, if the operation is unary, only the left child is considered. To allow an unbalanced tree, the authors introduce $\text{id}(\cdot)$ operation, which returns its input unchanged. To further reduce the complexity of the equations, the authors also randomly mark nodes as terminal, which results in shallower trees.

2.4.2 Coefficient handling

Currently, there are two approaches how to handle coefficient. The first one is not to handle the coefficients during the training and to find them in the postprocessing step using some global optimization such as BFGS [Fletcher, 1987]. This approach is taken by Valipour et al. [2021] and Biggio et al. [2021]. First they take the expression and remove the coefficients e.g. $2.1 \sin(-5x)$ becomes $C_1 \sin(C_2 x)$. The model is then trained on these so-called skeletons using cross-entropy, and in the postprocessing step, the coefficients are fitted using global optimization. The disadvantage of this approach is that the model does not see the coefficients during the training, which could theoretically help the model to

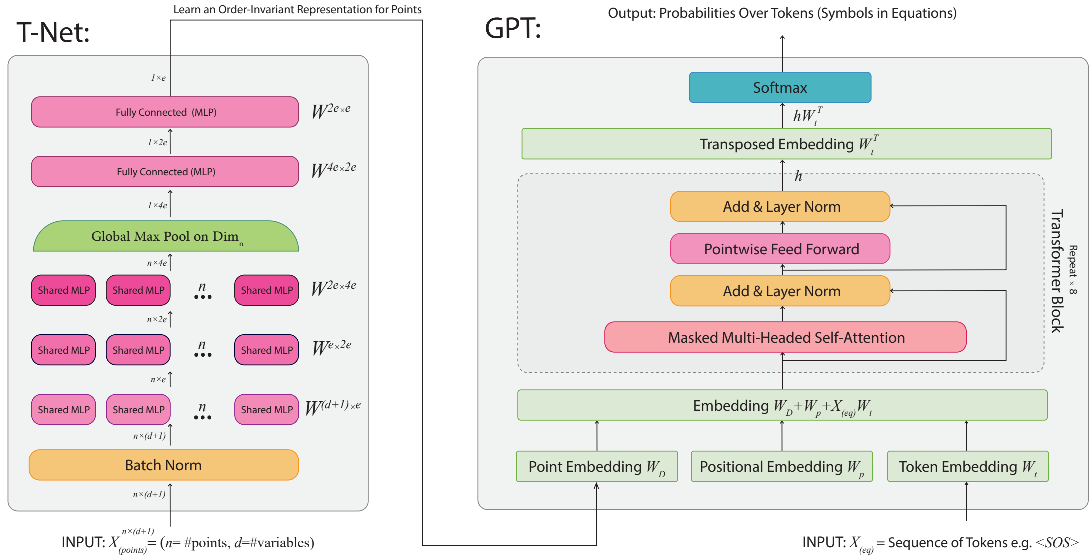


Figure 2.3: The architecture of Symbolic GPT. Source: [Valipour et al., 2021]

make better decisions, and the second one is that the global optimization can be quite slow, which slows down the whole inference and makes it harder to use larger beam width during the beam search.

The second approach is taken by d'Ascoli et al. [2022], where they encode the coefficients into the symbolic representation, and therefore they are part of the model vocabulary. There are two cases of how to handle the coefficients. In the case of integers, the authors encode the integers using some large base b and add the sign. So for example -325 would be represented in base $b = 10$ as $[-, 3, 2, 5]$ and in base $b = 30$ as $[-, 10, 25]$. The size of the base b is a hyperparameter, and it creates a tradeoff between the length of the sequence and the size of the vocabulary. In their experiments, they use base $b = 10000$. In the case of floats, they use the same approach as Charton [2021], where they encode the floats in the base of 10 floating-points notation and encode them as a sequence of 3 tokens. Sign, the four most significant digits (mantissa), and exponent. Therefore $\frac{1}{3}$ becomes $[+, 3333, \mathrm{E}{-}4]$. The disadvantage of this encoding is that it has limited precision due to the length of the mantissa. This has an effect that when approximating complex functions, only the largest term in its asymptotic expansion is usually predicted. The advantage, on the other side, is that it can handle quite large values (up to 10^{100}) [d'Ascoli et al., 2022].

2.4.3 Symbolic GPT

Another approach is called Symbolic GPT Valipour et al. [2021], which takes its inspiration from the famous GPT2 model [Radford et al., 2019] and combines it with ideas from T-Net [Qi et al., 2016]. The model is trained on expressions, generated by generating a completely balanced tree as described in Valipour et al. [2021] and Section 2.4.1. From this generated tree, the points are sampled. In the case of a single dimension, only 30 points are sampled. The model is then trained using these generated pairs.

The architecture can be seen in the Figure 2.3 and it consists of two parts. The first part of the network consists of the T-Net [Qi et al., 2016], whose goal is

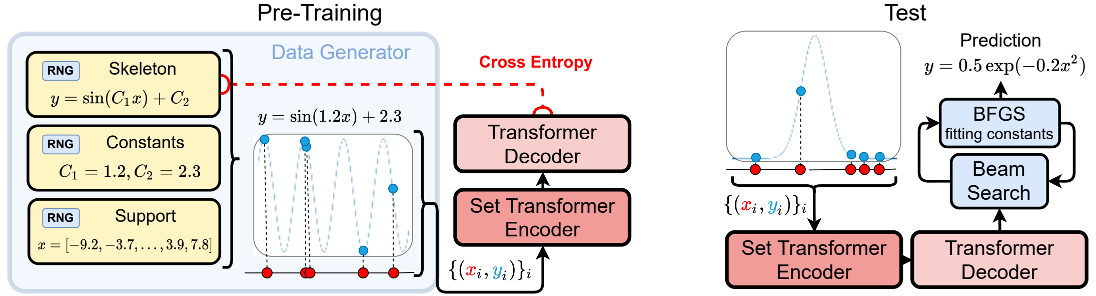


Figure 2.4: The architecture of Symbolic Regression that Scales. Source: [Biggio et al., 2021]

to create a representation of the input points. The T-Net [Qi et al., 2016] consists of several layers, first, it passes the input data through the batch norm layer [Ioffe and Szegedy, 2015], then several feed-forward layers are used. Each of these layers processes each point with the same feed-forward layer. Then the authors take the representation of each of the points and perform a global max pool over the feature dimension. This representation is then fed through feed-forward layers to get the final representation. This representation is then called Point Embedding. The second part is GPT encoder-like architecture which takes Point embedding, Positional Embedding for each of the input tokens, and then Token Embedding which embeds the symbolic tokens (sin, cos, ...). This overall representation is passed through the encoder layers, and the final prediction is made by the softmax layer, which gives the probability of the next token. This predicted token is concatenated to previous inputs, and the prediction is performed once again.

The whole model is trained in an end-to-end manner using cross-entropy loss, and the expression coefficients are fitted using global optimization BFGS [Fletcher, 1987].

2.4.4 Neural Symbolic Regression that Scales

Biggio et al. [2021] takes an inspiration from Lee et al. [2018] where they use its encoder and original decoder from transformer [Vaswani et al., 2017]. The architecture can be seen in Figure 2.4. First, Biggio et al. [2021] generates random expression as random unary-binary tree as described in Lample and Charton [2019] and samples uniformly 128 points from interval $[0, 1]$. Then it removes coefficients from the expression to create a skeleton (e.g. $x + 1$ becomes $x + C$). These sampled points are then fed into the transformer and trained using cross-entropy loss between the predicted skeleton and the ground truth. The skeleton is generated using beam search during the inference, and then the constants are fitted using BFGS [Fletcher, 1987].

2.4.5 Deep Symbolic Regression for Recurrent Sequences

d'Ascoli et al. [2022] extends the problem of symbolic regression to recurrent sequences. They use the idea from Lample and Charton [2019] to generate the dataset while altering the original algorithm by introducing recurrent terms. They

encode the floats and integers into the symbolic representation as described in the Section 2.4.2 and train the model using cross-entropy loss. Compared to other methods, they sample the equation equidistantly, since they are working with sequences that take integer input. They also compare a different number of sampled points ranging from 5 to 30. For the model, they use the original transformer [Vaswani et al., 2017] and train it on five million equations. They also use beam search in their experiments, and instead of ranking each of the hypotheses by the log probability and the length of the output, the authors use the error on the input points to select the best one.

3. Method

The goal of this chapter is to describe the method, that we use to solve the problem of symbolic regression. In short, we follow Lample and Charton [2019] algorithm to generate the dataset and then train our transformer model based on set encoder [Lee et al., 2018] and original decoder [Vaswani et al., 2017] in end-to-end manner. In comparison to current state of the art approaches, our method has many advantages. Mainly that the model is trained only once (genetic algorithms and Petersen [2019], Mundhenk et al. [2021] needs tens of minutes to find the right expression), predicts the coefficients jointly (Valipour et al. [2021], Biggio et al. [2021] needs to find the coefficients using the global optimization) and in comparison to d'Ascoli et al. [2022], our model does not have precision constraints in mantissa. We also introduce local gradient search to further improve the model coefficients.

3.1 Dataset

In order to create a dataset, we first randomly select the number of operators and then we follow the Algorithm 3 proposed by Lample and Charton [2019]. The unnormalized probabilities of unary operations can be seen in Table A.1, for binary operations in Table A.3 and for leafs in Table A.2. Then we use SymPy [Meurer et al., 2017] to simplify the expression. However if SymPy is unable to simplify the expression in 5 seconds, we ignore the generated expression. After that, we sample randomly uniformly n points from the range $[-5, 5]$. In our experiments, we have found out, that in some cases, the interval $[-1, 1]$ was not sufficient, since for example $\sin x$ was easily approximated by polynomial which had negative effect on the results. Therefore we have decided to extend the range to $[-5, 5]$. If there was some error on this range, meaning if there was some Not-A-Number, infinity, complex number or if the absolute value of the coefficient is larger than 1e7, we skip the range and continue with $[0, 5]$ or $[-5, 0]$ (similarly, for more dimensions, where we enumerate all these possible combinations). This error can happen in many natural cases such as $\log x$, which is not defined on $(-\inf, 0]$. We also throw away an expression if the number of tokens representing the expression is larger than 50 (expression is too complex), the found coefficients are larger than 10^9 (too large) or if the absolute value of the coefficient is in interval $(0, 10^{-9}]$ (almost zero coefficient). These constrains are there because of the numerical stability and the requirements of the encoding. Besides throwing out expressions which contain too large or too small values, we throw away linear functions if they were generated only after simplification, but not before e.g. if the generated expression is $x + x^0$, we throw it away, but if the expression is $x + 1$, we keep it. We also throw away any constant function e.g. 5. The reason for both of these filtering is, that many of the functions were linear after simplification and the dataset was dominated by the linear and constant functions, which prevented the model from learning more complex functions. The second reason is, that if we are trying to model a function, linear or constant function is usually the first choice, that we try, before exploring more complex functions. We also filter out expressions which were simplified and contains operation which is not in the

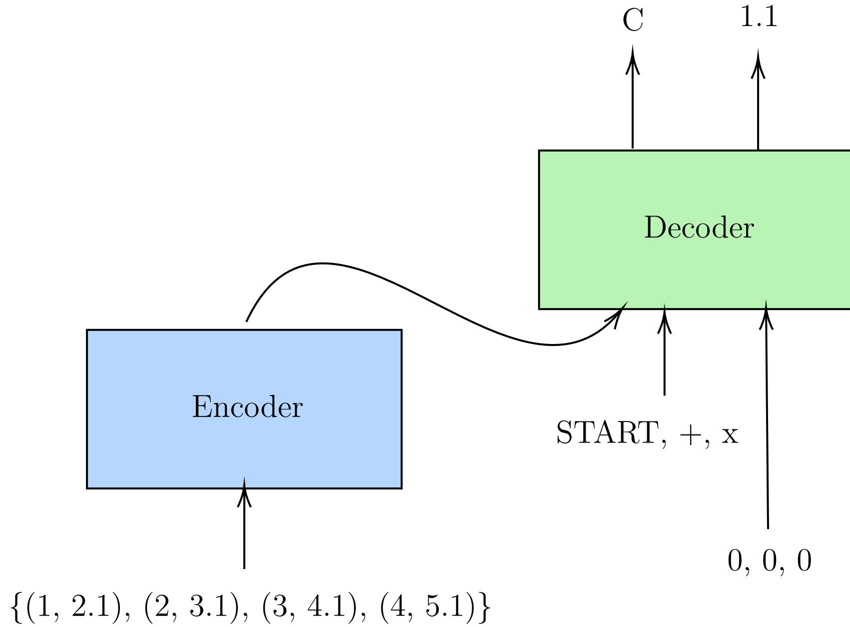


Figure 3.1: Model architecture

vocabulary (e.g. hyperbolic trigonometric functions or absolute value) its content will be described in Section 3.3 (generally it contains most of of the common mathematical operators such as sin, cos, exp, ...). The reason for throwing out these functions is that they can be computed using different kind of formula and they are very rare in the dataset (cosh appeared, at maximum, in 0.02 % of generated expressions). If we would like to keep these operations, we could substitute them with their equivalent form.

We have tested several settings for the maximum number of operators and empirically found that the value of 10 (same value as in d'Ascoli et al. [2022]) gives the best tradeoff between the complexity and the variety of the expressions. If we have used larger number, the expressions become too complex, which slows down the training process, on the other hand if we select the smaller number the complexity of the expressions is too low (mainly due to the simplification, which generally reduces the number of operators). Interestingly d'Ascoli et al. [2022] skipped the simplification step, which made the training loss larger, however the final performance was unchanged. We have also experimented with the number of equations, number of sampled points and number of variables. The effect of such changes will be described in next chapter.

3.2 Architecture

In general our architecture consists of Set Transformer encoder as described in Lee et al. [2018] and decoder as described in Vaswani et al. [2017]. The diagram of the transfomer architecture, can be found in Figure 3.1. The input to the encoder is set of points, and the output is hidden representation $z \in \mathbb{R}^{k \times d}$, where d is dimension of the output and k is number of seed vectors. The exact hyperparameters will be later described in experiments. First we take input points and pass them through linear layer, which embeds them into higher di-

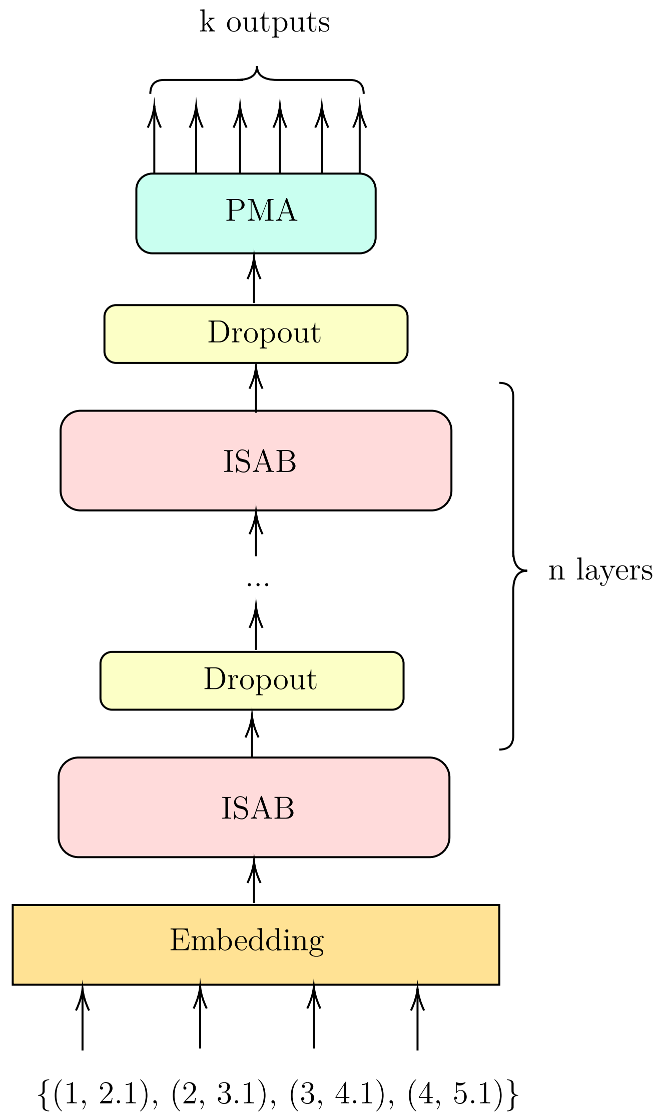


Figure 3.2: Encoder architecture

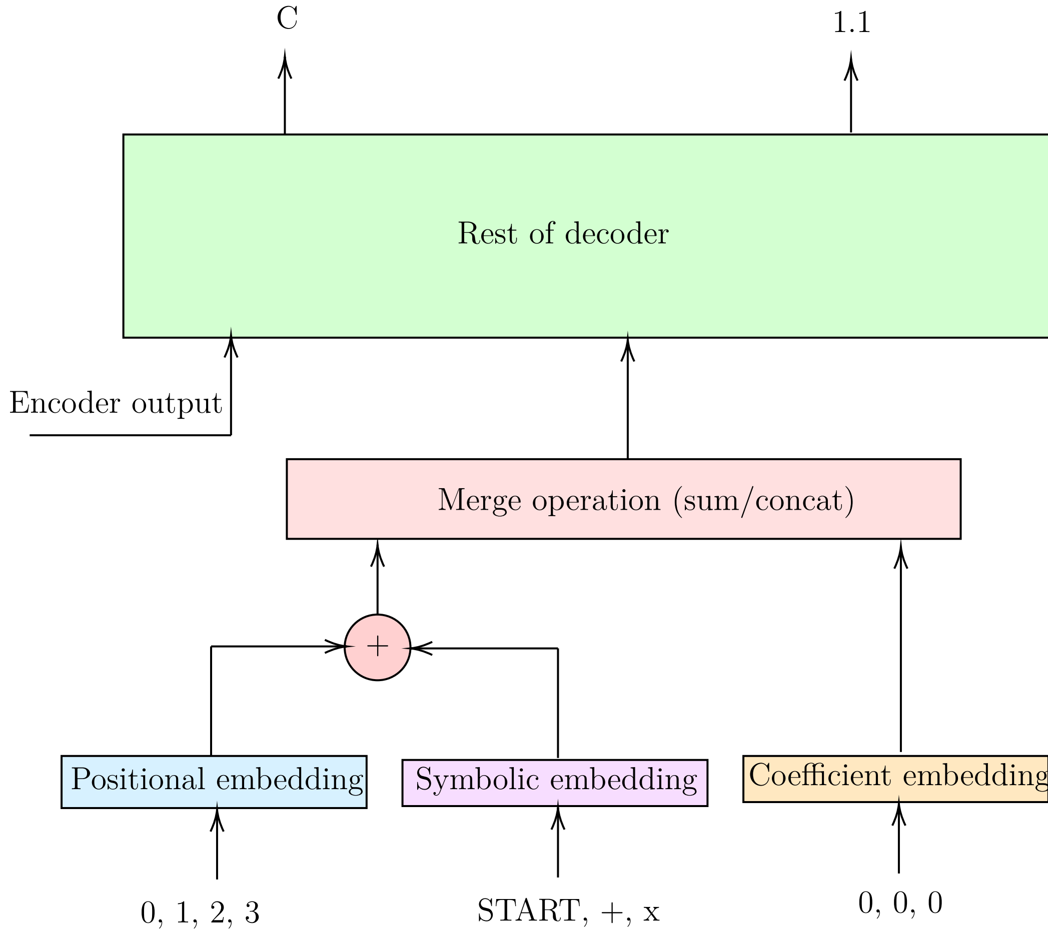


Figure 3.3: Decoder architecture

mension (similarly to embedding layer in language modelling). After that we pass it through several ISAB layers ending with Dropout [Srivastava et al., 2014] and use Pooling-by-Multi-Head-Attention to get final representation z. In comparison to Lee et al. [2018], we did not find beneficial to use SAB layer in the end, the performance was almost the same. Architecture of encoder can be seen in Figure 3.2. The representation from the encoder is then passed to the decoder with previous symbolic tokens and previous coefficients. These previous symbolic tokens are passed through the embedding layer to get high dimensional representation. After that, the learned positional encoding is summed with the symbolic representation to give the model some sense of the order. The final representation is then concatenated or summed (based on architecture) with the representation of symbolic tokens. The architecture of the decoder can be seen in Figure 3.3. This representation is then passed through decoder as described in Vaswani et al. [2017]. In the end, the internal representation is passed to classification head, which predicts the symbolic representation and to regression head which predicts the coefficient for the constant symbol.

Symbolic representation:	+	C	sin	*	2	x
Coefficients representation:	0	1.7	0	0	0	0
Improved symbolic representation:	+	C_1	sin	*	2	x
Improved coefficients representation:	0	0.17	0	0	0	0

Table 3.1: Example of function representation.

3.3 Function representation

In our work we use pre-order representation of an expression as the function representation. For example $1.7 + \sin(2x)$ will becomes, [+, 1.7, sin, *, 2, x], we further process the expression and replace the integers and floats with special symbol C which indicates that the coefficient should be predicted by the model. The previous expression therefore splits into two parts. The first part is symbolic representation [+, C, sin, *, C, x] and the second is coefficient representation [0, 1.7, 0, 0, 2, 0]. We call this base encoding.

We also propose improved way how to represent these coefficients. Instead of predicting the float number as 1.7, we first transform it to scientific-like format, where the mantissa is between $[-1, 1]$ and the exponent is part of symbolic representation. The preprocessing goes like this. First we take the logarithm of the constant to get the exponent $= \lceil \log_{10} C \rceil$ and then mantissa $= \frac{C}{10^{\text{exponent}}}$, this will transform the constant C to $C = \text{mantissa} \cdot 10^{\text{exponent}}$, where mantissa is in range $[-1, 1]$ and exponent is integer. So instead of predicting 1.7 and C, the model predicts 0.17 and C_1 which represents number as $0.17 \cdot 10^1 = 1.7$ (0.17 from the regression head and C_k which represents 10^k). Previous example [+, 1.7, sin, *, 2, x] therefore becomes [+, C1, sin, *, C1, x] and [0, 0.17, 0, 0, 2, 0]. We call this extended encoding.

Furthermore we also help the model by including several integers from -5 to 5 as part of vocabulary to better represent rational numbers or integers in general which naturally occur in equations e.g. $x^{\frac{1}{3}}$ or $3 + x$. The representation can be seen as a first part of the Table 3.1. We later empirically show that this representation is more stable since the output from the regression head can be only in interval $[-1, 1]$ and therefore the loss is bounded. It also solves issues with handling the different magnitudes of two losses. Note the maximum representable value is upperbounded due to the maximum exponent that we can represent, but it is not lowerbounded. Imagine, that our vocabulary only consist of C_1, C_0 and C_{-1}. We would like to predict 0.001, but the symbolic output can only have 10^{-1}. However nothing keeps the model from predicting 0.01 and C_{-1} and therefore it is able to represent number arbitrarly close to zero. On the other hand, this model would not be able to predict number 1000, since it is not able to predict 1 and C_3. However, we believe, that this should not be an issue, since vocabulary contains all exponents from $[10^{-10}, 10^{10}]$ (only 20 tokens) and therefore is able to represent numbers up to 10^{10} which is usually sufficient.

This representation has an advantage over the previous approaches such as Valipour et al. [2021], Biggio et al. [2021], in which they did not predicted these coefficients as part of output and therefore the model could not get some sense of coefficients and how the output changes under the change of these coefficients. Secondly it improves the approach by d'Ascoli et al. [2022], which is

able to represent only the four most significant digits. One of the issues with the extended approach is, that it becomes significantly harder to be more precise when predicting the coefficients, when the magnitude increase. So for example $\sin(x + 0.017) + 10781.5 \cdot x$ becomes [+, sin, +, x, C-1, mul, C5, x] as symbolic representation with [0, 0, 0, 0, 0.17, 0, 0.107815, 0] as coefficients. Since the binary representation of floats in IEEE 754 is sparse it will become harder and harder to predict the exact coefficient for larger values. The complete vocabulary of the model can be found in the Table A.4. Note, that we do not use subtraction operator "-", since $a - b = a + (-1) \cdot b$ and division operator, where $\frac{a}{b} = a \cdot b^{-1}$, this reduce the number of operators, however it lengths the expressions.

3.4 Training

During the training we have to consider several possible losses. Since we need to combine loss for both symbolic representation and for coefficients. The loss for symbolic representation is straightforward, since we are training the Transformer in supervised manner. The symbolic representation is trained using cross-entropy loss [Goodfellow et al., 2016] defined as

$$\mathcal{L}_{class} = -\mathbb{E}_{x,y\sim\hat{p}_{data}} \log p_{model}(y|x). \tag{3.1}$$

However for the regression loss, we have many possible options such as mean squared error defined as:

$$\mathcal{L}_{mse} = \frac{1}{k}\|\hat{y} - y\|_2^2, \tag{3.2}$$

where k is size of y, $\hat{y}$ is the prediction from the network and y is ground truth. Alternatively we can use mean absolute error:

$$\mathcal{L}_{mae} = \frac{1}{k}\left|\hat{y} - y\right|. \tag{3.3}$$

Interestingly, we can also combine these two losses to get Huber loss [Huber, 1964] which is linear for large values and quadratic for small values. The Huber loss is defined as:

$$\mathcal{L}_{huber} = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{for } |y - \hat{y}| \leq \alpha \\ \alpha(|(y - \hat{y})| - \frac{1}{2}\alpha) & \text{otherwise} \end{cases},$$

where α is a hyperparameter, usually $\alpha = 1$. In general, in case of simple coefficients representation (where coefficient representation can have arbitrary number), we have used Huber loss since the mean squared error was too noisy, for the extended coefficient representation, we have used mean squared error.

The final loss is then defined as

$$\mathcal{L} = \mathcal{L}_{class} + \lambda\mathcal{L}_{\text{reg}}, \tag{3.4}$$

where reg is one of the regression losses and λ is hyper-parameter. We also calculate the regression loss only on places, where the ground truth of symbolic output was C. One consideration which we need to take before the training is what value should λ have, since if the λ value is too large, than the model will not

learn to output the correct symbolic representation, however if the λ is too small, the model will not learn to output the correct coefficients. We have therefore decided to use cosine decay [Loshchilov and Hutter, 2016], which is defined as:

$$\text{lr} = 0.5 \cdot \text{init} \cdot (1 - \alpha) \cdot (1 + \cos \frac{\pi \cdot \text{step}}{\text{decay steps}}) + \alpha, \tag{3.5}$$

where step is minimum between decayed steps and current step, decay steps is the number of steps we want to decay, and α is minimum value as a fraction of initial value and init is initial value. However, since this equation decay with time, but we want to increase the λ with time, we use

$$\lambda = \text{init} - \text{lr}. \tag{3.6}$$

To further strength the effect, we also delay the schedule and keep $\lambda = 0$ for first k epochs, which can resembles curriculum learning [Soviany et al., 2021].

Note that when training the model with base encoding, another issue which we met during the training was that the model did not perform very well during the inference. The hypotheses was, that the model is trained using teacher forcing on the coefficients, which are always correct. However during the inference, this is not necessary true. To emulate this issue, we inject random noise from $\mathcal{N}(0, \sigma^2)$ during the training. The value for σ^2, is usually selected in similar manner as for the λ. We use cosine decay as defined in Equation 3.5, however in this case we lower the noise during the training. This phenomena did not appear using our extended encoding but we have found beneficial to inject small noise during the training.

3.5 Inference

During the inference, we use two different decoding strategies. The first one is greedy decoding, where each timestamp we select the most probable token given the previous tokens and the encoder output. The second one is a random sampling in which case we generate n independent beams and sample from them using Top-K [Fan et al., 2018], Top-P sampling [Holtzman et al., 2019] or temperature sampling [Ackley et al., 1985]. To further improve our results, we run gradient descent [Ruder, 2016] on the predicted coefficients. As a loss function we use mean squared error between the y values from the input points and y values which were obtained from predicted function $\hat{f}(x)$ on x from the input points. The final expression is then selected with the smallest mean squared error.

4. Experiments

In this chapter, we describe how the experiments were performed and evaluated. We focused mainly on metrics that tell us how the model would perform during the inference, but it is also important to monitor other metrics such as training or validation loss to catch any possible errors.

4.1 Metrics

This section will describe metrics that we use to assess the model quality and its predictions. We can generally divide metrics into three categories. The classification metrics tell us how well skeletons are predicted, the second is the coefficient metrics which measure how successfully the model can predict the coefficients. The goal of the third metrics is to combine both the symbolic and the coefficient predictions, which measure the overall quality of the predictions. During the training, we prioritize the classification metrics over the regression metrics since when the model cannot predict the correct formula, the perfect prediction of coefficients is useless. However, overall, we care the most about the last kind of metrics since they measure the desired performance.

4.1.1 Classification metrics

Since the goal of the network is to predict the formula exactly, natural way of measuring its success is accuracy. We define the per batch accuracy of prediction $\hat{y}$ as:

$$\mathrm{Acc}(y, \hat{y}) = \frac{1}{m} \sum_{j=1}^{|\mathrm{batch}|} \sum_{i=1}^{|y|} \mathbf{1}_{y_i^j = \hat{y}_i^j}, \tag{4.1}$$

where y_i^j is ground truth vector j on position i, and m is the number of positions where the accuracy was calculated. The prediction or ground truth can be longer, and therefore, we calculate the accuracy only on valid positions from the ground truth. We defined the accuracy per batch to be able to define what we call hard accuracy, which tells us exact match accuracy.

$$\mathrm{AccHard}(y, \hat{y}) = \frac{1}{n} \sum_{j=1}^{|\mathrm{batch}|} \mathbf{1}_{y^j = \hat{y}^j}, \tag{4.2}$$

where n is the number of vectors in the batch. These metrics are important in assessing how good the model predicts the formulas without constant. If the model cannot predict the symbolic representation successfully, then there is no need to predict the coefficients. We also use cross-entropy loss to measure how good the model is performing.

4.1.2 Coefficient metrics

To measure the quality of the predictions of the regression head and to be able to compare different losses, we use the same metrics for all models, mean squared error, mean absolute error, and Huber loss. Note that the loss computation is

done only if the model should predict the symbol C representing the constant since we can always zero out the values for non-constant tokens during the inference. Besides those metrics, we also introduce Hard regression metrics, which reflect how large the error on coefficients is if the symbolic prediction was completely correct.

$$\text{RegMetHard} = \frac{1}{n} \sum_{j}^{\forall \text{vectors}} \mathbf{1}_{y^j = \hat{y}^j} \text{RegMet}(c, \hat{c}), \tag{4.3}$$

where y^j is the symbolic ground truth, $\hat{y}^j$ is the symbolic prediction, c is ground truth of coefficients, $\hat{c}$ is prediction of coefficients, n is the number of symbolic predictions which were completely right and RegMet is some regression metric. It does not make a lot of sense to look at error of the regression head, if the model did not predict the expression completely correctly since the regression values can be on the same position, but in totally different expression e.g. $\exp(100x)$ vs $\ln(100x)$, regression error would be 0 but, the expressions are diametrically different.

4.1.3 Hybrid metrics

The disadvantage of using only the coefficient of the classification metrics above is that they do not truthfully reflect the model performance on the underlying task. The issue is that the model can have an accuracy of zero and a large coefficient error, but the prediction is entirely correct e.g. $\cos\left(-\frac{\pi}{2} + x\right)$ and $\sin x$ are equivalent, but the model would receive both large regression error and classification error. To address this issue, we introduce hybrid metrics, which consider both symbolic and regression output. All the hybrid metrics mainly work based on the error between the predicted values $f(x) = y \in \mathbb{R}$ from the ground truth and from the predicted function $\hat{f}(x) = \hat{y}$ on some input $x \in \mathbb{R}^d$, where d is the dimension of the input.

Before we compute the metrics on the inputs, we first need to consider in which range we will sample the points. Firstly, we sample the points from the range $[-5, 5]$ (or similar as we described in Section 3.1 to find the right function domain). We call this range an interpolation range since the model needs to find the function by interpolating the inputs. The second range, from $[-6, 6]$ (excluding the interpolation range), is called the extrapolation range since it shows how good the model is outside of known boundaries. To assess the quality of the model, we will introduce several metrics. In most of these metrics, we report median values instead of mean due to the outliers with large values, which will dominate the mean. Example of such large error are functions $\frac{1}{x}$ and $-\frac{1}{x}$ for $x \approx 0$.

Classic regression metrics

Similar to the metrics on coefficients, we can use similar metrics here, but on the points predicted by the ground truth function and the function predicted by our network. We can calculate mean squared error or mean absolute error on predicted $\hat{y}$ and ground truth values y. The disadvantage of such an error is that it does not consider the magnitude of the predicted values. For example, if we have a function x, we can expect low errors on the interval $[-5, 5]$, however, if

the ground truth is $1000x$, then the errors will be in a different magnitude. To address the issue, we use relative error, which is computed as:

$$\mathrm{RE}(y, \hat{y}) = \frac{1}{|y|} \sum_i |\frac{y_i - \hat{y}_i}{y_i}|, \tag{4.4}$$

where y is vector of values from ground-truth function $f(x) = y$, $\hat{y}$ is vector of values from the predicted function $\hat{f}(x) = \hat{y}$ and $|y|$ denotes the size of y.

Coefficient of determination

Another metric that is often used in regression analysis is the coefficient of determination, often called R^2 (R-squared). R^2 is a dimensionless measure of how well a model describes a data set. R^2 is defined as:

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}, \tag{4.5}$$

where y_i is ground truth, $\hat{y}_i$ is the prediction and $\bar{y}$ is the mean of y, $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ [Glantz and Slinker, 2000]. The advantage of R^2 is that it has a nice interpretation. If $R^2 \leq 0$, then it means it would be better just to predict the mean value. If the $R^2 = 1$, then we have perfect predictions. Another interpretation that can be used is that R^2 is the fraction of the variance that the model "explains," and therefore, one would want to maximize R^2 ideally.

4.2 Training

The code for training is implemented using Tensorflow [Abadi et al., 2015] and available at `https://gitlab.mff.cuni.cz/vastlm/master-thesis` or in supplementary materials, where they also contain pretrained models. We trained the model using a multi-GPU setup with batch sizes ranging from 128 to 768, depending on the model and the number of sampled points. During the hyperparameter tuning, we train the model for 130 epochs (or 390 epochs for the final model) using Adam optimizer [Kingma and Ba, 2014] with learning schedule as described in Vaswani et al. [2017], however, divided by 5.0 since the model sometimes diverged during multi-GPU setup. We also use label smoothing [Goodfellow et al., 2016] to regularize the model further. We trained the model on two Nvidia GTX 1080 Ti or on eight Nvidia A100. For the encoder, we use 4 layers, 12 heads in Multi-Head attention, the dimension of the model is 384, and we use 64 inducing points. We also use a dropout layer after each ISAB with a rate of 0.1. For the PMA, we use 32 seed vectors. For the decoder, we have 4 layers, with the model dimension of 256, a number of heads of 8, and the dimension of the feed-forward layer of 256. The names of hyperparameters are the same as in the original Transformer paper Vaswani et al. [2017]. For the regression head, we use two feed forwards layers of size 64 with tanh activation function in the end to squash into the $[-1, 1]$ interval for the model, which contains exponents in the symbolic output (extended encoding) and no activation function for the model which predicts the coefficients on its own. We call this model a large model. We also have a baseline model, which just has a smaller encoder. The only difference

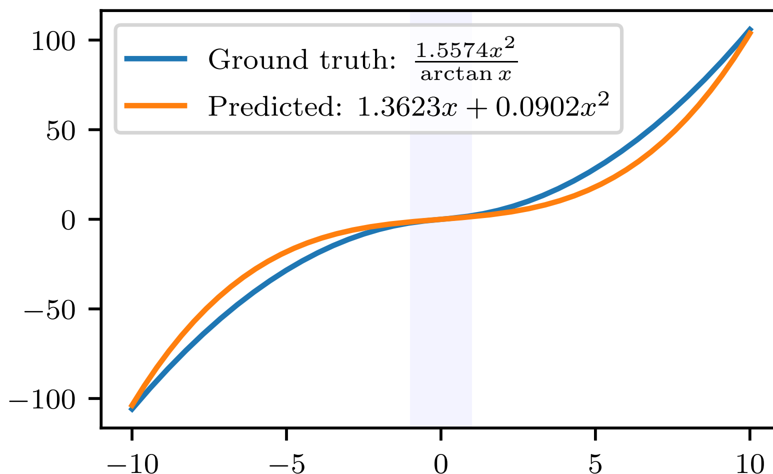


Figure 4.1: The model approximates function using a polynomial.

is that the number of heads is 8 instead of 12 and the model dimension is 256 instead of 384. We use this model to test several of our hypotheses and save the computation power. Our model has around 70 million parameters depending on the hyperparameters. We also concluded a small hyper-parameter search to find the best architecture. To compare the effect of different sizes of datasets, we set one epoch to be equal to one million equations. We also run the cosine scheduling on the regression parameter during the training and delay it for 50 steps. The starting value is 0 and ends at 1.0 in the end. For the injected noise into the coefficients, we have found it most beneficial to use noise with $\sigma^2 = 0.1$ and slowly decay it to 0 during the whole training. The comparison of different noises will be described later.

4.3 Dataset

The dataset is created as described in Section 3.1 using at most 10 operations and with a range $[-5, 5]$. We were also considering using a smaller range however we have run into the issue where the model approximated some functions using a polynomial. One example is function $\frac{1.5574x^2}{\arctan x}$, where model predicted $1.3623x + 0.0902x^2$. These functions are close, and therefore the model had problems distinguishing between them on the interval $[-1, 1]$ during the training the model learned to predict polynomials in most cases. In total, we have generated 130 million equations, which took around 500 CPU days, since generating 10 000 equations takes around 1 hour on average, mainly due to the simplification in SymPy [Meurer et al., 2017]. We have sampled 500 points for each equation, however, we test the effect of the number of sampled points in the results section. We have generated 10 000 equations with the same hyper-parameters as the training generator but with different random seeds for the validation set. Note that the validation set can contain the same expressions as the training set, but they have distinct sampled points. The 2D dataset was generated similarly, except that we have generated only 100 M equations.

We were also considering generating a dataset, which contains only unique equations. However, one issue with generating this dataset is that it is very hard to create a unique expression not contained in the dataset. Such problematic examples can be $\sin x$ and $\cos\left(x - \frac{\pi}{2}\right)$, which are semantically the same but have different symbolic and coefficient representations. This is not the only issue. Even if we could filter such equivalent pairs, we would have problems with functions that are the same on the closed interval but different outside of this interval. Filtering such equations by the points would be costly, and therefore we have decided not to create such a dataset.

4.4 Local search

Since the model is predicting symbolic representation and coefficients jointly, we would ideally want to improve the coefficients further to fit the input points better. We have chosen to use a local gradient search to improve the points further. We use gradient descent [Ruder, 2016] with a learning rate of 0.001, momentum 0.9, and clip norm 10. We use the mean squared error between the predicted values and the ground truth as a loss function. We ran the gradient search up until the loss did not improve by at least 0.1 % for 5 iterations. After that, we return the best coefficients with its loss. This approach, however has one issue, if we use coefficients as they are, we can run into problems with function domains. For example, let's imagine we are trying to find x^{10}, the model predicts $x^{10.15}$, and we would like to improve these coefficients further. The issue is that for $x < 0$, the result is a complex value. To solve this issue, we artificially add $\text{abs}(\cdot)$ into the expression if the operation would create a complex value. Note that we do not add the $\text{abs}(\cdot)$ if it is not necessary in cases such as x^3, where negative values can appear in the input.

4.5 Evaluation benchmarks

Since there is no single dataset that everyone uses, mainly due to the prevalence of genetic algorithms in the area of symbolic regression, everyone uses its own dataset. This has an effect that the final distribution of the expressions in the dataset is different for each method and each generator hyper-parameters. Because of this, some methods can generate datasets that are similar in distribution to the benchmark datasets, and therefore, they can have some advantage. However, the equations presented in these benchmarks are usually expressions, which could one see in real-world applications. Another problematic aspect of the evaluation is that many metrics measure the quality of the regression, and each author decides on its own which metric they will use.

To compare our results, we will mainly follow the benchmarks as described in Mundhenk et al. [2021]. They use the Nguyen benchmark [Uy et al., 2011], R rationals, [Krawiec and Pawlak, 2013] and their newly proposed benchmark Livermore [Mundhenk et al., 2021], we extend these benchmarks with Keijzer [Keijzer, 2003], Constant, and Koza [Koza, 1994]. However, these benchmarks are mainly built for methods that are optimized per equation and therefore contain the ground truth function, the range where it should be sampled, sampling

type (uniform or equidistant), number of sampled points, and the set of elementary functions, and number of variables. However, our method is trained on the $[-5, 5]$ range with uniform sampling, a fixed number of points, and a fixed set of elementary functions, and therefore the comparison will not be strictly comparable. The benchmark functions can be seen in Table A.5.

5. Results and discussion

This chapter will outline our results and compare several different settings and their effect on the results. Lastly, we will compare our best model to other approaches. Note that in most of these experiments, the different settings are dependent on each other, e.g. the number of generated equations and the number of sampled points. If not stated differently, we use 100 points and 130 million equations, we concatenate the representation of symbolic embedding with coefficient embedding and sum it with positional encoding. We also use the symbolic representation with constants containing exponents (extended encoding). We will compare the results from greedy decoding on 1024 random equations from the validation set for each setting. The results can be further improved by using random sampling with Top-k sampling [Fan et al., 2018] with a local search on the constants shown on the final model. The full metrics and training process will also be shown only for the final model. Later we will also assess the model's ability to generalize outside its training hyperparameters. To compare different hyperparameters, we use relative error, denoted as RE, and R^2 metric on interpolation range $[-5, 5]$ and extrapolation range $[-6, 6]$ (excluding the range $[-5, 5]$). Note that we use median values due to outliers.

5.1 Dataset hyperparameters

In this section, we will consider different settings of function representation, number of sampled points, and number of equations. Overall, using extended coefficient representation (the one with exponents), 100 points, and as much data as possible is the best choice of hyperparameters, which gives the best trade-off between the performance and computational effectiveness.

5.1.1 Effect of function representation

In this experiment, we want to compare how the expression representation impacts the model performance. We present two approaches how to predict these coefficients. Both of them predicts the symbolic output in the same way, however the difference is that for the first representation the model predicts the coefficient for the whole $\mathbb{R}$ e.g. function is $\sin x + 20 + x^{0.7}$ which is [+, sin, +, x, 20, pow, x, 0.7] and the symbolic output is [+, sin, +, x, C, pow, x, C] and the coefficient output is [0, 0, 0, 0, 20, 0, 0, 0.7]. The second approach uses the exponents in the symbolic output and values from $[-1, 1]$ in the coefficients so [+, sin, +, x, 20, pow, x, 0.7] becomes [+, sin, +, x, C1, pow, x, C0] for symbolic output and [0, 0, 0, 0, 0.2, 0, 0, 0.7] for coefficient output. The hypothesis is that the second representation is more stable during the training since the coefficient values are bounded, however, it could possibly have issues with values close to the correct values e.g. ground truth is 0.99 and the model could have problems if it should predict C0 or C1. One of the issues with the classical encoding is that the regression loss tends to be pretty large and noisy during the training. The second issue is that the regression loss is larger than the classification loss, and therefore we use the cosine schedule as defined in Section 3.4 with the final learning rate

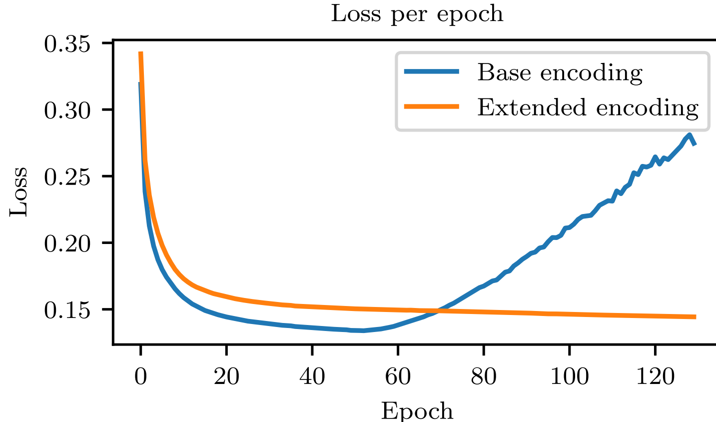


Figure 5.1: The effect of encoding.

Encoding type	RE inter. ↓	RE extrap. ↓	R^2 inter. ↑	R^2 extrap. ↑
Base	0.269	0.1612	0.9116	0.7314
Extended	0.1143	0.0775	0.9817	0.9392

Table 5.1: Effect of expression encoding.

of 0.01 and delay it for 50 epochs. We could start with some fixed value, in the beginning, however, in our experiments, the classification loss diverged when we started with a non-zero value for the regression loss. The results can be seen in Table 5.1. The extended encoding is superior in all of our metrics, and even though we would probably be able to find ideal hyperparameters to match the performance, it is easier to use the extended encoding and not to tune the parameters. Furthermore, the training is more stable, it requires less hyperparameter tuning since it is not sensitive to the regression loss schedule and noise schedule, and it does not diverge in the end due to the need to balance regression and classification loss. These issues can be best seen on the training loss in the Figure 5.1. Note that these losses are not comparable in absolute value. It is important to compare them by their shape since base encoding uses Huber loss and extended encoding uses mean squared error.

5.1.2 Effect of number of sampled points

One of the most critical hyperparameters is the number of sampled points. We can expect that with the increase of sampled points, the model's performance also increases. However, with a large number of points, the time and memory complexity also increase, which slows down the training. The second downside is that if we would like to use our model in a real-world scenario, we usually do not have that many points. We have tested 100, 200, 300, and 500 points for the smaller model. The results can be seen in the Table 5.2. As you can see, the more points were sampled, the better model performed. This is, however, expected. Interestingly the model with 100 and 200 points has similar performance, and then the 300 and 500 also have similar performance. The explanation could be that there is some threshold of the number of points (in this case, somewhere between

# points	RE inter. ↓	RE extrap. ↓	R^2 inter. ↑	R^2 extrap. ↑
100	0.1596	0.1154	0.9580	0.8223
200	0.1695	0.1169	0.9587	0.8378
300	0.1468	0.0897	0.9734	0.8620
500	0.1231	0.0881	0.9815	0.9127

Table 5.2: Effect of number of sampled points

# equations	RE inter. ↓	RE extrap. ↓	R^2 inter. ↑	R^2 extrap. ↑
1 M	0.1352	0.0905	0.9694	0.8864
10 M	0.1113	0.0636	0.984	0.9505
50 M	0.0984	0.0637	0.9853	0.9476
130 M	0.0957	0.0644	0.9872	0.9574

Table 5.3: Effect of number of equations

the 200 and 300) where the number of points below this does not improve the performance, but once we pass this threshold, the performance improves. In the end, we have decided to use the model with 100 points since it is comparable with the performance of 200 points, however, it is more computationally feasible than the 300 and 500 (500 was not able to finish 130 epochs in 48 hours for larger model). The second reason we have decided to use the 100 points is how the benchmarks are defined since they usually sample a low number of points.

5.1.3 Effect of number of equations

We want to see how the number of randomly generated equations impacts the model performance with these experiments. Usually, we want to use as much data as possible in machine learning, especially deep learning. Many problems can be solved by using a large amount of good quality data and a large model. Fortunately, in the case of symbolic regression, we can generate as much data as we want, therefore, we have generated 130 million random equations. One would expect that the model performance would improve with the increase of data if the model has enough capacity. It can be seen from the Table 5.3, that the performance of the model increases with the number of equations, however, the difference between the 50 million and 130 million is not so significant.

5.2 Architecture considerations

In this chapter, we will outline our architecture considerations, we mostly settled on using a concatenated representation of embedding and not using SAB after the PMA layer. We have also fixed the overall architecture regarding the encoder from Set Transformer [Lee et al., 2018] and decoder from the original Transformer [Vaswani et al., 2017]. However different architectures are possible, for example d'Ascoli et al. [2022] uses original Transformer.

Merge op.	RE inter. ↓	RE extrap. ↓	R^2 inter. ↑	R^2 extrap. ↑
Sum	0.1133	0.0747	0.9829	0.9371
Concatenate	0.0957	0.06.44	0.9872	0.9574

Table 5.4: Effect of a different merge operation

Size	RE inter. ↓	RE extrap. ↓	R^2 inter. ↑	R^2 extrap. ↑
Baseline (23 M)	0.1286	0.0824	0.981	0.9055
Larger (73 M)	0.0957	0.0644	0.9872	0.9574
Final (96 M)	0.0515	0.0315	0.9964	0.9879

Table 5.5: Effect of a different merge operation

5.2.1 Effect of the embedding merge operation

One decision we need to make is how we will merge the embeddings from symbolic input and the coefficient input. We have tested two settings. In one case, we have pairwise summed these representations, and in the other, we have concatenated them. To make a fair comparison between the two operations, we want to have the same final dimension after the merge operation. We, therefore, decided to use half of the dimensions for the symbolic embedding and half of the dimensions for coefficient embedding for the concatenate case, and full dimension for the summation case. We have tried two different settings of the merge operation and found that the concatenate operation tends to perform slightly better than the sum operation with the same number of parameters and the same hyperparameters. The results can be seen in Table 5.4.

5.2.2 Effect of the model size

The important hyperparameter of the model is the model size. We have decided to try three different models, the first one is the baseline in which the encoder has 8 heads and the model dimension is 256 and the decoder also 8 heads, the dimension of the feed-forward layer is 256, and the dimension of the model is 256. The second model is larger, with 12 heads and a dimension 384 in the encoder and 12 heads, the dimension of the feed-forward layer is 384, and the dimension of the model is 384. The third one, the final, was trained on slightly different hyperparameters, and therefore it is not precisely comparable with the larger and the baseline model just by the number of parameters. The exact hyperparameters of the final model will be described later. Note that most of these parameters are in the encoder part, which in the case of the baseline model has cca 20 million parameters and the decoder 3 million, and for the larger model, 67 million parameters for the encoder and cca 6 million parameters for the decoder. The idea is that the encoder should do the heavy-lifting i.e., transforming the points to latent representation, and the decoder should do the easier work. As expected from the results in Table 5.5, the larger the model, the better it performs, with a significant jump between the larger and final model. Note that the final model was scaled even more and trained for longer, so the difference can also be due to longer training.

Size	RE inter. ↓	RE extrap. ↓	R^2 inter. ↑	R^2 extrap. ↑
No noise	0.1126	0.0805	0.982	0.9339
$\mathcal{N}(0, 0.1)$	0.1019	0.0685	0.9864	0.9408
$\mathcal{N}(0, 1.0)$	0.1159	0.0732	0.9843	0.9378

Table 5.6: Effect of different noise injection

Metric name	Greedy search	Top-K sampling
R^2 interp.↑	0.9964	0.9995
R^2 extrap. ↑	0.9887	0.9984
RE interp. ↓	0.0515	0.0288
RE extrap. ↓	0.0315	0.0090
Accuracy ↑	58.96 %	43 %
Hard accuracy ↑	53.71 %	17.19 %
Hard MSE ↓	0.0008	0.3982

Table 5.7: Evaluation metrics using greedy and Top-K sampling.

5.2.3 Effect of the injected noise

Even though the amount of injected noise into the coefficients does not entirely change the model performance, it trains the model on the possibility of making an error. From Table 5.6 we can see that injecting too much or too little noise hurt the model. Therefore, we have decided to use $\sigma = 0.1$ and use cosine decay on this parameter. The idea is that in the beginning, the model makes these errors more often, and with time, it learns to make smaller and smaller errors. Note that we have used the smaller model to save computational resources in this case.

5.3 Final model

For the final model, we have decided to scale the model even more and trained it for 390 epochs (each epoch is one million equations, and therefore the model has seen the data three times). We have not used this model during the hyperparameter tuning because the training of the model takes around one day, and we wanted to save computational resources. The hyperparameters of the final model can be seen in Table A.6.

5.3.1 Greedy and Top-K sampling results

We use TopK sampling [Fan et al., 2018] with $K = 16$ and sampled 256 equations during the inference. In the end, we have selected the equation which has the lowest MSE between the predicted values and the ground-truth. The results can be seen in Table 5.7. Note that the metrics for Top-K sampling [Fan et al., 2018] are estimated on 256 equations and 1024 for the greedy search. From the Table 5.7 it can be seen that even with a greedy search, the model can perform very well, however, further improvements can be done using Top-K sampling. Interestingly, the accuracy and mainly hard accuracy were lower when using Top-K sampling. The explanation for this observation is that we select the equation

Ground-truth	Greedy search	Top-K sampling
$\cos(x^2)$	$\cos(x^2)$	$\cos(x^2)$
x^2	x^2	$(x^4)^{0.5}$
$(1.9881 + x^x)^{-1}$	$(2.1555 + x^x)^{-1}$	$2.0x^2 + x^2(x^x)^{-1}$
$0.0597 - 5x^{1-\sin\sqrt{x}}$	$-2.2x - 3.7\sin 2.1x - 2.3x\log x$	$-5x^{1-\sin\sqrt{x}}$

Table 5.8: Examples of generated expressions

by MSE and therefore is not selected as the most probable equation but the most fitting equation. This naturally decreases the accuracy since less likely equations are selected. However, on the other hand, the R^2 metric is maximized, and therefore we see better results. We have also decided to show only Hard MSE (MSE calculated only if the skeleton was predicted correctly) since it does not make sense to compare the ability of the model to predict correct coefficients if the skeleton is not identical. The hard MSE is larger for Top-K sampling than is for greedy search because the number of equations in which skeletons were entirely correct was lower in comparison to the greedy search, and therefore the sample is too small.

To show the ability of the model to predict the correct equations, we have plotted several model predictions and their ground-truth, which can be seen in Figure A.1 and in Table 5.8. The interesting ability of the model is to approximate one function using another. For example, in the Figure A.1f, model should predict $(1 + x^{-2})^{-0.5}$, but predicted $\sin(|\text{atan}(x)|)$. If we look at the difference between these two functions, we find that the absolute difference on the interval $[-10, 10]$ is at a maximum 10^{-16}. Note that the greedy search and Top-K sampling results are not necessarily generated by the same set of points, they just share the same ground-truth (some of the numbers can be rounded, so they fit the table).

Furthermore, we have decided to run several experiments using TopK [Fan et al., 2018], TopP, [Holtzman et al., 2019] and temperature sampling [Ackley et al., 1985] to select the best performing sampling method with 256 samples. From the results in Table 5.9, it can be seen that all methods are comparable in all of the metrics and the differences are subtle. However, the best performing method is Top-K [Fan et al., 2018] with $K = 20$, which will be used for the benchmarks. Another hyperparameter that we wanted to test is how the number of generated expressions helps with the model performance. The results run on one GPU with 64 GB RAM and 16 CPUs without coefficients optimization can be seen in Table A.7. There is a trade-off between the running time and the model performance. The best performing model uses 16384 equations, but it ran for almost 12 hours for 256 equations, which is not practical, however, it allows us to find the best possible equation. On the other hand, when selecting only 16 or 32 equations, we quickly received the results, but the results were cca 10 times worse. In the end, we have settled on using 1024 equations for the benchmark.

5.3.2 Local optimization

From the previous results, we can see that the model is performing really well. However, it can be further improved using local gradient search on the coefficients. As described before, we run the gradient search up until it does not further

Sampling method	Value	RE inter. ↓	R^2 inter. ↑	Time
Top-K	4	0.0403	0.9993	31:09
	8	0.0385	0.9995	35:59
	16	0.0404	0.9995	38:37
	20	0.0381	0.9995	41:32
	32	0.0450	0.9992	40:28
Top-P	0.90	0.0406	0.9992	29:41
	0.95	0.0426	0.9993	39:25
	0.98	0.0422	0.9993	43:34
Temperature	0.90	0.0406	0.9992	42:46
	0.95	0.0424	0.9992	45:11
	0.98	0.0428	0.9993	26:42
	1.00	0.0428	0.9992	26:46
	1.05	0.0439	0.9993	27:42
	1.10	0.0471	0.9990	28:35

Table 5.9: Effect of different sampling methods, its hyperparameter, denoted as value, and its performance. The time is minutes:seconds format.

Model	R^2 ↑	RE ↓
Without coeff. optimization	0.9992261	0.040194
With coeff. optimization	0.9999999	0.001029

Table 5.10: Comparison of the model with and without coefficients optimization.

improve the found expression based on the MSE between the input points and the prediction. The results of such comparison can be seen in the Table 5.10 (in this case, we have used Top-K [Fan et al., 2018] with $K = 20$ and 1024 sampled equations). The local optimization helps a model fine-tune its coefficients to the correct values and, therefore, further improve the function's fit. One concern, which will be addressed later, is if the local search overfits the equation on the input points and makes the performance worse outside the sampling range.

5.4 More dimensions

Similarly to the previous experiments, we have trained the model on more dimensional inputs, meaning that the input to the function can be more than one variable. We should ideally find the best hyper-parameters, similarly to the previous experiments, but this would be computationally expensive. Therefore, we have decided to train the model with the same hyperparameters as were previously found except for the number of sampled points, which we have chosen to be 200, and for the number of equations, where we have used 100 million equations. We have trained the model on two-dimensional inputs, however, scaling to more dimensions could be done in the same manner, except for number of sampled points and number of equations, which should ideally be larger. The final results can be seen in the Table 5.11. As you can see, the results are comparable, even slightly better than the one-dimensional case. This observation could be that the

Metric name	Greedy search	Top-K sampling
R^2 interp. ↑	0.9921	0.9996
R^2 extrap. ↑	0.9908	0.9996
RE interp. ↓	0.0921	0.0389
RE extrap. ↓	0.0801	0.0306
Accuracy ↑	57.55 %	42.37 %
Hard accuracy ↑	50.68 %	19.14 %
Hard MSE ↓	0.0009	0.1757

Table 5.11: Evaluation metrics using greedy search for 2D model.

validation set contains moreeasier examples such as $x+1$, $y+1$, $x \cdot y$, ... in comparison to the one-dimensional case. Interestingly, the same phenomena of lower (hard) accuracy for the Top-K sampling [Fan et al., 2018] than for the greedy also occurred here.

5.5 Benchmark results

We use all benchmark functions as defined in Table A.5, and we mainly focus on the comparison between Biggio et al. [2021], denoted NSRS, since it is currently the best performing transformer based solution (we cannot compare it with the promising solution by d'Ascoli et al. [2022], since no model was published) and Mundhenk et al. [2021], denoted as DSR, due to its performance. For the Biggio et al. [2021] we use their 100M model available from their published repository[1] with a beam size of 32 and 100 points for the 1D case and 200 points for the 2D case. For the Mundhenk et al. [2021], we use their implementation available from their published repository[2] with hyperparameters as defined by the authors in their configuration files, including early stopping. Note that for the transformer models, we use the range and sampling method as used during the training (however, different points for inference and evaluation), and therefore the dataset sampling method, range, and allowed functions are used only for the DSR method. We run all experiments only once using 16 CPU threads and 64 GB of RAM with a maximum running time of three hours. We use our best model, trained only for one dimension of the 1D cases and the best 2D model for the 2D benchmark equations to achieve the best possible results. We sample 1000 equations using Top-K with $K = 20$, and we also employ early stopping to speed up the whole process. The results can be seen in the Table 5.12.

Name	Ours		NSRS		DSR	
	R^2	Time (s)	R^2	Time (s)	R^2	Time (s)
Nguyen-1	1	13	1	80	1	14
Nguyen-2	1	30	1	105	1	21
Nguyen-3	1	32	1	131	1	11
Nguyen-4	0.9998	136	0.5484	151	1	34
Nguyen-5	1	16	1	121	1	11

[1] https://github.com/SymposiumOrganization/NeuralSymbolicRegressionThatScales
[2] https://github.com/brendenpetersen/deep-symbolic-optimization

Nguyen-6	1	27	1	101	1	11
Nguyen-7	1	31	0.9967	145	1	134
Nguyen-8	1	5	1	122	1	53
Nguyen-9	1	13	1	166	1	11
Nguyen-10	1	14	1	190	1	23
Nguyen-11	1	8	0.999	296	1	11
Nguyen-12	1	57	1	503	0.9015	1004
Average	**0.99998**	47.5	0.96744	169.46	0.99297	140.25
R-1	1	18	1	135	0.9931	851
R-2	1	38	1	78	0.9681	778
R-3	0.9999	227	1	74	0.9790	937
Average	0.99986	94.33	**1**	95.67	0.97488	855.33
Livermore-1	1	101	1	152	1	16
Livermore-2	1	15	0.1157	168	1	29
Livermore-3	1	15	0.1248	121	1	115
Livermore-4	1	22	0.9967	151	1	36
Livermore-5	1	54	1	463	1	167
Livermore-6	1	39	1	143	1	79
Livermore-7	1	42	0.9999	138	0.9999	707
Livermore-8	0.9998	33	1	118	0.9999	771
Livermore-9	0.9995	117	0.9831	130	1	872
Livermore-10	1	123	1	265	0.9694	971
Livermore-11	1	19	1	195	1	36
Livermore-12	1	14	1	212	1	68
Livermore-13	1	6	1	132	1	18
Livermore-14	1	30	1	572	1	110
Livermore-15	1	70	1	154	1	146
Livermore-16	1	33	0.9986	126	1	286
Livermore-17	1	13	1	288	0.9972	233
Livermore-18	1	15	0.2679	183	0.9677	891
Livermore-19	1	32	1	150	1	30
Livermore-20	1	7	1	115	1	14
Livermore-21	0.9999	138	0.9944	148	1	120
Livermore-22	1	8	1	124	0.9992	364
Average	**0.99996**	43	0.88551	193.09	0.99651	276.32
Koza-2	1	20	1	73	1	27
Koza-3	1	182	1	150	1	408
Average	**1**	101	0.99999	111.5	**1**	217.5
Keijzer-3	1	49	0.7549	174	0.6454	5802.62
Keijzer-4	0.9887	58	0.9991	179	0.8990	9171.36
Keijzer-6	1	13	1	108	1	510
Keijzer-7	1	5	1	138	1	1678.25
Keijzer-8	1	5	1	203	1	86
Keijzer-9	1	116	0.996	126	1	1304.82
Keijzer-10	1	12	0.9335	316	0.9856	2926.49

Keijzer-11	1	78	1	240	0.9536	5235.31
Keijzer-12	1	48	1	521	1	8124.74
Keijzer-13	1	96	1	321	0.9526	6848.41
Keijzer-14	1	37	1	274	1	3864.92
Keijzer-15	1	67	1	266	0.9999	1601.03
Average	**0.99904**	48.67	0.97392	255.50	0.95302	3929.50
Constant-1	1	23	1	175	1	972
Constant-2	1	162	0.0996	130	1	4836.01
Constant-3	1	89	1	348	1	976
Constant-4	1	11	0.9997	317	1	707
Constant-5	1	57	1	127	1	1094
Constant-6	1	19	1	221	1	907
Constant-7	1	146	1	353	1	4186.39
Constant-8	1	220	1	172	1	8851.11
Average	0.99998	90.88	0.88742	230.375	**1**	2816.19
Total avg.	**0.99978**	52.95	0.92901	199.63	0.99443	326.53

Table 5.12: Benchmark comparison, R^2 values are rounded to 4 decimals and time to whole seconds.

From the results, it can be seen that the DSR outperforms the NSRS, however, for more complex expressions (mainly Kaijzer), DSR can be very slow, taking thousands of seconds. Furthermore, if we would allow such longer times for NSRS and DSR, the results would probably be slightly better. We have run the DSR for longer than the others because sometimes we got the first results only after an hour. On the other hand, it can be quite fast for simple equations such as the Nguyen benchmark, where DSR outperforms NSRS in both speed and R^2 in most cases. However, this could be due to the choice of the beam size. If we had lowered the beam size for the NSRS, we would get similar results faster. If we compare our solution to NSRS and DSR, we are, in most cases, faster than NSRS, and also, we can recover almost all of the equations. For example, in the case of the Keijzer benchmark, where DSR mostly fails, we can recover the underlying equations. Model predictions and if the model could recover the original formula can be seen in Table A.8. In most cases, we were able to recover them, however, they are not always the most straightforward solutions. Example of such expression is Keijzer-6 $\frac{x^2+x}{2}$, however what the model found is $\ln(\exp(0.49991x)\exp(0.5x^2))$. These expressions are the same after simplification, however, the latter is unreasonably complex. The reason for this observation is that the model predicted better coefficients for the expression than in the case of simpler expressions.

One could argue that we have such high R^2 is just that we are using the local search, which is from 1000 samples, able to fit almost any function. To test this hypothesis, we have run the benchmark again in the same way as previously, but now, we turn off the local search which should improve the coefficients. From our experiments, it turned out that the model is slightly worse than when using the local search, however, the hypotheses were not confirmed, and the model would have good performance even without the local search. The R^2 and if the skeleton

was found without the local search can be seen in Table A.9.

5.5.1 Performance of the 2D model

To test the ability of our architecture to scale to more dimensions, we have run our 2D model even on the benchmark functions, which contain only one dimension. We have found out that there is some performance degradation, however, it is so small that it does not have almost any effect on the model performance. The model was performing equally good on the easier expression such as $\sqrt{x}$ or polynomials, but it had slight performance degradation on the more complex expressions. This is, however, no surprise since the 2D model also has to handle more dimensions, and therefore it needs to handle more complex scenarios with more variables. The full results can be seen in the Table A.10.

5.6 Discovering mathematical formulas

The goal of this section is to manually inspect the model predictions and look at some interesting cases. One example of such interesting case is ability of the model to learn that some functions are mathematically equivalent. Such examples, which was predicted by model are $\frac{-8}{\ln(x)} = \frac{8}{\ln(\frac{1}{x})}$, so the model found that $-\ln(x) = \ln(x^{-1})$ or $\frac{x}{\tan x^{-1}} - 0.8486 \approx x \cot x^{-1} - 1$, which shows that the model was able to find that $\tan(x) = \frac{1}{\cot x}$. Different cases show, that the model has some sense of the law of exponents $(x^a)^b = x^{a \cdot b}$. The model had to find $(x^{-1})^{1.5}$, but it found $\frac{1}{\sqrt{x^3}}$. Interesting identities are also found for trigonometric functions. Model goal is to find $\cos(3.5005+2x+x^2)$, however it outputs $-\sin(x^2+2x+2)$. These two results are pretty close together. To get from the one side to the other, we have to use this rule $\cos(\frac{\pi}{2} + x) = -\sin(x)$.

$$\cos(3.5 + 2x + x^2) = \cos(\frac{\pi}{2} + 1.9 + 2x + x^2) = -\sin(1.9 + 2x + x^2), \quad (5.1)$$

which is close to $\sin(2 + 2x + x^2)$. Another example of such equivalence is exponential rule $a^x = e^{x \cdot \ln a}$. The model should predict $(-1.3673x)$, but it predicted $\ln(0.2492^x)$ this can however be simplified $\ln(0.2492^x) = \ln(e^{x \ln 0.2492}) = x \ln 0.2492 = -1.3895x$, which is once again very close the required answer. In this case, it is really interesting that the model did not predict simpler function, which should be easier for the model. Another interesting rule, that the model used is $\ln a \cdot b = \ln a + \ln b$. For example, in one of the benchmark functions the model had to find $\frac{x^2+x}{2}$, however what the model found is $\ln(\exp(0.49991x)\exp(0.5x^2))$. This can be simplified as $\ln(\exp(0.49991x)\exp(0.5x^2)) = \ln(\exp(0.49991x)) + \ln(\exp(0.5x^2)) \approx 0.5x + 0.5x^2$.

However, these examples do not exactly prove that the model knows these rules, they could be selected during the sampling just because they had the lowest mean squared error, and it was just a coincidence that the model generated them. However, we can at least claim they belong to the high probable cases since they were sampled from the model.

5.7 Generalization

An interesting aspect of every machine learning model is how well it can perform outside of its training hyperparameters and training data. To assess the so-called generalization, we will look at the ability of the model to perform under different numbers of sampled points, different sizes of sampling range, and also how model performs outside of its sampling range, which was slightly touched before when we looked at the extrapolation range. We use Top-K sampling [Fan et al., 2018] with $K = 20$ and 256 examples in these experiments. If not stated otherwise, we do not use a local gradient search to improve the coefficients.

5.7.1 Extrapolation ability

In these experiments, we want to assess the ability of the model to extrapolate beyond the sampling range since it could be possible that the model can find an almost perfectly fitting expression on the range $[-5, 5]$, however, this expression would fail beyond the sampling range. Note that this especially hard task, since two functions can be almost the same on the closed interval, but totally different outside of it. We also want to test the hypotheses if the local search overfits the inputs points and therefore hurts the model performance in the ability to extrapolate. To test these hypotheses, we run an experiment in which we calculate the R^2 and the relative error based on the distance from the sampling range. We select $\{x \in \mathbb{R} | 5 < |x| < 5 + a\}$, where a is wanted distance and compute the metrics for their prediction $f(x) = y$. From our results, which can be seen in the Figure A.3 and A.4, we can conclude that the relative error increases with the distance from the sampling range, which is nothing surprising, however interestingly, the total error, even for the furthest distance 95, is not so large. This effect is even less apparent for the model using local search to improve the coefficients where the absolute difference between the relative error in the distance of 1 and 95 is just 0.0001. This shows that the model can successfully extrapolate over the sampling range and that the hypothesis that the local search on the coefficients could overfit the equation is incorrect.

5.7.2 Effect of number of sampled points

One of the most critical aspects of the model is how it will behave under the change in the number of sampled points. In real life problems, one usually does not have exactly 100 points. To assess the quality of the model under this change, we use the trained final model on the same equations from the validation set, but for each experiment, we use a different number of sampled points. Each of them is evaluated on new 100 points from the same $[-5, 5]$ range. We have tried several different settings starting from 10 and ending at 1000 points. The full results can be seen in the Figure A.2 and the exact values in Table A.11. It turned out that 10 or 20 points were not enough for the best performance, but from 50 points, the model's performance almost did not change. Therefore we can conclude that if we have at least 50 samples, the model performance should be good enough.

Range	Original		Scaled		Scaled with opt.	
	$R^2 \uparrow$	RE $\downarrow$	$R^2 \uparrow$	RE $\downarrow$	$R^2 \uparrow$	RE $\downarrow$
$[-1, 1]$	0.9836	0.1211	0.9961	0.0514	0.99997681	0.0041
$[-3, 3]$	0.9976	0.0889	0.9955	0.0738	0.99997211	0.0134
$[-5, 5]$	0.9992	0.0402	0.9993	0.0423	0.99999996	0.0010
$[-10, 10]$	0.9967	0.0977	0.9986	0.0650	0.99999541	0.0110
$[-15, 15]$	0.9895	0.1756	0.9970	0.0703	0.99998027	0.0174
$[-20, 20]$	0.9404	0.3352	0.9977	0.0767	0.99997954	0.0225
$[-30, 30]$	0.8350	0.5045	0.9954	0.1028	0.99989361	0.0434
$[-50, 50]$	0.4060	0.6598	0.9840	0.1105	0.99986374	0.0558
$[-100, 100]$	0.2196	0.9933	0.9850	0.1217	0.99889144	0.0336

Table 5.13: Effect of sampling range during the inference.

5.7.3 Effect of sampling range

In this experiment, we want to test how the model will perform under the change of the sampling range, which is one of the most essential hyperparameters which has to be decided before the training. During the experiment, we fix the range from the points sampled e.g. $[-5, 5]$, and then sample 100 points. To evaluate its performance, we use the same range however sample new 100 points once again. We also test three settings, in the first, we take the points as they are without any modification, in the second case, we re-scale its x values to the interval $[-5, 5]$, and in the last setting, we scale the values once again, but we also use the local search to improve the coefficients further. From the results from the Table 5.13, we can see that the sampling range has an impact on the models' ability to predict the right equations. If we do not scale the range to the range that was used by the transformer during the training, then the model has a problem with recovering equations, and the error increase with the distance from the original range. However, this explanation is not the only possible explanation. The error could possibly be observed just due to an insufficient number of sampled points, since it is much harder to predict the correct function using the same number of points on the interval $[-100, 100]$ than on interval $[-5, 5]$. We could also try an experiment in which we use more points, however the issue is, that the model could focus only on the interval $[-5, 5]$ and ignore the other points, which would not give us any interesting insight. Another experiment could be to train the transformer not just on random equations, but also by randomly selecting the range from which are the points generated. This could help the model learn to handle different range sizes and also asymmetric ranges.

5.8 Ablation study

In the ablation study, we want to look at the effect of changing different parts of the network. We want to mainly focus on the coefficient encoding since it is one of the most crucial parts of the network. We have trained our model with the same hyperparameters as in the final model while changing only the final regression loss weight (with 1.0 for the extended encoding and 0.001 for the base encoding) and also the way, how are the coefficients represented for the network. We have tested

Model	R^2 inter. ↑	R^2 extrap. ↑	RE inter. ↓	RE extrap. ↓
No coefficients	0.9929	0.9929	0.1547	0.0802
Base encoding	0.9979	0.9874	0.0669	0.0216
Extended encoding	**0.9995**	**0.9984**	**0.0288**	**0.0090**

Table 5.14: Comparison of different encoding.

three different settings. In the first one, we trained the model only on the skeletons and used BFGS [Fletcher, 1987] to find the coefficients. This is similar setting to what Biggio et al. [2021] propose. In the second experiment, we have used base encoding in which, we have let the network predict the coefficients without any pre-processing, and in the last experiment, we have used our extended encoding in which we have transformed the coefficients into "scientific" like notation as described in Section 2.4.2. The results, when using random sampling with Top-K $K = 16$ sampling, can be seen in Table 5.14. This table also shows the achieved performance gains by using coefficients are not just because of a larger model or more data but because of the usage of the coefficients during the training and inference.

5.9 Acknowledgement

This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic through the e-INFRA CZ (ID:90140).

Conclusion

The goal of this thesis was to find a way to solve the issues that previous approaches had. Mainly the possibility of the slow inference of genetic-based and reinforcement-based solutions and the slightly worse performance of previous transformer-based methods. To solve them, we have introduced a transformer-based model for predicting expressions based on the input data. We have proposed several novel ways to represent and handle the coefficients during the training and the inference. Furthermore, we have also used a local search on the coefficients to improve our predictions further. During our experiments, we found out that transformer architecture can predict the equations accurately even when they contain multiple input variables with enough data and model capacity.

We have measured our models' ability on several metrics and showed that it can generalize in both the number of points given to it, the number of input variables, and the range outside of the sampled points. This shows that the equations found by the model are not just over-fitted on the inputs points and are correct even outside of the sampling range. However, we have also found that the model has problems finding the correct equation when used with sampled points outside its training range. This inability to generalize when using points outside its range is magnified by how much the range differs from the original one. Some improvements can be done when the range is scaled to the original one and when the local search on the coefficients is used. However, this still does not result in the same performance as before, and the original formulas are mostly not recovered. The solution to this problem is still an open question for further research. Another drawback when using the transformer model is that its hyperparameters are fixed before the training and, therefore, can not be changed for each equation. We have also manually evaluated model outputs to inspect its predictions in relation to the ground-truth. We have found that the model can predict mathematically equivalent functions using different mathematical rules.

To compare our solution to the previous approaches, we have used several benchmarks, namely the Nguyen benchmark [Uy et al., 2011], R rationals [Krawiec and Pawlak, 2013], Livermore [Mundhenk et al., 2021], Keijzer [Keijzer, 2003], Constant, and Koza [Koza, 1994]. Our model is comparable with previous approaches while maintaining the speed of transformer-based solutions and achieving high-quality results as in reinforcement-based approaches. This shows the competitiveness of the model to previous solutions.

Even though our solution has some limitations, such as the need for a fixed range, fixed vocabulary, and the maximum number of variables, it still performs quickly and accurately when used under the same conditions as during the training. Compared to previous approaches, it combines both accuracy and speed, and therefore it has a great potential to be one of the main branches of future research in symbolic regression.

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147–169, 1985.

Abien Fred Agarap. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375, 2018. URL `http://arxiv.org/abs/1803.08375`.

R. Arumugam and R. Shanmugamani. *Hands-On Natural Language Processing with Python: A practical guide to applying deep learning architectures to your NLP applications*. Packt Publishing, 2018. ISBN 9781789135916. URL `https://books.google.cz/books?id=ipplDwAAQBAJ`.

Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016. URL `http://arxiv.org/abs/1607.06450`.

Luca Biggio, Tommaso Bendinelli, Alexander Neitz, Aurélien Lucchi, and Giambattista Parascandolo. Neural symbolic regression that scales. *CoRR*, abs/2106.06427, 2021. URL `https://arxiv.org/abs/2106.06427`.

Tobias Blickle and Lothar Thiele. A Comparison of Selection Schemes Used in Evolutionary Algorithms. *Evolutionary Computation*, 4(4):361–394, 12 1996. ISSN 1063-6560. doi: 10.1162/evco.1996.4.4.361. URL `https://doi.org/10.1162/evco.1996.4.4.361`.

Jure Brence, Ljupco Todorovski, and Saso Dzeroski. Probabilistic grammars for equation discovery. *CoRR*, abs/2012.00428, 2020. URL `https://arxiv.org/abs/2012.00428`.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. URL `https://arxiv.org/abs/2005.14165`.

François Charton. Linear algebra with transformers. *CoRR*, abs/2112.01898, 2021. URL https://arxiv.org/abs/2112.01898.

Yann Chevaleyre and Jean-Daniel Zucker. Solving multiple-instance and multiple-part learning problems with decision trees and rule sets. application to the mutagenesis problem. In Eleni Stroulia and Stan Matwin, editors, *Advances in Artificial Intelligence, 14th Biennial Conference of the Canadian Society for Computational Studies of Intelligence, AI 2001, Ottawa, Canada, June 7-9, 2001, Proceedings*, volume 2056 of *Lecture Notes in Computer Science*, pages 204–214. Springer, 2001. doi: 10.1007/3-540-45153-6_20. URL https://doi.org/10.1007/3-540-45153-6_20.

Davide Chicco, Matthijs J. Warrens, and Giuseppe Jurman. The coefficient of determination r-squared is more informative than smape, mae, mape, MSE and RMSE in regression analysis evaluation. *PeerJ Comput. Sci.*, 7:e623, 2021. doi: 10.7717/peerj-cs.623. URL https://doi.org/10.7717/peerj-cs.623.

Stéphane d'Ascoli, Pierre-Alexandre Kamienny, Guillaume Lample, and François Charton. Deep symbolic regression for recurrent sequences. *CoRR*, abs/2201.04600, 2022. URL https://arxiv.org/abs/2201.04600.

Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de Las Casas, et al. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897):414–419, 2022.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.

Thomas G. Dietterich, Richard H. Lathrop, and Tomás Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89(1):31–71, 1997. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(96)00034-3. URL https://www.sciencedirect.com/science/article/pii/S0004370296000343.

Angela Fan, Mike Lewis, and Yann N. Dauphin. Hierarchical neural story generation. *CoRR*, abs/1805.04833, 2018. URL http://arxiv.org/abs/1805.04833.

Roger Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, New York, NY, USA, second edition, 1987.

Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 1848829132.

S. Glantz and B. Slinker. *Primer of Applied Regression & Analysis of Variance*. McGraw-Hill Education, 2000. ISBN 9780071360869. URL https://books.google.cz/books?id=fzV2QgAACAAJ.

Fabrício M. Gomes, Félix M. Pereira, Aneirson F. Silva, and Messias B. Silva. Multiple response optimization: Analysis of genetic programming for symbolic regression and assessment of desirability functions. *Knowledge-Based Systems*, 179:21–33, 2019. ISSN 0950-7051. doi: https://doi.org/10.1016/j.knosys.2019.05.002. URL https://www.sciencedirect.com/science/article/pii/S0950705119302096.

Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. http://www.deeplearningbook.org.

Daniel Hein, Steffen Udluft, and Thomas A. Runkler. Interpretable policies for reinforcement learning by genetic programming. *CoRR*, abs/1712.04170, 2017. URL http://arxiv.org/abs/1712.04170.

Ari Holtzman, Jan Buys, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *CoRR*, abs/1904.09751, 2019. URL http://arxiv.org/abs/1904.09751.

Peter J. Huber. Robust Estimation of a Location Parameter. *The Annals of Mathematical Statistics*, 35(1):73 – 101, 1964. doi: 10.1214/aoms/1177703732. URL https://doi.org/10.1214/aoms/1177703732.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

Mateusz Jurewicz and Leon Strømberg-Derczynski. Set-to-sequence methods in machine learning: a review. *CoRR*, abs/2103.09656, 2021. URL https://arxiv.org/abs/2103.09656.

Uday Kamath, John Liu, and James Whitaker. *Deep Learning for NLP and Speech Recognition*. Springer Publishing Company, Incorporated, 1st edition, 2019. ISBN 3030145956.

Maarten Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming*, pages 70–82, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36599-0.

Urvashi Khandelwal, Kevin Clark, Dan Jurafsky, and Lukasz Kaiser. Sample efficient text summarization using a single pre-trained transformer. *CoRR*, abs/1905.08836, 2019. URL http://arxiv.org/abs/1905.08836.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. URL http://arxiv.org/abs/1412.6980. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *CoRR*, abs/2001.04451, 2020. URL https://arxiv.org/abs/2001.04451.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.

John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0262111896.

Krzysztof Krawiec and Tomasz Pawlak. Approximating geometric crossover by semantic backpropagation. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, page 941–948, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319638. doi: 10.1145/2463372.2463483. URL https://doi.org/10.1145/2463372.2463483.

Jirí Kubalík, Jan Zegklitz, Erik Derner, and Robert Babuska. Symbolic regression methods for reinforcement learning. *CoRR*, abs/1903.09688, 2019. URL http://arxiv.org/abs/1903.09688.

Guillaume Lample and François Charton. Deep learning for symbolic mathematics. *CoRR*, abs/1912.01412, 2019. URL http://arxiv.org/abs/1912.01412.

Trang T. Le, William G. La Cava, Joseph D. Romano, John T. Gregg, Daniel J. Goldberg, Praneel Chakraborty, Natasha L. Ray, Daniel S. Himmelstein, Weixuan Fu, and Jason H. Moore. PMLB v1.0: an open source dataset collection for benchmarking machine learning methods. *CoRR*, abs/2012.00058, 2020. URL https://arxiv.org/abs/2012.00058.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R. Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer. *CoRR*, abs/1810.00825, 2018. URL http://arxiv.org/abs/1810.00825.

Ilya Loshchilov and Frank Hutter. SGDR: stochastic gradient descent with restarts. *CoRR*, abs/1608.03983, 2016. URL http://arxiv.org/abs/1608.03983.

Georg Martius and Christoph H. Lampert. Extrapolation and learning equations. *CoRR*, abs/1610.02995, 2016. URL http://arxiv.org/abs/1610.02995.

Konstantin T. Matchev, Katia Matcheva, and Alexander Roman. Analytical modelling of exoplanet transit specroscopy with dimensional analysis and symbolic regression, 2021. URL https://arxiv.org/abs/2112.11600.

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103. URL https://doi.org/10.7717/peerj-cs.103.

T. Nathan Mundhenk, Mikel Landajuela, Ruben Glatt, Cláudio P. Santiago, Daniel M. Faissol, and Brenden K. Petersen. Symbolic regression via neural-guided genetic programming population seeding. *CoRR*, abs/2111.00053, 2021. URL `https://arxiv.org/abs/2111.00053`.

Brenden K. Petersen. Deep symbolic regression: Recovering mathematical expressions from data via policy gradients. *CoRR*, abs/1912.04871, 2019. URL `http://arxiv.org/abs/1912.04871`.

Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008. ISBN 1409200736.

Charles Ruizhongtai Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *CoRR*, abs/1612.00593, 2016. URL `http://arxiv.org/abs/1612.00593`.

Hossein Izadi Rad, Ji Feng, and Hitoshi Iba. GP-RVM: genetic programing-based symbolic regression using relevance vector machine. *CoRR*, abs/1806.02502, 2018. URL `http://arxiv.org/abs/1806.02502`.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016. URL `http://arxiv.org/abs/1609.04747`.

Subham S. Sahoo, Christoph H. Lampert, and Georg Martius. Learning equations for extrapolation and control. *CoRR*, abs/1806.07259, 2018. URL `http://arxiv.org/abs/1806.07259`.

Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009. doi: 10.1126/science.1165893. URL `https://www.science.org/doi/abs/10.1126/science.1165893`.

Andrew W. Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Zídek, Alexander W. R. Nelson, Alex Bridgland, Hugo Penedones, Stig Petersen, Karen Simonyan, Steve Crossan, Pushmeet Kohli, David T. Jones, David Silver, Koray Kavukcuoglu, and Demis Hassabis. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020. doi: 10.1038/s41586-019-1923-7. URL `https://doi.org/10.1038/s41586-019-1923-7`.

Petru Soviany, Radu Tudor Ionescu, Paolo Rota, and Nicu Sebe. Curriculum learning: A survey. *CoRR*, abs/2101.10382, 2021. URL `https://arxiv.org/abs/2101.10382`.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL `http://jmlr.org/papers/v15/srivastava14a.html`.

Silviu-Marian Udrescu and Max Tegmark. Ai feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020.

Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, R. I. McKay, and Edgar Galván-Lopez. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12(2):91–119, June 2011. URL http://sc.snu.ac.kr/PAPERS/uy_gpem.pdf.

Mojtaba Valipour, Bowen You, Maysum Panju, and Ali Ghodsi. Symbolicgpt: A generative transformer model for symbolic regression. *CoRR*, abs/2106.14131, 2021. URL https://arxiv.org/abs/2106.14131.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

Digvijay Wadekar, Leander Thiele, Francisco Villaescusa-Navarro, J. Colin Hill, Miles Cranmer, David N. Spergel, Nicholas Battaglia, Daniel Anglés-Alcázar, Lars Hernquist, and Shirley Ho. Augmenting astrophysical scaling relations with machine learning : application to reducing the sz flux-mass scatter, 2022. URL https://arxiv.org/abs/2201.01305.

Edward Wagstaff, Fabian B. Fuchs, Martin Engelcke, Ingmar Posner, and Michael A. Osborne. On the limitations of representing functions on sets. *CoRR*, abs/1901.09006, 2019. URL http://arxiv.org/abs/1901.09006.

Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *CoRR*, abs/2006.04768, 2020. URL https://arxiv.org/abs/2006.04768.

Matthias Werner, Andrej Junginger, Philipp Hennig, and Georg Martius. Informed equation learning. *CoRR*, abs/2105.06331, 2021. URL https://arxiv.org/abs/2105.06331.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, may 1992. ISSN 0885-6125. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.

Casper Wilstrup and Jaan Kasak. Symbolic regression outperforms other models for small data sets. *CoRR*, abs/2103.15147, 2021. URL https://arxiv.org/abs/2103.15147.

Zhirong Wu, Shuran Song, Aditya Khosla, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets for 2.5d object recognition and next-best-view prediction. *CoRR*, abs/1406.5670, 2014a. URL http://arxiv.org/abs/1406.5670.

Zhirong Wu, Shuran Song, Aditya Khosla, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets for 2.5d object recognition and next-best-view prediction. *CoRR*, abs/1406.5670, 2014b. URL http://arxiv.org/abs/1406.5670.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J. Smola. Deep sets. *CoRR*, abs/1703.06114, 2017. URL http://arxiv.org/abs/1703.06114.

List of Figures

1.1 Example of an expression tree. 6
1.2 Transformer model architecture. Source: Vaswani et al. [2017] . . 8
1.3 Computation of Multi-Head attention. Source: Vaswani et al. [2017] 10

2.1 Initial population 17
2.2 The architecture of deep symbolic regression. Inspired by Petersen [2019] 19
2.3 The architecture of Symbolic GPT. Source: [Valipour et al., 2021] 22
2.4 The architecture of Symbolic Regression that Scales. Source: [Biggio et al., 2021] 23

3.1 Model architecture 26
3.2 Encoder architecture 27
3.3 Decoder architecture 28

4.1 The model approximates function using a polynomial. 35

5.1 The effect of encoding. 39

A.1 Examples of generated functions. 66
A.2 R^2 and relative errors for a different number of sampled points, starting from 20 points. 76
A.3 R^2 based on the distance from the original sampling range starting with 1 and ending with 95. 77
A.4 The relative error is based on the distance from the original sampling range, starting with 1 and ending with 95. 78

List of Tables

3.1 Example of function representation. 29

5.1 Effect of expression encoding. 39
5.2 Effect of number of sampled points 40
5.3 Effect of number of equations 40
5.4 Effect of a different merge operation 41
5.5 Effect of a different merge operation 41
5.6 Effect of different noise injection 42
5.7 Evaluation metrics using greedy and Top-K sampling. 42
5.8 Examples of generated expressions 43
5.9 Effect of different sampling methods, its hyperparameter, denoted as value, and its performance. The time is minutes:seconds format. 44
5.10 Comparison of the model with and without coefficients optimization. 44
5.11 Evaluation metrics using greedy search for 2D model. 45
5.12 Benchmark comparison, R^2 values are rounded to 4 decimals and time to whole seconds. 47
5.13 Effect of sampling range during the inference. 50
5.14 Comparison of different encoding. 51

A.1 Unnormalized probabilities of unary operators 63
A.2 Unnormalized probabilities of leaf values 63
A.3 Un-normalized probabilities of binary operators 64
A.4 Model vocabulary. 65
A.5 Benchmark function, $U(a, b, k)$ stands for sampling k points uniformly randomly from the range $[a, b]$, $E(a, b, k)$ denotes equidistant sampling. 68
A.6 Hyperparameters for the final model. 69
A.7 Performance of number of sampled equations for Top-K Fan et al. [2018] with $K = 20$ 70
A.8 Benchmark predictions with coefficients rounded to four decimals and simplified. The recovered column tells if the expression is semantically equivalent to the searched equation. 71
A.9 Benchmark predictions without using local optimization. Coefficients are rounded to four decimals and simplified. The recovered column tells if the expression is semantically equivalent to the searched equation. 73
A.10 Comparison of the 1D model and the 2D model on benchmark. . 74
A.11 Effect of a number of sampled points. 75

List of Algorithms

1 Beam search. Source: [Kamath et al., 2019] 11
2 Random sampling. 11
3 Random unary-binary tree generation. Source: Lample and Charton [2019] 21

List of Abbreviations

ISAB Induced Set Attention block. 14, 28, 34

MAB Multihead Attention block. 14

MSE Mean squared error. 17, 42, 43, 44

NLP Natural language processing. 6, 11

PMA Pooling by Multi-Head attention. 15, 34, 40

RE Relative error. 39, 40, 41, 42, 44, 45, 50, 51, 70, 75

rFF Row-wise feed forward layer. 14

RNN Recurrent neural network. 4, 6, 7, 18, 19, 20

SAB Set Attention block. 14, 15, 40

A. Attachments

A.1 Dataset generator unnormalized probabilities

Operation	Mathematical meaning	Unormalized probability
pow2	$(\cdot)^2$	8
pow3	$(\cdot)^3$	6
pow4	$(\cdot)^4$	4
pow5	$(\cdot)^5$	4
pow6	$(\cdot)^6$	3
inv	$(\cdot)^{-1}$	8
sqrt	$\sqrt{\cdot}$	8
exp	$\exp \cdot$	2
ln	$\ln \cdot$	4
sin	$\sin \cdot$	4
cos	$\cos \cdot$	4
tan	$\tan \cdot$	2
cot	$\cot \cdot$	2
asin	$\arcsin \cdot$	1
acos	$\arccos \cdot$	1
atan	$\arctan \cdot$	1
acot	$\text{arccot}\cdot$	1

Table A.1: Unnormalized probabilities of unary operators

Operation	Unnormalized probability
variable	20
integer in interval $[-5, 5]$ (excluding zero)	10
float in interval $[-5, 5]$	10
zero	1

Table A.2: Unnormalized probabilities of leaf values

Operation	Unnormalized probability
+	8
-	5
*	8
/	5
pow	2

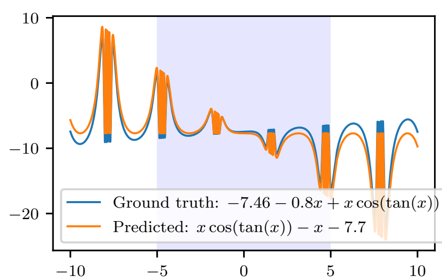
Table A.3: Un-normalized probabilities of binary operators

A.2 Model vocabulary

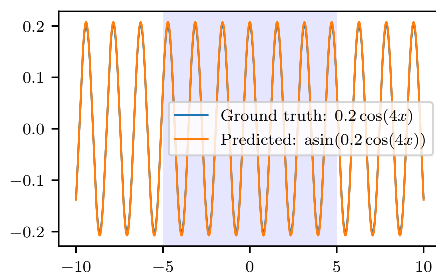
Token	Mathematical meaning
Integers from [-5, 5]	Integers from [-5, 5]
Variables	Variables
pow	a^b
+	addition
*	multiplication
sqrt	$\sqrt{\cdot}$
abs	$\lvert\cdot\rvert$
pow2	$(\cdot)^2$
pow3	$(\cdot)^3$
ln	$\ln \cdot$
exp	$\exp \cdot$
sin	$\sin \cdot$
cos	$\cos \cdot$
tan	$\tan \cdot$
cot	$\cot \cdot$
asin	$\arcsin \cdot$
acos	$\arccos \cdot$
atan	$\arctan \cdot$
acot	acot$\cdot$
neg	$(-1)(\cdot)$
C	Constants, optionally C-10, $\ldots$, C10 for exponents

Table A.4: Model vocabulary.

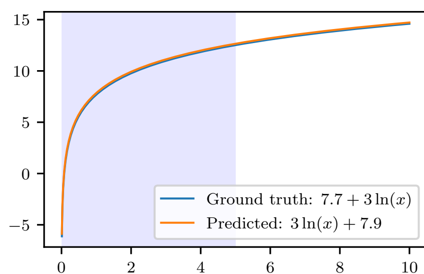
A.3 Examples of generated functions



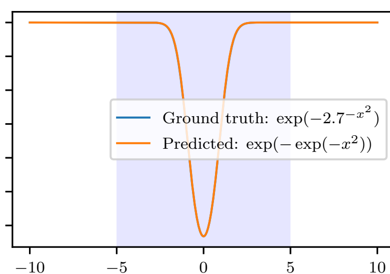
(a) Example 1.



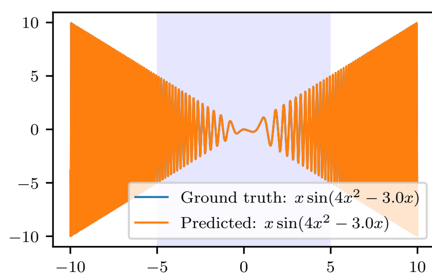
(b) Example 2.



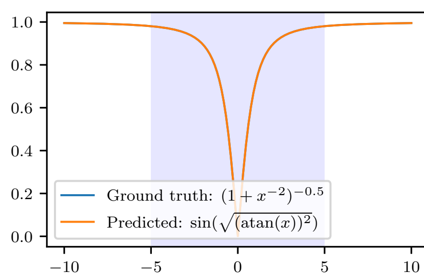
(c) Example 3.



(d) Example 4.



(e) Example 5.



(f) Example 6.

Figure A.1: Examples of generated functions.

A.4 Benchmark functions

Name	Expression	Dataset
Nguyen-1	$x^3 + x^2 + x$	$U(-1, 1, 20)$
Nguyen-2	$x^4 + x^3 + x^2 + x$	$U(-1, 1, 20)$
Nguyen-3	$x^5 + x^4 + x^3 + x^2 + x$	$U(-1, 1, 20)$
Nguyen-4	$x^6 + x^5 + x^4 + x^3 + x^2 + x$	$U(-1, 1, 20)$
Nguyen-5	$\sin(x^2)\cos(x) - 1$	$U(-1, 1, 20)$
Nguyen-6	$\sin(x) + \sin(x + x^2)$	$U(-1, 1, 20)$
Nguyen-7	$\ln(x + 1) + \ln(x^2 + 1)$	$U(0, 2, 20)$
Nguyen-8	$\sqrt{x}$	$U(0, 4, 20)$
Nguyen-9	$\sin(x) + \sin(y^2)$	$U(0, 1, 20)$
Nguyen-10	$2\sin(x)\cos(y)$	$U(0, 1, 20)$
Nguyen-11	x^y	$U(0, 1, 20)$
Nguyen-12	$x^4 - x^3 + \frac{1}{2}y^2 - y$	$U(0, 1, 20)$
R-1	$\frac{(x+1)^3}{x^2-x+1}$	$E(-1, 1, 20)$
R-2	$\frac{x^5-3x^3+1}{x^2+1}$	$E(-1, 1, 20)$
R-3	$\frac{x^6+x^5}{x^4+x^3+x^2+x+1}$	$E(-1, 1, 20)$
Livermore-1	$\frac{1}{3} + x + \sin(x^2)$	$U(-10, 10, 1000)$
Livermore-2	$\sin(x^2)\cos(x) - 2$	$U(-1, 1, 20)$
Livermore-3	$\sin(x^3)\cos(x^2) - 1$	$U(-1, 1, 20)$
Livermore-4	$\ln(x + 1) + \ln(x^2 + 1) + \ln(x)$	$U(0, 2, 20)$
Livermore-5	$x^4 - x^3 + x^2 - y$	$U(0, 1, 20)$
Livermore-6	$4x^4 + 3x^3 + 2x^2 + x$	$U(-1, 1, 20)$
Livermore-7	$\sinh(x)$	$U(-1, 1, 20)$
Livermore-8	$\cosh(x)$	$U(-1, 1, 20)$
Livermore-9	$x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x$	$U(-1, 1, 20)$
Livermore-10	$6\sin(x)\cos(y)$	$U(0, 1, 20)$
Livermore-11	$\frac{x^2 x^2}{x+y}$	$U(-1, 1, 50)$
Livermore-12	$\frac{x^5}{y^3}$	$U(-1, 1, 50)$
Livermore-13	$x^{\frac{1}{3}}$	$U(0, 4, 20)$
Livermore-14	$x^3 + x^2 + x + \sin(x) + \sin(x^2)$	$U(-1, 1, 20)$
Livermore-15	$x^{\frac{1}{5}}$	$U(0, 4, 20)$
Livermore-16	$x^{\frac{2}{5}}$	$U(0, 4, 20)$
Livermore-17	$4\sin(x)\cos(y)$	$U(0, 1, 20)$
Livermore-18	$\sin(x^2)\cos(x) - 5$	$U(-1, 1, 20)$
Livermore-19	$x^5 + x^4 + x^2 + x$	$U(-1, 1, 20)$
Livermore-20	$\exp(-x^2)$	$U(-1, 1, 20)$
Livermore-21	$x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x$	$U(-1, 1, 20)$
Livermore-22	$\exp(-0.5x^2)$	$U(-1, 1, 20)$
Koza-2	$x^5 - 2x^3 + x$	$U(-1, 1, 20)$
Koza-3	$x^6 - 2x^4 + x^2$	$U(-1, 1, 20)$
Keijzer-3	$0.3 \cdot x \cdot \sin(2 * \pi * x)$	$E(-3, 3, 0.1)$
Keijzer-4	$x^3 \exp(-x)\cos(x)\sin(x)(\sin x^2 \cos x - 1)$	$E(0.05, 10.05, 0.1)$

Keijzer-6	$\frac{x*(x+1)}{2}$	$E\,(1, 120, 1)$
Keijzer-7	$\ln x$	$E\,(1, 100, 0.1)$
Keijzer-8	$\sqrt{x}$	$E\,(1, 100, 0.1)$
Keijzer-9	$\ln\,(x + \sqrt{(x^2+1)})$	$E\,(1, 100, 0.1)$
Keijzer-10	x^y	$E\,(0, 1, 0.01)$
Keijzer-11	$xy + \sin\,((x-1)(y-1))$	$E\,(-3, 3, 0.01)$
Keijzer-12	$x^4 - x^3 + \frac{y^2}{2} - y$	$E\,(-3, 3, 0.01)$
Keijzer-13	$6\sin\,(x)\cdot\cos\,(y)$	$E\,(-3, 3, 0.01)$
Keijzer-14	$\frac{8}{2+x^2+y^2}$	$E\,(-3, 3, 0.01)$
Keijzer-15	$\frac{x^3}{5} + \frac{y^3}{2} - y - x$	$E\,(-3, 3, 0.01)$
Constant-1	$3.39x^3 + 2.12x^2 + 1.78x$	$U\,(-1, 1, 20)$
Constant-2	$\sin x^2 \cdot \cos x - 0.75$	$U\,(-1, 1, 20)$
Constant-3	$\sin\,(1.5x)\cdot\cos\,(0.5y))$	$U\,(0, 1, 20)$
Constant-4	$2.7x^y$	$U\,(0, 1, 20)$
Constant-5	$\sqrt{1.23x}$	$U\,(0, 4, 20)$
Constant-6	$x^{0.426}$	$U\,(0, 4, 20)$
Constant-7	$2\sin\,(1.3x)\cdot\cos y$	$U\,(0, 1, 20)$
Constant-8	$\ln(x+1.4) + \ln(x^2+1.3)$	$U\,(0, 2, 20)$

Table A.5: Benchmark function, $U(a, b, k)$ stands for sampling k points uniformly randomly from the range $[a, b]$, $E(a, b, k)$ denotes equidistant sampling.

A.5 Final hyperparameters

Encoder	
Number of row wise FF	2
Number of heads	12
Number of layers	4
Dimension of model	384
Dimension of the first FF layer	1536
Dimension of the second FF layer	384
Number of inducing points	64
Number of seed vectors	32
Dropout rate	0.1
Decoder	
Dimension of model	512
Number of heads	12
Dimension of FF layer	2048
Number of layers	4
Dropout rate	0.1
Vocabulary size	54
Dataset	
Number of equations	130 million
Number of sampled points	100
Training	
Number of epochs	390
Starting σ^2 noise	0.1
Ending regression λ	0.1

Table A.6: Hyperparameters for the final model.

A.6 Effect of number of sampled equations

Number of equations	$R^2 \uparrow$	RE interp. $\downarrow$	Time (hh:mm:ss)
16	0.998256	0.066528	00:04:46
32	0.998532	0.053106	00:04:36
64	0.999068	0.046484	00:06:46
128	0.999336	0.037968	00:07:19
256	0.999449	0.030616	00:19:04
512	0.999702	0.033389	00:30:05
1024	0.999826	0.022776	01:05:00
2048	0.999902	0.023793	01:03:57
4096	0.999942	0.017102	04:08:42
8192	0.999976	0.013216	06:15:53
16384	0.999985	0.008698	11:58:18

Table A.7: Performance of number of sampled equations for Top-K Fan et al. [2018] with $K = 20$

A.7 Full predictions

Name	Exact	Prediction
Nguyen-1	✓	$x^3 + \sqrt{x^4} + x$
Nguyen-2	✓	$x^3 + x^2 + x + x^4$
Nguyen-3	✓	$x^3 + x^2 + x + x^4 + x^5$
Nguyen-4		$0.2811^{x^2} + x^3 + x^2 + x + x^5 + \lvert x\rvert^{6.0298}$
Nguyen-5	✓	$\sin(x^2)\cos(x) - 1$
Nguyen-6	✓	$\sin(x) + \sin(x + x^2)$
Nguyen-7		$\ln(x^3 + x^2 + \sqrt{x^2} + 1.0)$
Nguyen-8	✓	$\frac{x}{\sqrt{x}}$
Nguyen-9	✓	$\ln(\exp(\sin(x))) + \sin(y^2)$
Nguyen-10	✓	$2\sin(x)\cos(y)$
Nguyen-11	✓	$\frac{1}{x^{-y}}$
Nguyen-12		$-x^3 + x^4 - y + (y^2)^{0.7395} - 0.0015$
R-1	✓	$x^{-2} + 4 + \frac{3}{x}$
R-2		$\frac{-3x^3}{x^2+1} + \frac{\cos(x)}{x^2+1.0} + \frac{x^5}{x^2+1} + \frac{1}{x^2+1}$
R-3		$\frac{x^5}{x^3+x+0.1616+x^{-1}}$
Livermore-1	✓	$x + \sin(\ln(\exp(x))^2) - 0.3333$
Livermore-2	✓	$\sin(x^2)\cos(x) - 2$
Livermore-3	✓	$-0.99834 + \cos(6.281 - 0.0011x + x^2)\sin(x^3)$
Livermore-4		$\ln(x^3 + x^2 + x + x^4)$
Livermore-5	✓	$-x^3 + x^4 - y + \ln(\exp(x^2))$
Livermore-6	✓	$3x^3 + 2x^2 + x + 4x^4$
Livermore-7		$x\exp(0.2072x(\operatorname{atan}(x))^3)$

Livermore-8		$\exp(0.6259 \cdot x \cdot \mathrm{atan}(x))$
Livermore-9		$\frac{x^2}{x+1.9673} + \frac{2.1942x}{x+1.9794} + \frac{3.4366\lvert x\rvert^{6.4462}}{x+1.9318} + \frac{2.2568\lvert x\rvert^{9.4658}}{x-1.0625}$
Livermore-10	✓	$6\sin(x)\cos(y)$
Livermore-11	✓	$\frac{x^2y^2}{x+y}$
Livermore-12	✓	$\frac{x^5}{y^3}$
Livermore-13	✓	$3.3296^{0.2778\ln(x)}$
Livermore-14	✓	$x^3 + x^2 + x + \sin(x) + \sin(y^2)$
Livermore-15	✓	$\ln(\exp(x))^{0.2001}$
Livermore-16	✓	$1.9477^{\ln(x)}$
Livermore-17	✓	$4\sin(x)\cos(y)$
Livermore-18	✓	$\sin(x^2)\cos(x) - 5$
Livermore-19	✓	$x^2 + x + x^4 + x^5$
Livermore-20	✓	$(\exp(-x))^x$
Livermore-21		$2.9x + \lvert x\rvert^4(\mathrm{acot}(x))^2 + 1.5615\lvert x\rvert^{6.9}\mathrm{acot}(x) + \lvert x\rvert^8$
Livermore-22	✓	$\sqrt{\exp(-x^2)}$
Koza-2	✓	$-2x^3 + x + x^5$
Koza-3		$x^2 - 2.0011\lvert x\rvert^{4.001} + \lvert x\rvert^{6.0005} + 0.0027$
Keijzer-3	✓	$0.3 \cdot x \cdot \sin(6.2832x - 6.2832)$
Keijzer-4		$2.6612x^2\exp(-x)\sin(1.9876x)$
Keijzer-6	✓	$\ln(\exp(0.49914x)\exp(0.5x^2))$
Keijzer-7	✓	$\ln x$
Keijzer-8	✓	$\sqrt{x}$
Keijzer-9		$\frac{x}{0.4136x^2+1.0021)^{0.3173}}$
Keijzer-10	✓	$\frac{1}{x^{-y}}$
Keijzer-11	✓	$xy - \sin(-xy + x + y - 1)$
Keijzer-12		$-x^3 + x^4 - 0.01835y^3 + 0.4286y^2 - y + 0.0012y^4$
Keijzer-13	✓	$4.7063\sqrt{1.6253}\sin(x)\cos(y)$
Keijzer-14		$\sqrt{y^4} + \frac{4}{(0.455x^2+1)}$
Keijzer-15	✓	$0.2048x^3 - x + 0.4973y^3 - y$
Constant-1	✓	$3.3881x^3 + 2x^2 + 1.7031x + 1$
Constant-2	✓	$\sin(x^2)\cos(x) - 0.75$
Constant-3	✓	$\sin(1.5x)\cos(0.5y)$
Constant-4		$\cot(\frac{0.371}{x^y})$
Constant-5	✓	$\ln(\exp(\sqrt{1.23}\sqrt{x}))$
Constant-6	✓	$1.5311^{\ln(x)}$
Constant-7	✓	$2\sin(\ln(3.6693^x))\cos(y)$
Constant-8		$1.9671\ln(x^{1.5567} + 1.3927)$

Table A.8: Benchmark predictions with coefficients rounded to four decimals and simplified. The recovered column tells if the expression is semantically equivalent to the searched equation.

A.8 Benchmark results without local optimization

Name	Skeleton found	R^2
Nguyen-1	✓	1
Nguyen-2	✓	1
Nguyen-3	✓	1
Nguyen-4	✓	0.9991
Nguyen-5	✓	1
Nguyen-6	✓	1
Nguyen-7		1
Nguyen-8	✓	1
Nguyen-9	✓	1
Nguyen-10	✓	1
Nguyen-11	✓	1
Nguyen-12		1
R-1	✓	1
R-2		0.9996
R-3		0.9990
Livermore-1	✓	1
Livermore-2	✓	1
Livermore-3	✓	1
Livermore-4		1
Livermore-5	✓	1
Livermore-6	✓	1
Livermore-7		0.9993
Livermore-8		0.9995
Livermore-9		0.9869
Livermore-10	✓	0.9911
Livermore-11	✓	1
Livermore-12	✓	1
Livermore-13	✓	0.9982
Livermore-14		0.9996
Livermore-15	✓	1
Livermore-16	✓	0.9983
Livermore-17	✓	1
Livermore-18	✓	1
Livermore-19	✓	1
Livermore-20	✓	1
Livermore-21		0.9922
Livermore-22	✓	1
Koza-2	✓	1
Koza-3		0.9990
Keijzer-3		0.9400
Keijzer-4		0.9564

Keijzer-6	✓	0.9998
Keijzer-7	✓	1
Keijzer-8	✓	1
Keijzer-9		0.9990
Keijzer-10	✓	1
Keijzer-11	✓	0.9993
Keijzer-12		1
Keijzer-13	✓	0.9992
Keijzer-14		0.9998
Keijzer-15	✓	0.9973
Constant-1	✓	1
Constant-2	✓	0.9879
Constant-3	✓	0.8987
Constant-4	✓	1
Constant-5		0.9992
Constant-6	✓	1
Constant-7	✓	0.7340
Constant-8		0.9997

Table A.9: Benchmark predictions without using local optimization. Coefficients are rounded to four decimals and simplified. The recovered column tells if the expression is semantically equivalent to the searched equation.

A.9 Comparison of 1D and 2D model

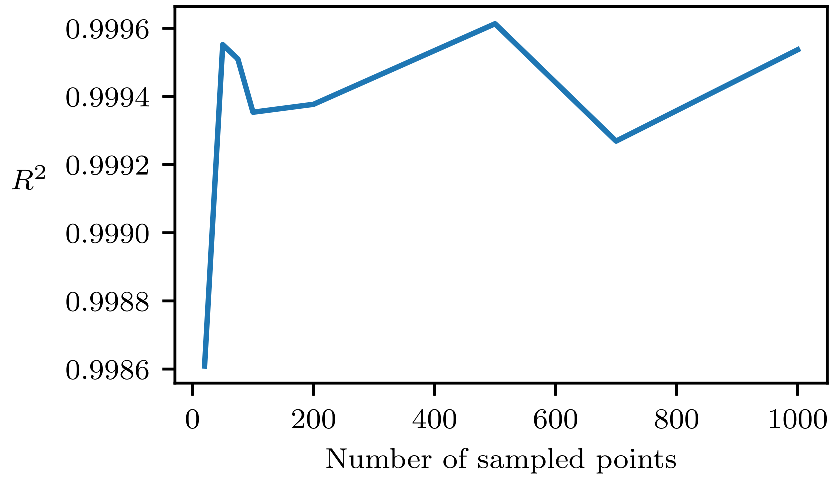
Name	R^2 of 1D model	R^2 of 2D model
Nguyen-1	1	1
Nguyen-2	1	1
Nguyen-3	1	1
Nguyen-4	0.9998	0.9995
Nguyen-5	1	1
Nguyen-6	1	1
Nguyen-7	1	1
Nguyen-8	1	1
R-1	1	1
R-2	1	0.9999
R-3	0.9999	0.9995
Livermore-1	1	1
Livermore-2	1	1
Livermore-3	1	1
Livermore-4	1	1
Livermore-6	1	1
Livermore-7	1	1
Livermore-8	0.9998	0.9999
Livermore-9	0.9991	0.9995
Livermore-13	1	1
Livermore-15	1	1
Livermore-16	1	1
Livermore-18	1	1
Livermore-19	1	1
Livermore-20	1	1
Livermore-21	0.9999	0.9997
Livermore-22	1	1
Koza-2	1	1
Koza-3	1	1
Keijzer-3	1	1
Keijzer-4	0.9927	0.9893
Keijzer-6	1	1
Keijzer-7	1	1
Keijzer-8	1	1
Keijzer-9	1	1
Constant-1	1	1
Constant-2	1	1
Constant-5	1	1
Constant-6	1	1
Constant-8	1	1

Table A.10: Comparison of the 1D model and the 2D model on benchmark.

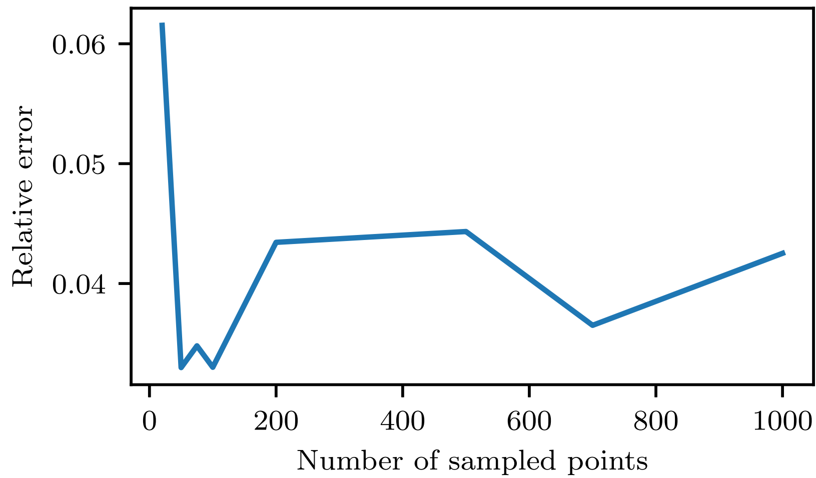
A.10 Effect of number of sampled points during inference

Number of points	$R^2 \uparrow$	RE interp. $\downarrow$
10	0.985261	0.253189
20	0.998609	0.061538
50	0.999552	0.032989
75	0.999510	0.034804
100	0.999354	0.033015
200	0.999377	0.043438
500	0.999613	0.044338
700	0.999269	0.036514
1000	0.999537	0.042521

Table A.11: Effect of a number of sampled points.



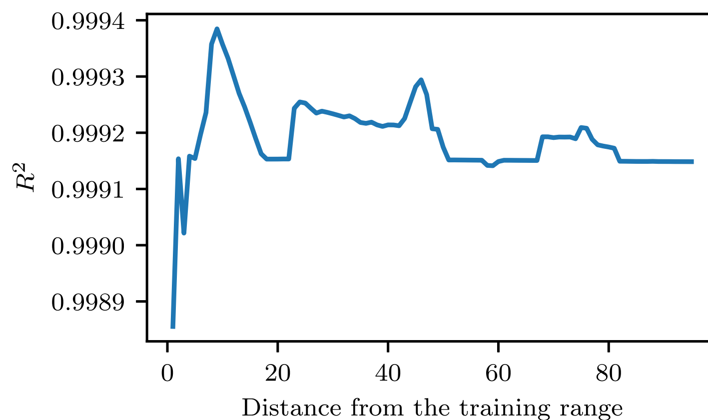
(a) R^2 based on the number of sampled points, starting from 20 points.



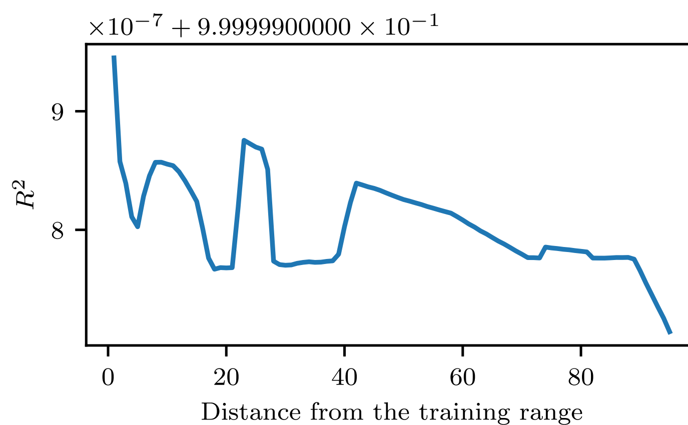
(b) The relative error is based on the number of sampled points, starting from 20 points.

Figure A.2: R^2 and relative errors for a different number of sampled points, starting from 20 points.

A.11 Model ability to perform outside of sampling range

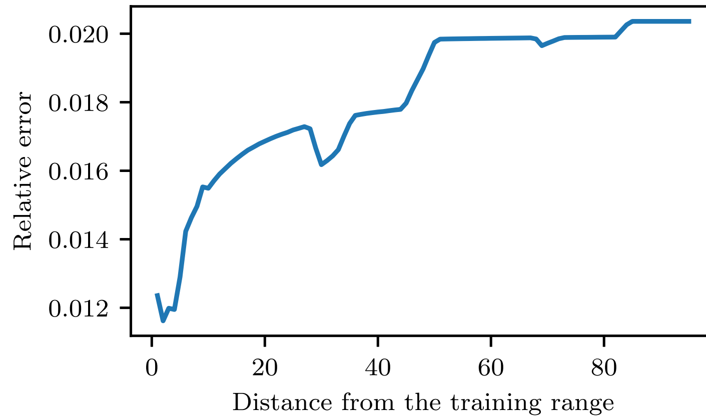


(a) R^2 is based on the distance from the original sampling range without coefficient optimization.

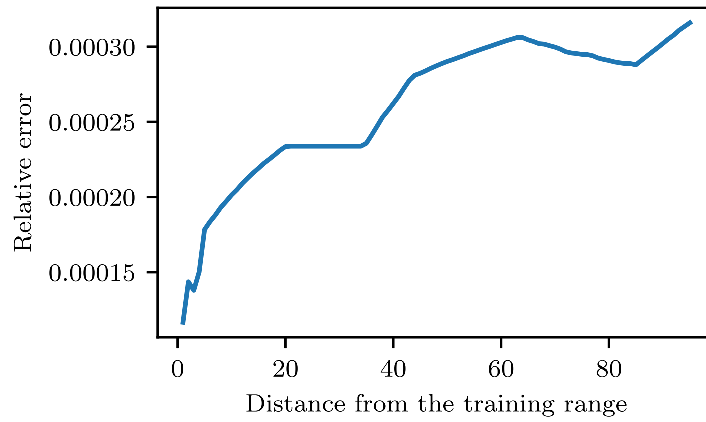


(b) R^2 is based on the distance from the original sampling range with coefficient optimization.

Figure A.3: R^2 based on the distance from the original sampling range starting with 1 and ending with 95.



(a) The relative error is based on the distance from the original sampling range without coefficient optimization.



(b) The relative error is based on the distance from the original sampling range with coefficient optimization.

Figure A.4: The relative error is based on the distance from the original sampling range, starting with 1 and ending with 95.