
SymFormer: End-to-end symbolic regression using transformer-based architecture

Martin Vastl^{1,2} Jonáš Kulhánek^{1,3} Jiří Kubalík¹ Erik Derner^{1,3}
Robert Babuška^{1,4}

¹ Czech Institute of Informatics, Robotics and Cybernetics,
Czech Technical University in Prague
{vastlmar, jonas.kulhanek, jiri.kubalik, erik.derner}@cvut.cz
² Faculty of Mathematics and Physics, Charles University
³ Faculty of Electrical Engineering, Czech Technical University in Prague
⁴ Cognitive Robotics Faculty of 3mE, Delft University of Technology
r.babuska@tudelft.nl

Abstract

Many real-world problems can be naturally described by mathematical formulas. The task of finding formulas from a set of observed inputs and outputs is called symbolic regression. Recently, neural networks have been applied to symbolic regression, among which the transformer-based ones seem to be the most promising. After training the transformer on a large number of formulas (in the order of days), the actual inference, i.e., finding a formula for new, unseen data, is very fast (in the order of seconds). This is considerably faster than state-of-the-art evolutionary methods. The main drawback of transformers is that they generate formulas without numerical constants, which have to be optimized separately, so yielding suboptimal results. We propose a transformer-based approach called SymFormer, which predicts the formula by outputting the individual symbols and the corresponding constants simultaneously. This leads to better performance in terms of fitting the available data. In addition, the constants provided by SymFormer serve as a good starting point for subsequent tuning via gradient descent to further improve the performance. We show on a set of benchmarks that SymFormer outperforms two state-of-the-art methods while having faster inference.

1 Introduction

Many natural processes and technical systems can be described by mathematical formulas. Knowing the correct formula would not only provide us with some insight into the process's inner workings, but it would also allow us to predict how the process will evolve in the future. Therefore, being able to automatically derive a simple formula that fits the observed data would have a tremendous impact on applications in all areas of science. The task of finding such a formula from the observed data is called *symbolic regression* (SR) and has already been applied to a variety of real-world problems, e.g., in physics [Wadekar et al., 2022, Matchev et al., 2021], robotics [Kubalík et al., 2019, Hein et al., 2017], or machine learning [Wilstrup and Kasak, 2021].

Historically, most of the symbolic regression methods [Schmidt and Lipson, 2009b, Kubalík et al., 2020] were tackled by means of genetic programming [Koza, 1992, Schmidt and Lipson, 2009a, Staelens et al., 2013, Arnaldo et al., 2015, Bładek and Krawiec, 2019]. Unfortunately, they have to be carefully designed for each problem. Also, predicting formulas was slow and computationally expensive. For a single problem, an entire population of formulas had to be evolved and evaluated

repeatedly through many generations. In recent years neural approaches emerged [Petersen, 2019, Mundhenk et al., 2021] to tackle these problems. They were trained end-to-end with the sampled points as the input and the symbolic representation of the formula as the output. However, these methods are still trained from scratch for each formula. Although they work well for simple formulas, they are impractically slow and inefficient for more complex formulas due to their reliance on reinforcement learning. To increase the efficiency, fully supervised approaches were proposed [Biggio et al., 2021, Valipour et al., 2021], that train a transformer model on a large collection of formulas. The formula is autoregressively generated by predicting each symbol conditioned on previously decoded symbols. The generated expression is decoded without constants, i.e., all constants were replaced by a special symbol, and the concrete values were found using global optimization. We argue that the concrete values of constants have a large impact on the generated function, and without predicting them, the model will not learn to represent the data well. For example, we can imagine a simple model returning a sum of sine and cosine functions with increasingly higher frequencies. Since every function can be expressed using the Fourier basis, by changing the constants, we are able to represent most functions with low error.

Inspired by d’Ascoli et al. [2022], where a similar idea was applied to the problem of recurrent sequences, we propose a novel approach called the SymFormer that generates the concrete values for constants alongside the symbolic representation of the formula. More specifically, we introduce the following contributions:

- We design a transformer-based architecture trained end-to-end on a large set of formulas consisting of hundreds of millions of formulas.
- The model generates both the symbolic representation of the formula and the concrete values for all constants at the same time. This allows the symbolic decoder to condition on the generated constants and it improves the quality of the symbolic representation.
- We also use the generated constants to initialize the local gradient search to fine-tune the final constants effectively and reliably.
- Our approach was thoroughly evaluated and compared to relevant methods. Also, we validate our design choices in an ablation study.
- The source code and the pre-trained model checkpoints are publicly available ¹.

2 Related work

Genetic Programming approaches are a traditional way of solving SR [Koza, 1992]. Genetic programming evolves expressions encoded as trees using selection, crossover, and mutation. A limitation of the genetic algorithm-based approaches is that they are sensitive to the choice of hyperparameters [Petersen, 2019]. They need to evolve each equation from scratch, which is slow, and the models tend to increase in complexity without much performance improvement. It is also problematic to tune expression constants only by using genetic operators.

Neural Network approaches can be generally divided into three categories. The first one is approaches based on Equation learner (EQL) [Martius and Lampert, 2016, Sahoo et al., 2018, Werner et al., 2021]. The idea behind EQL is to find function $f(x) = y$ by training a neural network on x as input and y values as output while using as few network weights as possible. As activation functions, elementary functions (sin, log, ...) are used, and after the training, they are read from the network with corresponding weights. A limitation of such an approach is that they require special handling of functions that are not defined over the whole \mathbb{R} (e.g., log), that the depth of the network limits the complexity of the predicted equation. Lastly, they can be slow since they need to find each equation from scratch.

The second approach is based on training a recurrent neural network (RNN) using reinforcement learning [Petersen, 2019]. The idea is to let the RNN generate the equation and then calculate the reward function as an error between the ground truth $f(x)$ values and the values from the predicted function $\hat{f}(x)$. An interesting extension is proposed by Mundhenk et al. [2021], where they sample from the RNN, and the output is then taken as an initial population for a genetic algorithm. The limitations of both of these approaches are that the model does not predict the constants, and therefore

¹<https://github.com/vastlik/symformer>

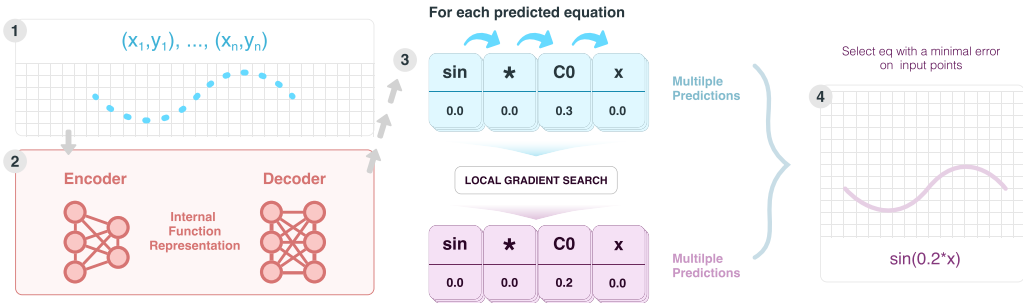


Figure 1: Schematic diagram of inference. The input points are passed through the transformer, generating several candidate equations using Top-K sampling. These candidates are further improved using gradient descent. The final equation is then selected by the lowest mean squared error.

they have to be found through global optimization in postprocessing steps which slows down the whole training loop [Mundhenk et al., 2021, Petersen, 2019].

The transformer-based approach is proposed by Valipour et al. [2021], Biggio et al. [2021], d’Ascoli et al. [2022], where they first generate a large amount of training data and train a transformer [Vaswani et al., 2017] model in a supervised manner. Valipour et al. [2021] train a GPT-2 [Radford et al., 2019] model on pairs of points and symbolic output. Then they use global optimization to find the constants for each equation. Biggio et al. [2021] uses the encoder from the Set transformer Lee et al. [2018] and the decoder from original transformer architecture [Vaswani et al., 2017]. Similar to Valipour et al. [2021], they train the models only on skeletons (expression without the constants), and afterward, they fit the constants using global optimization. Another extension is introduced by d’Ascoli et al. [2022], where they train the transformer model [Vaswani et al., 2017] on recurrent sequences. They predict the expression constants jointly by encoding them into the symbolic output. To encode integers, they use their base b representation e.g., for $x = -325$ and base $b = 30$ the representation would be $[-, 10, 25]$. In the case of floats, they use the IEEE 754 float representation and round the mantissa to the four most significant digits. They also introduce new tokens representing exponents. For example if we have number -0.015 , then they encode it as $[-, 15, e-4]$. The disadvantage of this approach is that the mantissa has only finite precision. Therefore, the model typically only predicts the largest terms when approximating complicated functions [d’Ascoli et al., 2022].

3 Method

In symbolic regression, it is assumed that there is an unknown function f and that we observe its output on a finite set of input points. The goal is then to find the mathematical formula of this function. Therefore, we want to find a function \hat{f} such that the squared difference between the function’s output on the input points and the outputs of the unknown function f is minimized.

Given a set of observed input-output pairs, our model generates the symbolic representation of the formula together with values of all constants present in the formula in a single forward pass of a neural network. This is visualized in Figure 1. First, we transform all input-output pairs using an encoder block to obtain an internal representation. Given this representation, the decoder then autoregressively generates individual symbols and corresponding constants. This means that in order to generate the next token in the symbolic representation of the function, we pass all previously generated tokens and constants to the decoder. We proceed in this fashion until we obtain the entire formula. During inference, we sample multiple predictions from the model and fine-tune all constants to minimize the error between the predicted formula’s outputs and the observed outputs. Finally, we select the prediction with the lowest error.

3.1 Model architecture

Our neural network is an autoregressive transformer-based architecture that contains an encoder with cross-attention blocks [Lee et al., 2018, Jaegle et al., 2021] and a simpler self-attention decoder

[Vaswani et al., 2017]. The input to the encoder consists of the data points, which are first passed through a trainable affine layer to project them into a latent space. The resulting vectors are then passed through several induced set attention blocks [Lee et al., 2018], which are two cross attention layers. First, cross attention uses a set of trainable vectors as the queries and the input features as keys and values. Its output is used as the keys and values for the second cross attention, and the original input vectors are used as the queries. After these cross attention layers, we add a dropout layer [Srivastava et al., 2014]. In the end, we compute cross attention between a set of trainable vectors (queries) to fix the size of the output such that it does not depend on the number of input points. This final representation is then passed to the decoder.

The decoder autoregressively generates the symbolic representation and the constants given the encoder’s representation. The input symbols are first passed through the embedding layer and then pairwise summed with trainable positional encoding vectors. These newly formed vectors are pairwise concatenated with affine-projected constants. The resulting vectors are then passed through several decoder layers [Vaswani et al., 2017]. The decoder has two heads, where the first one is a classification head which predicts the probability distribution over the next symbol in the sequence. If the predicted symbol is a constant, the other (regression) head outputs its value. Training can efficiently process each sequence in a single forward pass of the network thanks to the masked attention and teacher forcing [Vaswani et al., 2017].

3.2 Training & inference

We train the model using cross-entropy \mathcal{L}_{class} for the symbolic expression and mean squared error \mathcal{L}_{MSE} for the constants:

$$\mathcal{L} = \mathcal{L}_{class} + \lambda \mathcal{L}_{MSE}, \tag{1}$$

where λ is a hyperparameter. If λ is too small, the model will not learn to predict the constants at all, and if λ is too large, the model will not learn to predict the symbolic output well, and therefore, the constants will be useless. Therefore, at the beginning of the training, we set λ to zero, and after a few epochs, we gradually increase it using the cosine schedule [Loshchilov and Hutter, 2016]. Note that we calculate the regression loss only at indices where the model should predict a constant. We have also found it beneficial to add a small random noise sampled from $\mathcal{N}(0, \sigma^2)$ to the constants during the training since, during inference, the constants are not always precise. Parameter σ is decreased according to the cosine schedule.

During inference, we use Top-K sampling [Fan et al., 2018] to generate candidate equations. Then, we fix the symbolic expression and run gradient descent on all constants. We use the mean squared error between the predicted function’s outputs and the outputs of the ground-truth function. Finally, we select the equation with the lowest error on the input points.

3.3 Dataset generation

We generate two training datasets, one with 130 million equations containing only univariate functions and the second one with 100 million functions containing bivariate functions, by following the same algorithm as described by Lample and Charton [2019] with the maximum of ten operators. The algorithm starts by generating a random unary-binary tree and filling the nodes with appropriate operations. The unnormalized probabilities of each operation and operator and the hyperparameters of the generator are given in *Supp. Mat.* In our dataset, we have also introduced new operators such as $\text{pow}2(\cdot) = (\cdot)^2$, $\text{pow}3(\cdot) = (\cdot)^3$, ... to make it easier to represent them. The generated expressions are then simplified using SymPy [Meurer et al., 2017]. We discard the expressions that cannot be simplified in 5 seconds. Finally, we sample uniformly at random 100 points (200 for bivariate functions) from the interval $[-5, 5]$, and if there are any non-finite values (NaN or $\pm\infty$), we try $(0, 5]$ and then $[-5, 0)$ (similarly for the bivariate dataset). The reason for selecting these intervals is that functions such as $\log x$, \sqrt{x} or $\log -x$ are not defined on the full interval. Furthermore, we ignore any equations with values on the sampled points larger than 10^7 (in absolute value). Similarly, we discard equations with constants smaller than 10^{-10} or larger than 10^{10} in absolute value. We also ignore any linear functions created by the simplification process. We do not want to keep all the linear functions because the dataset would be biased towards linear functions. Finally, we throw away any constant functions and functions that contain more than 50 symbolic tokens.

3.4 Expression encoding

We use preorder tree representation to encode expressions and replace constants with special symbols. The constants are encoded using a scientific-like notation where a constant C is represented as a tuple of the exponent c_e and the mantissa c_m :

$$C \approx c_m \cdot 10^{c_e}, \quad c_e = \lceil \log_{10} C \rceil, \quad c_m = \frac{C}{10^{c_e}}. \quad (2)$$

In this representation, the mantissa is in the range $[-1, 1]$, and the exponent is an integer. For example the expression $0.017 \cdot x + 1781.5$ will have symbols $[+, \text{mul}, x, \text{C-1}, \text{C4}]$ and constants $[0, 0, 0, 0.17, 0.17815]$. To further help the model represent constants, we add all integers from interval $[-5, 5]$ into the model vocabulary. Different encodings are compared in Section 4.7. In contrast to d’Ascoli et al. [2022], who are able to express constants only up to four most significant digits, our approach achieves full float precision.

4 Experiments

This section describes our training setting and the metrics that we used to demonstrate the model’s ability to predict the formulas and compare our model to previous approaches. We also show how the SymFormer generalizes to two dimensions and outside of the known range. Furthermore, we manually inspect the model’s predictions to examine different equivalent mathematical formulas that the SymFormer found. In the end, we compare different encodings and their impact on the model’s performance. In our experiments, we refer to the model trained only on univariate functions as the Univariate SymFormer and the model trained on both the univariate and bivariate functions as the Bivariate SymFormer. We also always use a local gradient search on the constants if not stated otherwise.

4.1 Training

We train our model using the Adam optimizer [Kingma and Ba, 2014] for 3 epochs on 8 NVIDIA A100 GPUs. The training of the model takes roughly 33 hours. We use 130 million univariate equations for the training set and 10 000 for the validation set. Furthermore, we randomly selected 256 equations to calculate the metrics using the beam search. We use a training schedule similar to the original transformer [Vaswani et al., 2017]. However, we divide the learning rate by five since the training often diverged when using the original learning rate. The regression λ is set according to the cosine schedule and delayed for 97 700 gradient steps, reaching 1.0 at the end of the training.² For the random noise, we sample from $\mathcal{N}(0, \epsilon)$, where ϵ is initially set to 0.1 and decreased to zero during training using the same schedule. The complete set of hyperparameters for the model containing approximately 95 million parameters can be seen in *Supp. Mat.* The hyperparameters were found empirically. We use the same settings for our Bivariate SymFormer.

4.2 Metrics

To assess the quality of the model, we have selected two metrics: the relative error and the coefficient of determination (R^2). The relative error is the average absolute difference between the predicted value and the ground truth divided by the absolute value of the ground truth:

$$\text{RE}(y, \hat{y}) = \frac{1}{k} \sum_{i=1}^k \left| \frac{y_i - \hat{y}_i}{y_i} \right|, \quad (3)$$

where y_i and \hat{y}_i are the ground-truth and predicted values for point i , respectively. The coefficient of determination (R^2) [Glantz and Slinker, 2000] is defined as follows:

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^k (y_i - \hat{y}_i)^2}{\sum_{i=1}^k (y_i - \bar{y})^2}, \quad (4)$$

²The regression λ and the learning rate were updated every $10^6/1024 \approx 977$ gradient steps (batch size was 1 024).

Table 1: Results comparing the **SymFormer** with state-of-the-art methods on several benchmarks. The **SymFormer** uses Top-K sampling with $K = 20$ while generating 1024 samples and improving them using local gradient search with early stopping. We report R^2 and the average time to generate an equation in seconds.

Benchmark	SymFormer		NSRS [Biggio et al., 2021]		DSO [Mundhenk et al., 2021]	
	$R^2 \uparrow$	Time (s) \downarrow	$R^2 \uparrow$	Time (s) \downarrow	$R^2 \uparrow$	Time (s) \downarrow
Nguyen	0.99998	47.50	0.96744	169.46	0.99297	140.25
R	0.99986	94.33	1.00000	95.67	0.97488	855.33
Livermore	0.99996	43.00	0.88551	193.09	0.99651	276.32
Koza	1.00000	101.00	0.99999	111.50	1.00000	217.50
Keijzer	0.99904	48.67	0.97392	255.50	0.95302	3929.50
Constant	0.99998	90.88	0.88742	230.38	1.00000	2816.19
Overall avg.	0.99978	52.95	0.92901	199.63	0.99443	326.53

where y_i and \hat{y}_i are the ground-truth and predicted values for point i , respectively. \bar{y} is the average of y_i over all the points. The advantage of using R^2 is its nice interpretation. If $R^2 > 0$, then our prediction is better than the predicting just the average value and if $R^2 = 1$, then we have a perfect model.

4.3 In-domain performance

To demonstrate the SymFormer’s ability to predict the formulas successfully, we use Top-K sampling [Fan et al., 2018] with $K = 16$ and 256 samples to generate the best equation. In our experiments, we report the median values of all metrics since the mean can be skewed by outliers. In Figure 2, we plot some of the model predictions. The Univariate SymFormer achieved an R^2 of 0.9995 and a relative error of 0.0288. Furthermore, when we used the local gradient search, the model improved to R^2 1.0000 and a relative error of 0.0010. The Bivariate SymFormer achieved an R^2 of 0.9996 and a relative error of 0.0389 using Top-K [Fan et al., 2018] with $K = 16$ and 1.0000 R^2 and a relative error of 0.0035 when the local gradient search was used. This demonstrates the model’s ability to generalize to higher dimensions.

4.4 Comparison to previous approaches

To compare our results to current state-of-the-art approaches, we use the Nguyen benchmark [Uy et al., 2011], R rationals [Krawiec and Pawlak, 2013], Livermore, [Petersen, 2019], Keijzer [Keijzer, 2003], Constant, and Koza [Koza, 1994]. The complete benchmark functions are given in *Supp. Mat.* Unfortunately, these benchmarks are better suited for methods where parameters such as the number of variables, set of symbols, or sampling range are set specifically to match the problem at hand. In our method, these parameters are fixed in the beginning and cannot be changed later. Note that it is difficult to make a completely fair comparison on the benchmarks for two reasons. The first one is that some methods use a restricted vocabulary and thus have a smaller search space giving them an advantage over our method. The second problem arises from the different sampling ranges and the number of sampled points.

We use Top-K sampling with $K = 20$ and 1 024 samples with early stopping for the benchmark. From the results in Table 1 we can see that the SymFormer method is competitive in terms of the model performance on all of the benchmarks while outperforming both NSRS [Biggio et al., 2021] and DSO [Mundhenk et al., 2021] in the time required to find the equation. One of the observations that we have found is that the model sometimes predicts semantically the same expression as the ground-truth, but using a more complex expression, *e.g.*, in one case model had to predict $\frac{x^2+x}{2}$, but predicted $\ln(\exp(0.5x^2)\exp(0.5x))$, which is same after simplification, but unreasonably complex. This was likely caused by the distribution of our dataset.

Furthermore, to demonstrate the Bivariate SymFormer performance on both the univariate and bivariate functions, we evaluated a single model on all univariate and bivariate functions using the same benchmark. Note that the benchmark functions are mostly univariate. In Table 2, we can notice

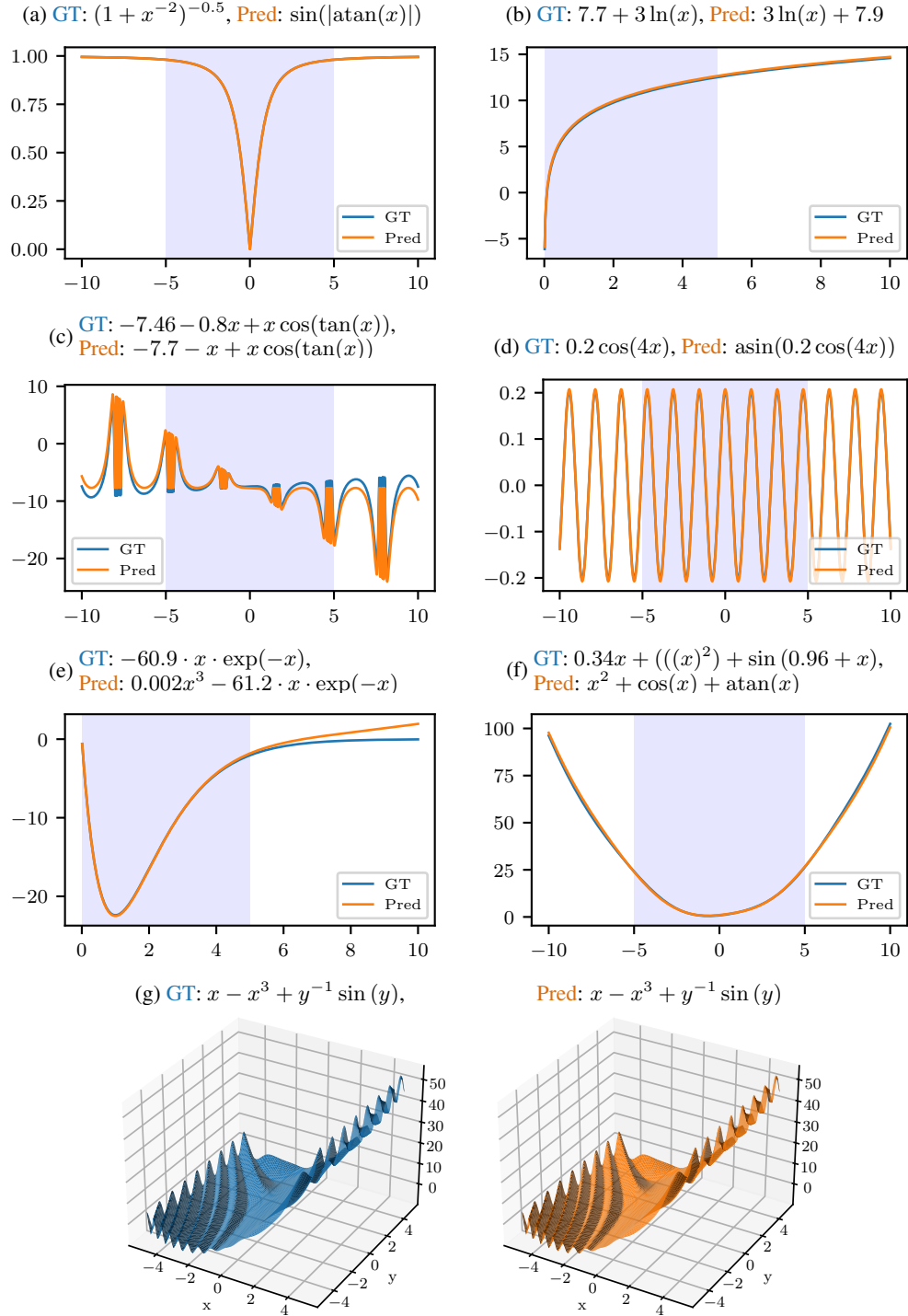


Figure 2: Examples of model predictions using Top-K sampling with $K = 20$ and 256 samples. The shaded area represents the sampling range. For the 3D functions, x and y were sampled from $[-5, 5]$. ‘GT’ denotes ground-truth and ‘Pred’, the model prediction. The first six images are generated using univariate SymFormer, and the last one was generated by bivariate SymFormer.

Table 2: Results comparing the **SymFormer**, when the different model is used for univariate functions and bivariate functions and when the Bivariate SymFormer is used for all the benchmark functions. The (Bivariate) **SymFormer** uses Top-K sampling with $K = 20$ while generating 1024 samples and improving them using local gradient search with early stopping. We report R^2 and the average time to generate an equation in seconds.

Benchmark	SymFormer		Bivariate SymFormer	
	$R^2 \uparrow$	Time (s) \downarrow	$R^2 \uparrow$	Time (s) \downarrow
Nguyen	0.99998	47.50	0.99996	139.46
R	0.99986	94.33	0.99985	418.67
Livermore	0.99996	43.00	0.99992	170.00
Koza	1.00000	101.00	1.00000	81.50
Keijzer	0.99904	48.67	0.99884	250.66
Constant	0.99998	90.88	0.99997	188.50
Overall avg.	0.99978	52.95	0.99946	174.32

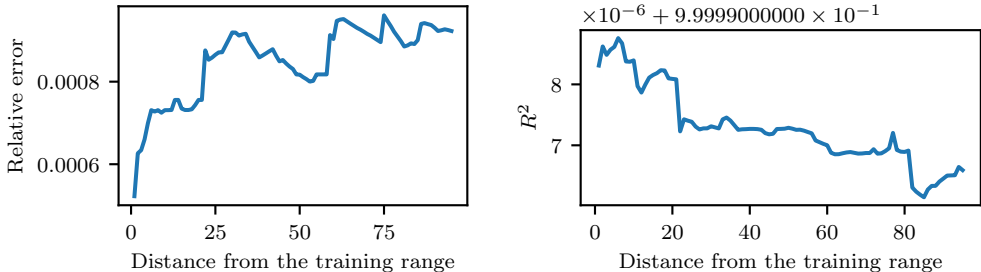


Figure 3: The effect of a distance when calculating the relative error outside the sampling range. They are estimated on 1024 equations generated with Top-K sampling with $K = 20$ and 256 samples. Each of the equations is also improved by a local gradient search on constants.

only a slight drop in performance. However, the average inference time increased, which could be explained by a larger search space the model needed to handle during the optimisation of constants. Furthermore, we have manually inspected the model’s predictions on benchmark functions. We found that the model had no problems recovering simple equations but was slow or failed in cases of more complicated functions.

4.5 Out-of-domain performance

One intriguing property of symbolic regression is its ability to predict the correct values outside the sampling range. To test it, we first run the inference on the points sampled from the training range and then evaluate these predicted functions on points outside the sampling range. More formally we calculate the metrics on the function values for points sampled from the set $\{x \in \mathbb{R} | 5 < |x| < 5 + d\}$, where d is the maximal distance. The effect of the distance on the relative error and R^2 can be seen in Figure 3. Even though the error increases with the distance, the final relative error is minimal, even for the maximal range. Therefore, we can conclude that the model generalises outside of the sampling range, and the local gradient search does not overfit the constants to the sampled data.

4.6 Discovering mathematically equivalent expressions

To qualitatively evaluate the SymFormer’s prediction capabilities, we have manually inspected the model’s predictions. The model is often able to find mathematically equivalent expressions. In one case the model discovered the rule $-\ln x = \ln x^{-1}$. The goal was to predict $-\frac{8}{\ln x}$, but the model predicted $\frac{8}{\ln(x^{-1})}$. Another rule the model discovered was the law of exponents $(x^a)^b = x^{a \cdot b}$. It was observed when the model had to predict $(x^{-1})^{1.5}$, but found an equivalent form $\frac{1}{\sqrt{x^3}}$.

Table 3: Comparison of expression encoding strategies and local gradient search (‘LGS’). The **SymFormer** uses both ‘extended encoding’ and local gradient search. Metrics are estimated using 256 equations using Top-K sampling with $K = 20$ and 256 samples, where the equation with the lowest mean squared error on the input points is selected. We report the R^2 and the relative error. The base encoding refers to case when no preprocessing for the constants is used. BFGS init refers to a situation when the predicted constants are used as a starting point for the BFGS [Fletcher, 1987]. The ‘GS’ refers to the case when the gradient search was used to find or improve the constants further.

Model	GS	$R^2 \uparrow$	Relative error \downarrow
No constants + BFGS	✓	0.9929	0.1547
Base encoding	✗	0.9979	0.0669
Base encoding + LGS	✓	1.0000	0.0089
Extended encoding	✗	0.9995	0.0288
Extended encoding + BFGS init	✓	0.9998	0.0303
SymFormer	✓	1.0000	0.0010

Furthermore, the model was also able to find some trigonometric equivalencies such as $\frac{1}{\tan x} = \cot x$. However, a more interesting example is the expression $\cos(3.5 + 2x + x^2)$. In this case, the model predicted $-\sin(2 + 2x + x^2)$, which has a very small error. The rule, that the model discovered is $\cos(\frac{\pi}{2} + x) = -\sin(x)$. Applying this rule we get $\cos(3.5 + 2x + x^2) \approx \cos(\frac{\pi}{2} + 1.9 + 2x + x^2) \approx -\sin(1.9 + 2x + x^2)$. Another interesting example the model predicted is the exponential rule $a^x = e^{x \cdot \ln a}$. This rule is probably used when the SymFormer needs to deal with precise constants, due to the numerical stability of \ln . For example, the model had to find $(-1.3673x)$, but it found $\ln(0.2492^x) = \ln(e^{x \ln 0.2492}) = x \ln 0.2492 = -1.3895x$ which is close to the previous expression.

4.7 Ablation study

This ablation study aims to look at the effect of different constant encodings. In the first setting, we did not predict the constants and used a global optimisation (BFGS [Fletcher, 1987]) to find the constants. This setting is the same as used by Biggio et al. [2021]. In the second setting we used the constants during training, but we did not preprocess them. Therefore, a single symbol, ‘const’, was used to represent any constant regardless of its magnitude. In the last case, we have used encoding as described in Section 3.4, while also trying to use the predicted values of constants as a starting point for global optimisation. From the results in Table 3, we can see that the constants help the model performance in terms of both the R^2 and the relative error. Therefore, one can conclude that the performance of SymFormer in comparison to Biggio et al. [2021] is better not because of a different dataset or a larger model but because of the usage of constants during training. The last row shows the results for the extended encoding, which uses a local gradient search to improve the constants further. The extended encoding clearly outperforms the base encoding in terms of both the R^2 and the relative error. We believe this to be the case because it is easier for the model to attend to previous symbolic tokens than to real values and, therefore, the model can make a more informed decision when predicting the next symbol in the sequence.

5 Conclusion

To tackle the problem of symbolic regression, we introduced a novel transformer-based approach called the SymFormer that uses a neural network trained on hundreds of millions of formulas to be able to generate a symbolic representation of a previously unseen formula given a set of input-output pairs efficiently. Our model jointly predicts the symbolic representation of a function and the values of all constants in a single forward pass of a neural network. A local gradient search is used to improve constants further to fit the input points better. We demonstrated that the SymFormer is competitive with current state-of-the-art approaches while outperforming them in terms of the time required to find the expression. We validated the importance of the proposed encoding of constants in an ablation study. Furthermore, by evaluating the SymFormer outside the sampling range, we showed that it has good extrapolation capabilities. Finally, in a qualitative evaluation, we present

some intriguing mathematical relations the model was able to recover just by learning on a large collection of formulas.

Limitations. One of the limitations of our approach is that the maximal number of dimensions and the sampling range cannot be changed after the model was trained. In the future, this can be partially tackled by varying the sampling distribution of the input points during training.

Acknowledgments

This work was supported by the European Regional Development Fund under the project Robotics for Industry 4.0 (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000470). Jonáš Kulhánek was funded under project IMPACT (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000468) and by the Grant Agency of the Czech Technical University in Prague (grant no. SGS22/112/OHK3/2T/13). Computational resources were further funded by the Ministry of Education, Youth and Sports of the Czech Republic through the e-INFRA CZ (ID:90140).

References

- Ignacio Arnaldo, Una-May O’Reilly, and Kalyan Veeramachaneni. Building predictive models via feature synthesis. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO ’15*, page 983–990, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334723. doi: 10.1145/2739480.2754693.
- 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>.
- Iwo Bładek and Krzysztof Krawiec. Solving symbolic regression problems with formal constraints. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO ’19*, pages 977–984, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6111-8. doi: 10.1145/3321707.3321743.
- 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>.
- 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.
- 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>.
- 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>.
- Andrew Jaegle, Felix Gimeno, Andrew Brock, Andrew Zisserman, Oriol Vinyals, and João Carreira. Perceiver: General perception with iterative attention. *CoRR*, abs/2103.03206, 2021. URL <https://arxiv.org/abs/2103.03206>.
- 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.
- 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.
- 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>.
- Jirí Kubalík, Erik Derner, and Robert Babuska. Symbolic regression driven by training data and prior knowledge. *CoRR*, abs/2004.11947, 2020. URL <https://arxiv.org/abs/2004.11947>.
- Guillaume Lample and François Charton. Deep learning for symbolic mathematics. *CoRR*, abs/1912.01412, 2019. URL <http://arxiv.org/abs/1912.01412>.
- 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 spectroscopy 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>.
- Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019.
- 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>.
- M. Schmidt and H. Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009a.
- Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923): 81–85, 2009b. doi: 10.1126/science.1165893. URL <https://www.science.org/doi/abs/10.1126/science.1165893>.
- 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>.
- Nicolas Staelens, Dirk Deschrijver, Ekaterina Vladislavleva, Brecht Vermeulen, Tom Dhaene, and Piet Demeester. Constructing a no-reference h.264/avc bitstream-based video quality metric using genetic programming-based symbolic regression. *IEEE Trans. Cir. and Sys. for Video Technol.*, 23(8):1322–1333, August 2013. ISSN 1051-8215. doi: 10.1109/TCSVT.2013.2243052.
- Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O’Neill, R. I. McKay, and Edgar Galvan-Lopez. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12(2):91–119, June 2011. ISSN 1389-2576. doi: doi:10.1007/s10710-010-9121-2.

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

Supplementary Material

SymFormer: End-to-end symbolic regression using transformer-based architecture

In the supplementary material, we give more details on the experiments presented in the paper, data generator’s and model’s hyperparameters, and we also include additional experiments that evaluate the performance of the presented approach in more detail. In Section A, we investigate the effect of the number of sampled equations using Top-K sampling [Fan et al., 2018] on the model’s performance and the inference time. In Appendix B, we visualise several different model predictions using both univariate and bivariate SymFormer. In Appendix C, we enumerate all benchmark functions that were used to compare our method with the current state of the art, and in Appendix D, we present the exact values that we have measured while comparing the methods including the used hyperparameters. In Appendix E, we present the exact hyperparameters that were used to generate both datasets, and in Appendix F, we present the model hyperparameters, including its vocabulary and other training details.

Furthermore, we have also published the source code containing the necessary files to run the training and the inference ³. We have also included a video ⁴ which shows a visualisation of inference on several functions. The process starts by sampling 256 equations and selecting the best eight functions from the sampled equations. In the video, they are visualised as light blue functions. The orange points represent the sampled points used for the inference. We then show how the functions improve with each constant optimization step.

A Effect of number of sampled equations

Table 4: Performance of number of sampled equations for Top-K [Fan et al., 2018] with $K = 20$ without a local gradient search on 256 equations. The time is in hours:minutes:seconds format.

# of equations	$R^2 \uparrow$	RE \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

First, we evaluate the model’s performance with varying number of sampled equations during inference. In Table 4, we can see that the SymFormer’s ability to predict the correct expression improves with the number of sampled equations, however, the time increases substantially, averaging approximately three minutes per equation for the largest number of equations. These results are measured by running the inference with Top-K sampling [Fan et al., 2018] with $K = 20$ and without a local gradient search. If we had used a local gradient search, the R^2 and relative error (RE) would improve, however, the time would increase.

³<https://github.com/vastlik/symformer>

⁴<https://vastlik.github.io/symformer/video>

B Examples of generated functions

We plotted several predictions generated by the SymFormer with the Top-K sampling [Fan et al., 2018] with $K = 20$ with 256 sampled equations and the local gradient search. The shaded area represents the sampling range. The generated predictions can be seen in Figure 4. The predicted functions are not always the same as the original formula, however, they fit almost perfectly on the function domain. In Figure 5, we show similar visualisation for the bivariate SymFormer.

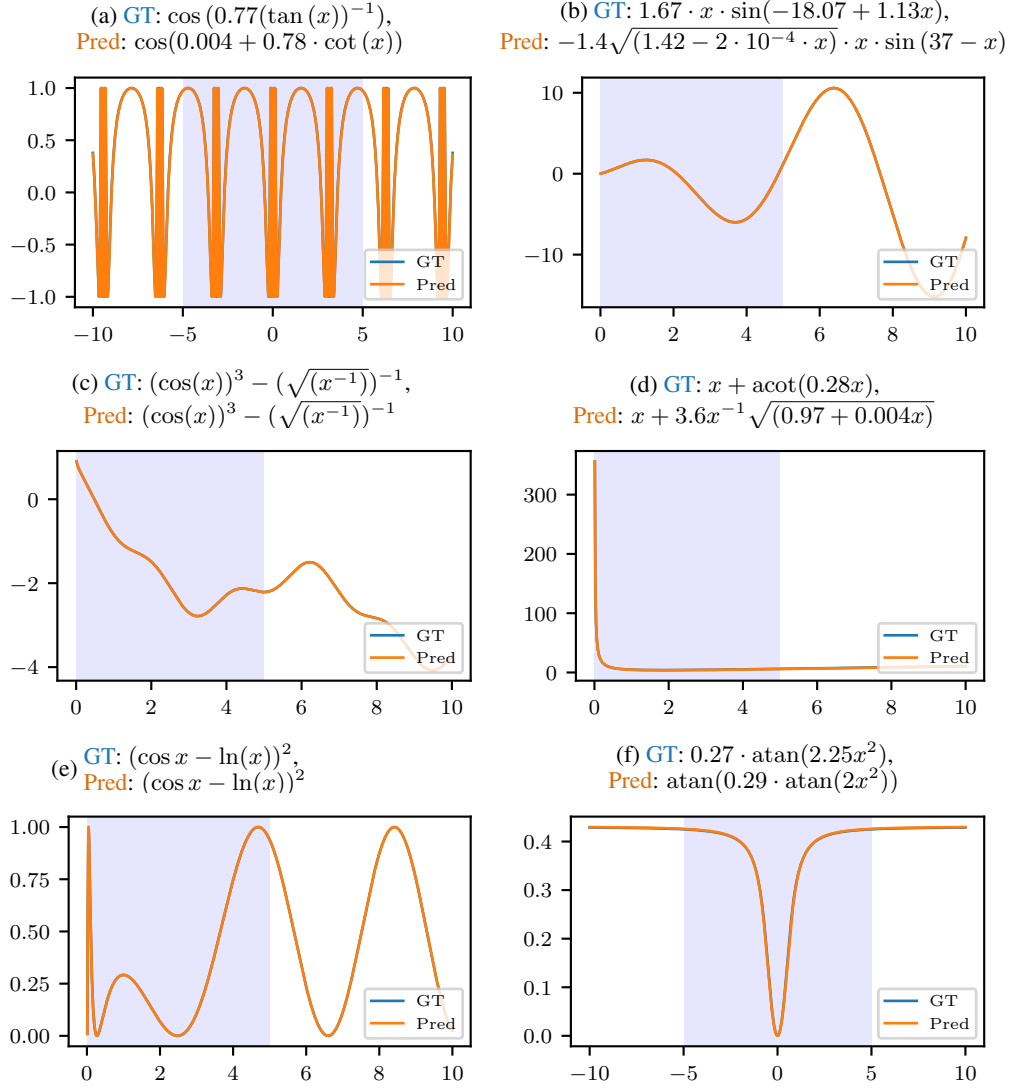
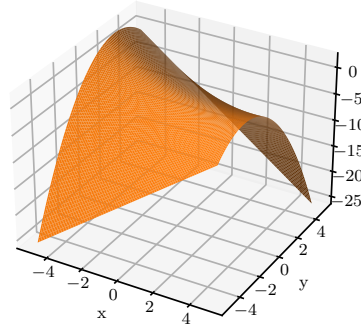
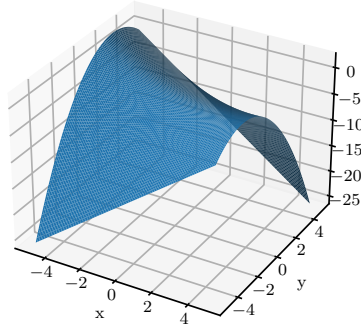


Figure 4: Examples of model predictions using Top-K sampling [Fan et al., 2018] with $K = 20$ and 256 samples. The shaded area represents the sampling range. ‘GT’ denotes ground-truth and ‘Pred’, the univariate SymFormer prediction.

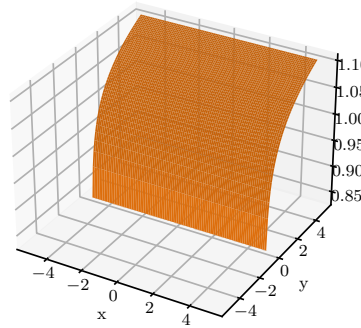
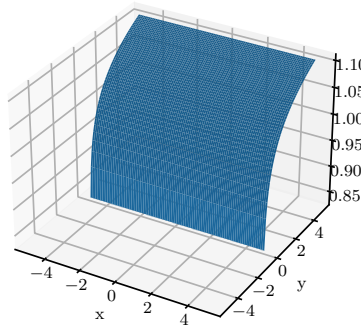
(a) GT: $-0.99 - 0.5y^2 - 0.5xy$,

Pred: $-1 - 0.5y^2 - 0.5xy$



(b) GT: $y^{0.0625}$,

Pred: $y^{0.0625}$



(c) GT: $-1 - 0.94y + xy - y \sin(x)$,

Pred: $-1.14 - 0.94y + xy - y \sin(x)$

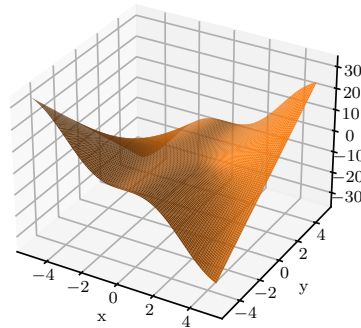
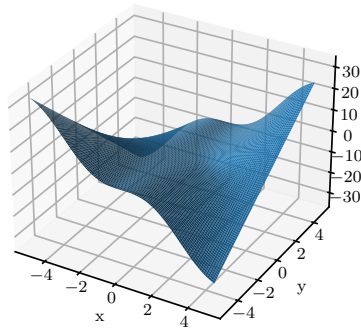


Figure 5: Examples of the bivariate SymFormer predictions using Top-K sampling [Fan et al., 2018] with $K = 20$ and 256 samples. ‘GT’ denotes ground-truth and ‘Pred’, the model prediction.

C Benchmark functions

This section describes the exact functions used to compare the SymFormer with the current state-of-the-art methods. The benchmark's names and the contained functions can be seen in Table 5.

Table 5: Benchmark functions that we have used in our experiments. We have restricted ourselves only to the univariate and bivariate functions.

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

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

D Benchmark comparison

This section presents a comparison for each equation from the used benchmarks. NSRS denotes Neural Symbolic Regression that Scales [Biggio et al., 2021], and DSO denotes Symbolic Regression via Neural-Guided Genetic Programming Population Seeding [Mundhenk et al., 2021]. You can see that DSO tends to perform similarly to SymFormer, but it usually takes more time to find the underlying equation due to the usage of reinforcement learning, for example, in the case of the Keijzer [Keijzer, 2003] benchmark, where the DSO required thousands of seconds in most cases. For the NSRS, we can see that the times are consistent across the benchmarks, however larger than for SymFormer. The reason for this observation is that the model uses global optimisation to find the coefficients, which slows down the inference. The hyperparameters for the DSO can be found in <https://github.com/brendenpetersen/deep-symbolic-optimization/> and for the NSRS in <https://github.com/SymposiumOrganization/NeuralSymbolicRegressionThatScales>. The experiments were run using 32 CPU threads and 64 GB of RAM. From the results in Table 6, we can see that in most cases, the SymFormer is competitive in terms of R^2 , however, it outperforms them in case of the time in the average case.

Table 6: Comparison between methods on each of the benchmark functions, R^2 values are rounded to 4 decimals and time to whole seconds.

Name	SymFormer		NSRS		DSO	
	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
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
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
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-18	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
<hr/>						
Koza-2	1	20	1	73	1	27
Koza-3	1	182	1	150	1	408
<hr/>						
Keijzer-3	1	49	0.7549	174	0.6454	5802
Keijzer-4	0.9887	58	0.9991	179	0.8990	9171
Keijzer-6	1	13	1	108	1	510
Keijzer-7	1	5	1	138	1	1678
Keijzer-8	1	5	1	203	1	86
Keijzer-9	1	116	0.996	126	1	1304
Keijzer-10	1	12	0.9335	316	0.9856	2926
Keijzer-11	1	78	1	240	0.9536	5235
Keijzer-12	1	48	1	521	1	8124
Keijzer-13	1	96	1	321	0.9526	6848
Keijzer-14	1	37	1	274	1	3864
Keijzer-15	1	67	1	266	0.9999	1601
<hr/>						
Constant-1	1	23	1	175	1	972
Constant-2	1	162	0.0996	130	1	4836
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
Constant-8	1	220	1	172	1	8851

E Hyperparameters for dataset generation

In this section, we describe the exact hyperparameters used to generate the bivariate and univariate datasets. Note that in the case of the bivariate dataset, the probability of selecting the x variable was the same as the y variable. The probability of selecting the unary operation is also the same as the binary operation. The unnormalized probabilities for unary operations can be seen in Table 7, for binary in Table 8 and for leafs in Table 9.

Table 7: Unnormalised probabilities of unary operators used by the dataset generator. We have also used special operations to generate them more often.

Operation	Mathematical meaning	Unnormalized 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	$\operatorname{arccot} \cdot$	1

Table 8: Unnormalised probabilities of binary operators as used by the dataset generator.

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

Table 9: Unnormalised probabilities of leaf values as used by the dataset generator.

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

F Model hyperparameters and vocabulary

In this section, we give more details about the model’s hyperparameters that were used to train our model. We have generated two datasets – one with univariate functions containing 130 million equations and the second one with bivariate functions containing 100 million equations. Note that the bivariate dataset can also contain functions that have only one input variable. The main difference between these two datasets is the number of sampled points, 100 for the univariate case, and 200 for the bivariate case. For the model, we have decided to use a large encoder which has the majority of parameters (approximately 75 millions) and the rest for the decoder. The idea is that the encoder should not just encode the points, but also represent the function on a high level such that the decoder only prints the representation as a sequence of symbols. The full set of hyperparameters can be seen in Table 10.

We also present our model’s vocabulary which can be seen in Table 11 where we have used the common functions such as \ln or \sin . However, we have also used some operators, that are not elementary functions, *e.g.*, $\text{neg}(x) = -1x$. We have also added integers from -5 to 5 , so it is easier for the model to represent them. We also did not use the hyperbolic trigonometric functions, since they can be represented by an equivalent expression *e.g.* $\sinh x = \frac{\exp(x) - \exp(-x)}{2}$.

Table 10: Hyperparameters for both the univariate and bivariate models.

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 (bivariate)	130 million (100 million)
Number of sampled points (bivariate)	100 (200)
Training	
Number of epochs (bivariate)	390 (300)
Starting σ^2 noise	0.1
Ending regression λ	1

Table 11: Vocabulary as used by the model. Besides the elementary functions, it contains special functions such as pow2 and pow3, which are commonly used.

Token	Mathematical meaning
Integers from [-5, 5]	Integers from [-5, 5]
Variables	Variables
pow	(\cdot)
+	addition
*	multiplication
sqrt	$\sqrt{\cdot}$
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	$\text{acot} \cdot$
neg	$(-1)(\cdot)$
C	Constants, optionally C-10, ..., C10 for scientific-like encoding