Received 21 January 2024, accepted 24 February 2024, date of publication 7 March 2024, date of current version 15 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3374649

RESEARCH ARTICLE

SymFormer: End-to-End Symbolic Regression Using Transformer-Based Architecture

MARTIN VASTL[1,2], JONÁŠ KULHÁNEK[1,3], JIŘÍ KUBALÍK[1], ERIK DERNER[1], AND ROBERT BABUŠKA[1,4], (Member, IEEE)

[1]Czech Institute of Informatics, Robotics, and Cybernetics, Czech Technical University in Prague, 16000 Prague, Czech Republic
[2]Faculty of Mathematics and Physics, Charles University, 12116 Prague, Czech Republic
[3]Faculty of Electrical Engineering, Czech Technical University in Prague, 16000 Prague, Czech Republic
[4]Department of Cognitive Robotics, Delft University of Technology, 2628 CD Delft, The Netherlands

Corresponding author: Erik Derner (erik.derner@cvut.cz)

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) and by the Ministry of Education, Youth and Sports of the Czech Republic through the e-INFRA CZ (ID:90140) and through the e-INFRA CZ (ID:90254). This work has also received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No. 101070254 CORESENSE. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the Horizon Europe programme. Neither the European Union nor the granting authority can be held responsible for them.

ABSTRACT Many real-world systems can be naturally described by mathematical formulas. The task of automatically constructing formulas to fit observed data is called symbolic regression. Evolutionary methods such as genetic programming have been commonly used to solve symbolic regression tasks, but they have significant drawbacks, such as high computational complexity. Recently, neural networks have been applied to symbolic regression, among which the transformer-based methods seem to be most promising. After training a transformer on a large number of formulas, 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, yielding suboptimal results. We propose a transformer-based approach called SymFormer, which predicts the formula by outputting the symbols and the constants simultaneously. This helps to generate formulas that fit the data more accurately. In addition, the constants provided by SymFormer serve as a good starting point for subsequent tuning via gradient descent to further improve the model accuracy. We show on several benchmarks that SymFormer outperforms state-of-the-art methods while having faster inference.

INDEX TERMS Symbolic regression, neural networks, transformers.

I. INTRODUCTION

Many systems in various fields ranging from industrial processes to social sciences can be described by mathematical formulas. Knowing the governing equations of a nonlinear system provides insight into the system's inner workings and it also allows us to predict how the system will behave in the future. Deriving mathematical models from first principles is often tedious and for some systems even impossible. In such a case, methods to automatically construct formulas fitting the data observed on the system can be used. The task of finding such a formula from the observed data is called *symbolic regression* (SR). This method has already been applied to a variety of real-world problems, *e.g.*, in physics [1], [2], robotics [3], [4], or machine learning [5].

In the past decades, symbolic regression tasks [6], [7] were commonly solved by means of genetic programming [6], [8], [9], [10], [11]. However, discovering formulas in this way is slow and computationally expensive. For each SR instance, an entire population of formulas has to be evolved and evaluated repeatedly through many generations. In recent years, approaches based on neural networks emerged [12], [13], [14], [15]. Among them, the most efficient ones are methods that train a transformer model on a large

The associate editor coordinating the review of this manuscript and approving it for publication was Fahmi Khalifa.

collection of data and the corresponding formulas [14], [15]. The transformer uses the data to autoregressively generate formulas by predicting each symbol conditioned on the previously generated symbols. The expressions are decoded without constants, i.e., all constants are replaced by a special symbol and are sought afterward using global optimization [14], [15]. However, the values of the constants have a large impact on how well the generated function fits the data. Without generating the constants simultaneously with the equations, the model will not represent the data well.

Inspired by [16], where a similar idea was applied to the problem of recurrent sequences, we propose *SymFormer*, a novel transformer-based architecture trained end-to-end on hundreds of millions of formulas. Our work makes the following contributions:

1) SymFormer generates a symbolic representation of the formula, including the numerical values of the constants. This allows the symbolic decoder to use the constants generated so far and so to improve the model's accuracy.
2) We use the generated constants to initialize a local gradient optimizer to fine-tune the final constants.
3) Our approach is thoroughly evaluated and compared to relevant alternative methods on a large set of univariate and bivariate functions. We also study the effect of using different constant encoding methods.
4) The source code and the pre-trained model checkpoints are publicly available at https://github.com/vastlik/symformer.

II. RELATED WORK

Genetic Programming approaches are a traditional way of solving SR [8], [17], [18]. Genetic programming (GP) evolves expressions encoded as trees using selection, crossover, and mutation. The drawbacks of GP-based approaches are that they evolve each equation from scratch, which is slow, and that the models tend to increase in complexity without much performance improvement [19], [20]. GP is also inefficient in fine-tuning the constants only by using genetic operators [21], [22].

Neural Network approaches can be generally divided into three categories. The first one includes approaches based on the Equation learner (EQL) [23], [24], [25]. The idea behind EQL is to find a function $f(x) = y$ by training a neural network on x as the input and y as the output. Through regularization, the neural network is forced to use as few network weights as possible. Elementary functions (sin, log, etc.) are used as activation functions, and after the training, they are read from the network with the corresponding weights. A limitation of the EQL-like approaches is that one has to design a strategy to force convergence towards a sparse neural network model representing a compact analytic formula. Typically, such a strategy implements a trade-off between the neural network's accuracy and sparsity. Lastly, these approaches can be slow as they need to find each equation from scratch.

The second set of approaches is based on training a recurrent neural network (RNN) using reinforcement learning [12]. The idea is to let the RNN generate the equation and then calculate the reward 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 in [13], where the RNN is used to sample an initial population for a genetic algorithm. A limitation of both of these approaches is that the model does not generate the constants, which have to be found through nonlinear optimization, slowing down the whole training loop and limiting the achievable accuracy of the model [12], [13].

A transformer-based approach was introduced in [14], [15], [16], where a large amount of training data is generated and used to train a transformer [26] in a supervised manner. A similar approach is proposed in [15], where the GPT-2 [27] model is trained on the input-output data with the corresponding symbolic expression as the output. Global optimization finds the constants for each equation. In [14], the encoder from the Set Transformer [28] and the decoder from the original transformer architecture [26] are used. Similar to [15], the models are trained only on skeletons (expressions without constants), and afterward, the constants are fitted using global optimization. An approach where the transformer directly generates the constants is introduced in [16]. The constants are predicted jointly by encoding them into the symbolic output. Integers are represented through the base-b direct encoding, and the IEEE 754 float representation is used with the mantissa rounded to the four most significant digits. New tokens are introduced to represent exponents. A disadvantage of this approach is that the mantissa has a finite precision. The authors also observed that when approximating difficult functions, the symbolic model typically only predicts the largest terms in its asymptotic expansion. This method was further extended in [29], where the constants found by the model serve as initial values for global optimization to improve the accuracy further. Unlike these methods, SymFormer does not limit the precision of the constants it outputs.

III. METHOD

The symbolic regression task can be formulated as finding an unknown function f given a finite set of input data points along with the corresponding outputs. The goal is to construct a function (mathematical formula) $\hat{f}$ minimizing the squared difference between the function's output on the input points and the outputs of the unknown function f. In this work, we focus on univariate and bivariate functions.

Given a set of observed input-output pairs, the model generates the structure of the formula together with the values of all constants present in the formula in a single forward pass of the transformer model [26]. This is visualized in Figure 1. The input-output pairs are concatenated into a sequence which is processed by the transformer's encoder. The transformer's decoder attends to the encoder's resulting sequence and autoregressively predicts the formula as a

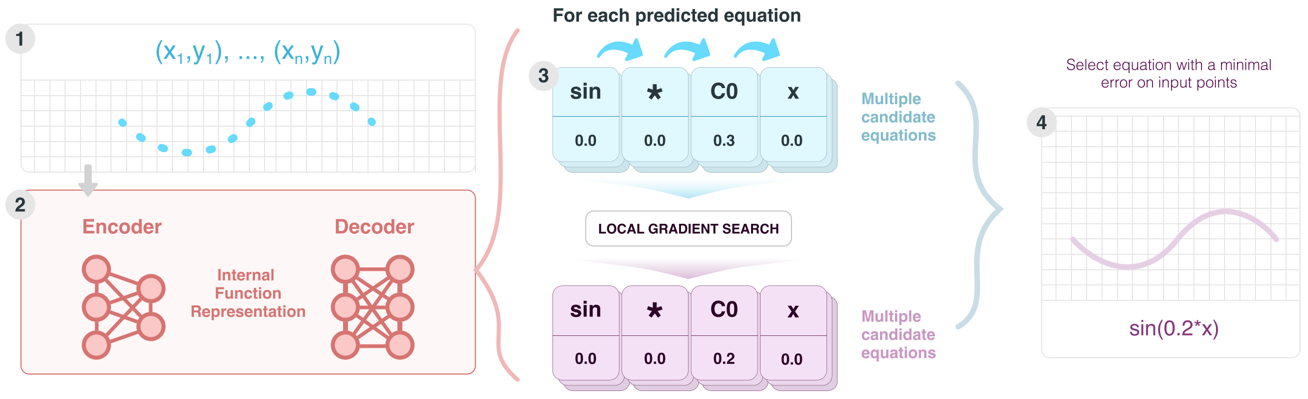
1
$(x_1,y_1), ..., (x_n,y_n)$
2
Encoder
Decoder
Internal Function Representation
For each predicted equation
3
sin * C0 x
0.0 0.0 0.3 0.0
Multiple candidate equations
LOCAL GRADIENT SEARCH
sin * C0 x
0.0 0.0 0.2 0.0
Multiple candidate equations
Select equation with a minimal error on input points
4
sin(0.2*x)

FIGURE 1. **Schematic diagram of SymFormer inference. The input-output data 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 minimizing the mean squared error.**

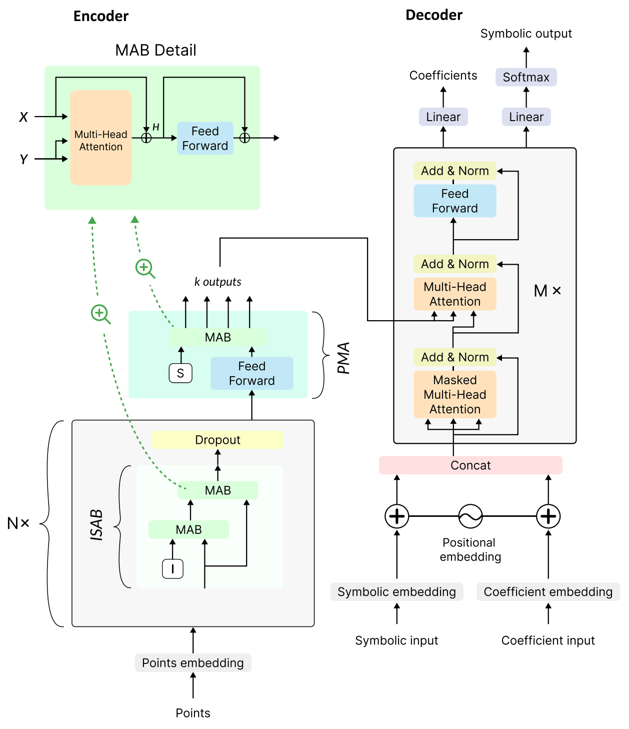

FIGURE 2. **Schematic overview of the SymFormer architecture. MAB refers to the Multihead Attention Block, ISAB is the Induced Set Attention Block, and PMA stands for Pooling by Multihead Attention.**

sequence of individual symbols in the formula and the corresponding constants. In fact, the decoder models the probability distribution over the next symbol and the value of the constant, given the expression prefix and the associated values of the constants.

A. MODEL ARCHITECTURE

The SymFormer model architecture is depicted in Figure 2. Following [26], our model consists of an encoder and a decoder. The encoder takes as its input the data points (input-output pairs) sampled from the formula. Each data point is first passed through an affine layer to increase the dimensionality to the hidden size of the transformer. The resulting vectors are then concatenated to build the sequence that the transformer encoder processes. The order of these vectors is not important since the positional encodings are not used, and, therefore, the encoder treats each vector independently of its position in the sequence. We pass the sequence of hidden vectors through four *induced set attention blocks* [28], each of which consists of two cross-attention layers and two traditional feed-forward blocks [26]. Following the standard practice in transformer architectures [26], for all cross-attention layers and feed-forward blocks, we pairwise sum the output with the layer's input before passing it through a normalization layer [30].

The first cross-attention uses the hidden vectors as its keys and values and a set of trainable vectors as the queries. The resulting sequence will have the same length as the number of these trainable query vectors. Note that each encoder block has its own independent set of query vectors. The resulting sequence is passed through the first feed-forward block. The second cross-attention layer then uses the previously generated sequence as the keys and values, and it uses the original hidden sequence (the input to the first cross-attention) as its queries, which results in the same sequence length as the length of the original input. The second cross-attention is followed by the second feed-forward block and a dropout layer [31]. Finally, to get a representation of the input independent of the number of data points, we compute the cross-attention between the output of the last block and a set of trainable vectors. Similarly, as in the individual encoder blocks, this cross-attention fixes the sequence length to the number of trainable query vectors.

The input to the decoder is a sequence of trainable positional embeddings that are summed with embedded ground truth output symbols and concatenated with affine-projected ground-truth constants. The decoder then passes the sequence through four standard transformer blocks [26], consisting of a self-attention layer, a cross-attention between the hidden sequence and the encoder's output, and a feed-forward block. Two heads are applied to the output of the last block: a classification head outputting the symbols and a regression head outputting the values of constants.

B. TRAINING AND INFERENCE

The model is trained to predict the next symbol and the associated value of a constant given the previous symbols in the expression. Thanks to the transformer architecture [26], the model can be optimized for all symbols in the sequence with linear overhead. The classification head is trained using the cross-entropy loss $\mathcal{L}_{class}$ with the symbols as the target. The regression head is trained using the mean squared error (MSE) $\mathcal{L}_{\text{MSE}}$ to output the value of the constant if the associated symbol is a constant. The regression loss is masked such that it is optimized only on positions where the associated symbols are the constants. The total loss $\mathcal{L}$ is a weighted sum of the two losses:

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

where λ is a hyperparameter. At the beginning of the training, we set λ to zero, and after approximately 97 700 gradients steps, we gradually increase it using the cosine schedule [32]. During training, we also found it beneficial to add a small random noise sampled from $\mathcal{N}(0, \sigma^2)$ to the constants because the constants are not always precise during the inference. The parameter σ is also decreased according to the cosine schedule.

At inference time, we encode the set of input-output pairs using the encoder. The decoder then autoregressively decodes a sequence of symbols and constants one at a time by first sampling from the categorical distribution of the symbol classification head and then taking the prediction from the regression head if the predicted symbol was a constant. This process is terminated when the model reaches the end-of-sequence (EOS) token. In practice, we sample not one but multiple independent sequences. For all these sequences, we fine-tune the values of all constants by minimizing the MSE on the observed data samples, see Section III-C. Finally, we select the formula with the lowest error after the optimization.

C. CONSTANTS FINE-TUNING

To optimize the constants in the predicted expressions, we use a stochastic gradient descent (SGD) method to minimize the MSE between the function prediction and the observed outputs in the training data. For each expression, we initialize the gradient descent procedure with the learning rate of 0.001 and the momentum of 0.9. Next, we run a gradient-descent iteration while clipping the gradient norm to 10 and calculate the MSE with the newly found constants. If the loss does not decrease by at least 0.1 %, we divide the learning rate by 10. This process continues until the loss stops decreasing, i.e., it does not improve by at least 0.1 % for five consecutive iterations.

This process and its parameters were inspired by classical optimization, balancing the trade-off between convergence rate and performance. The SGD optimization process starts with the constant values predicted by the model and aims to fine-tune these constants rather than finding significantly different ones. In our empirical evaluation, the optimization has shown to be efficient thanks to the initial constant values being closely aligned with the desired ones. After the constants are optimized for all predicted expressions, we select the formula with the lowest error.

D. EXPRESSION ENCODING

Technically, we encode each expression as a sequence of symbols and a sequence of real values. The symbols correspond to different mathematical operators, e.g., $+$ for addition, $\cdot$ for multiplication, etc., expression's variables, e.g., x, y, or expression's constants – numbers occurring in the expression. All possible symbols constitute the model's vocabulary. The constants are encoded using a scientific-like notation where a constant α is represented as a tuple of the exponent c_e and the mantissa c_m:

$$\alpha \approx c_m \cdot 10^{c_e}, \qquad c_e = \lceil \log_{10} \alpha \rceil, \qquad c_m = \frac{\alpha}{10^{c_e}}. \tag{2}$$

In this representation, the mantissa is in the range $[-1, 1]$, and the exponent is an integer. The integer exponent is encoded as a special symbol starting with 'C' and followed by the exponent c_e. The mantissa is kept as the real number (it is optimized using the regression loss). In order to transform the expression tree into this representation, we flatten the tree using the preorder traversal [33]. For example, the expression $0.017 \cdot x + 1781.5$ has symbols [+, ·, x, C-1, C4] and constants [0, 0, 0, 0.17, 0.17815]. To further help the model represent common integers, we add all integers from the interval $[-5, 5]$ to the model vocabulary. In contrast to the approach in [16], which is able to express constants only up to the four most significant digits, our approach achieves full float precision.

IV. EXPERIMENTS

This section describes our training setup and the metrics that we used to demonstrate the model's ability to predict the formulas and compare our model to relevant approaches from the literature. We also present an ablation study comparing 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.

To train and test our method, we have generated two datasets: one containing 130 million univariate functions and another one containing 100 million bivariate functions. The univariate and bivariate test datasets contain 10 000 randomly sampled equations each. To generate the formulas, we modified the algorithm described in [33] with a maximum of ten operators. Then, we sampled uniformly at random 100 points (200 for bivariate functions) from the interval $[-5, 5]$. The

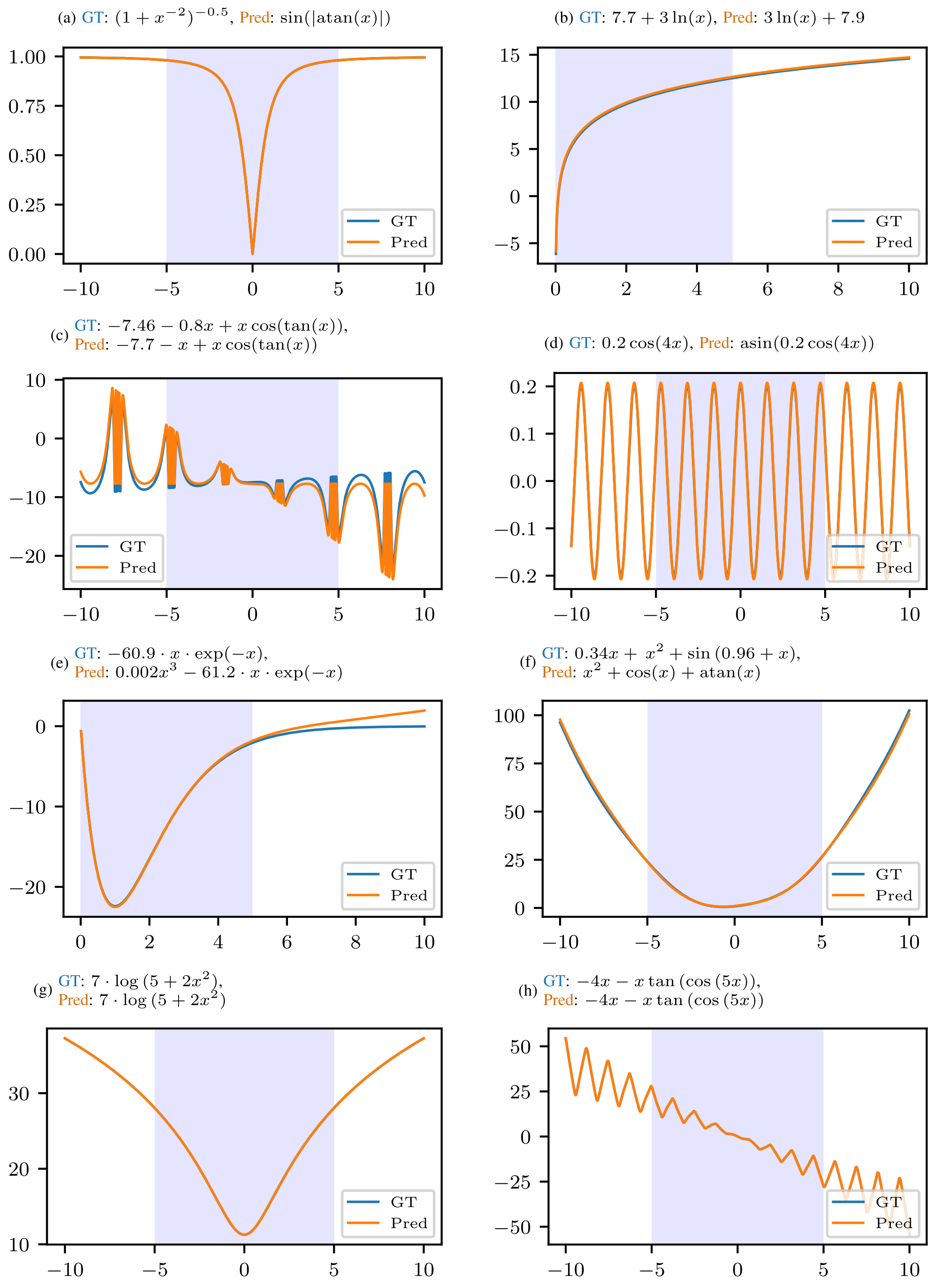

FIGURE 3. Examples of model predictions from the Univariate SymFormer. The shaded area represents the training range. 'GT' denotes the ground truth and 'Pred' is the model prediction.

interval was reduced if the function was not defined on the full interval.

A. TRAINING AND EVALUATION

We train our model using the Adam optimizer [34] for 3 epochs on 8 NVIDIA A100 GPUs. The training of the model takes roughly 33 hours. We use a training schedule similar to the original transformer [26]. However, we divide the learning rate by five since the training often diverges when using the original learning rate. The regression parameter λ is set according to the cosine schedule and delayed for 97 700 gradient steps, reaching 1.0 at the end of the training.[1] The random noise is sampled from $\mathcal{N}(0,\epsilon)$ where ϵ is initially set to 0.1 and decreased to zero during training using the same schedule. The same setup is used both for the Univariate and Bivariate SymFormer.

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

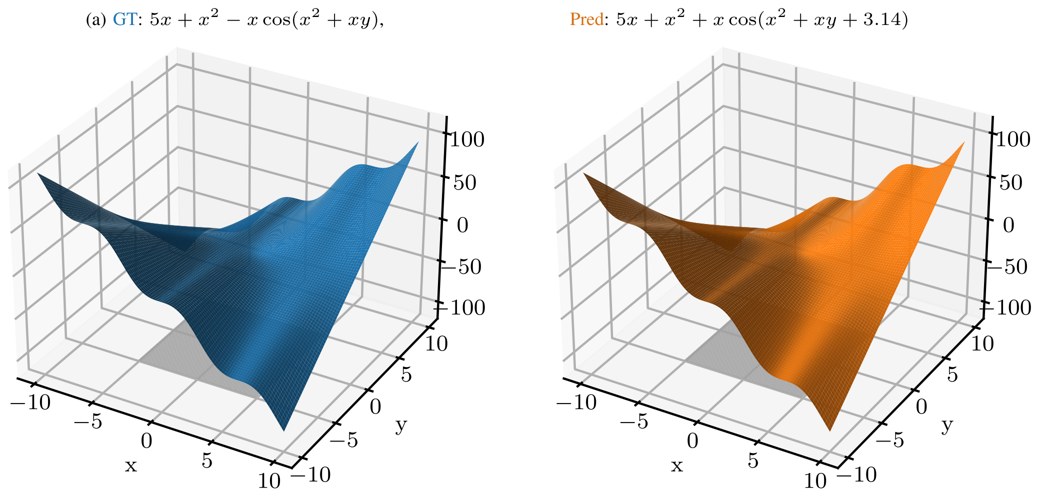

FIGURE 4. Example of model prediction using the Bivariate SymFormer. The inputs *x* and *y* were sampled from the range [−5, 5] × [−5, 5]. 'GT' denotes the ground truth and 'Pred' is the model prediction. We visualize the range [−10, 10] × [−10, 10] to demonstrate the extrapolation abilities.

The hyperparameters were chosen empirically based on several trial-and-error experiments. Systematic hyperparameter tuning is impractical for a transformer application of this complexity. We discovered that using the proposed schedule is essential, as the model needs sufficient precision in the symbolic representation in order for the constant tuning to be meaningful. Without the schedule, the learning process did not converge. However, we found that the schedule is not particularly sensitive to the exact setting of its parameters.

To evaluate the SymFormer's performance, we use 1024 test equations, and for each of them (if not stated otherwise), we generate 256 candidate equations using Top-K sampling with $K = 16$ and further improve them using a local gradient search (LGS) with early stopping. The best candidate equation is then selected based on the lowest error on the input points.

To be able to compare with previous approaches, we used the same metrics and hyperparameters used in these methods, namely the percentage of close points (with atol $=$ 0.001 and rtol $=$ 0.05) [14], the recovery rate with tolerance (*RR*) introduced in [14] – percentage of equations for which at least 95 % of points fall within a tolerance (atol $=$ 0.001 and rtol $=$ 0.05), the coefficient of determination (R^2) [35], and the average time of prediction (in seconds).

B. IN-DOMAIN PERFORMANCE

Figures 3 and 4 demonstrate SymFormer's ability to predict formulas successfully, showing several plots of the model's predictions. The Univariate SymFormer achieved an R^2 of 0.9995, and when we used the local gradient search, the performance further improved to $R^2 = 1$. The Bivariate SymFormer achieved an R^2 of 0.9996 and $R^2 = 1$ when the local gradient search was employed.

C. PERFORMANCE BASED ON THE NUMBER OF SAMPLES

The transformer used a fixed number of points during training. The trained model, however, is robust to the change in the number of input points. To demonstrate it, we performed an experiment where we varied the number of input points during inference. While the model was trained on 100 points, we observe only 2.9 % and 6.5 % decrease in performance when we use 50 and 20 points respectively, and 3.6 % drop in performance when we increase to 1 000 points.

D. COMPARISON TO ALTERNATIVE APPROACHES

To compare our results to state-of-the-art approaches, we use the Nguyen [36], R rationals [37], Livermore, [12], Keijzer [38], Koza [39], and Constant benchmarks. We strive to make the comparison as fair as possible given the following limitations. 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 number of training points and the ranges they are sampled from.

We compare our approach to two state-of-the-art approaches: the transformer-based Neural Symbolic Regression that Scales (NSRS) [14], which is a pre-trained transformer model, and the RL-based Deep Symbolic Optimization (DSO) [13], trained for each equation from scratch.

We use Top-K sampling with $K = 20$ and 1024 candidate equations with early stopping. In comparison to previously used $K = 16$ and 256 candidate equations, this setting offers larger exploration at the cost of an increased inference time. The results in Table 1 show that SymFormer is competitive in terms of model performance on all the benchmarks while outperforming both NSRS and DSO in the time required to find the equation.

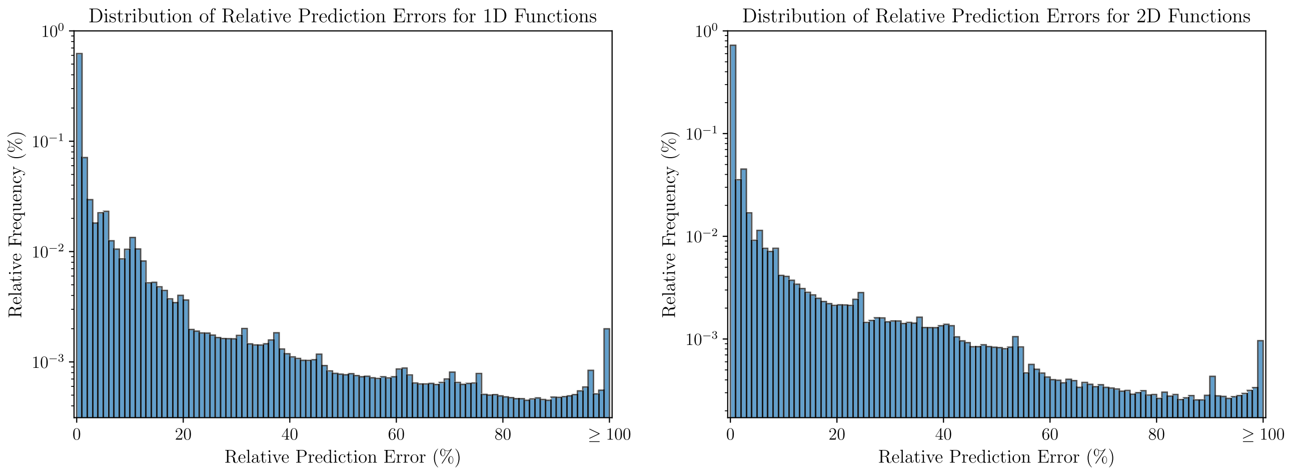

FIGURE 5. Relative prediction errors on all benchmark functions reported in this paper. The logarithmic vertical axis represents the frequency of the error represented by the given bin.

TABLE 1. Comparison of SymFormer with state-of-the-art methods on various benchmarks. 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.

	SymFormer		NSRS [14]		DSO [13]	
Benchmark	$R^2 \uparrow$	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

TABLE 2. Comparison of SymFormer and Bivariate SymFormer performance on all benchmarks. 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.

	SymFormer		Bivariate SymFormer	
Benchmark	$R^2 \uparrow$	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

Furthermore, to demonstrate the Bivariate SymFormer's 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 that Bivariate SymFormer has only slightly worse performance. However, the average inference time increased, which could be explained by the larger search space the model needed to handle during the optimization of constants.

Figure 5 shows the histogram of relative prediction errors, calculated as an absolute value of the difference between the model output and the ground truth, divided by the absolute value of the ground truth plus a constant of 10^{-8}. The relative prediction errors were calculated on all benchmark functions listed in Section IV-D. For each function, 10 000 points were sampled uniformly from their domains. The results indicate that most of the errors are smaller than 10 %.

E. OUT-OF-DOMAIN PERFORMANCE

A desirable property of the algorithm is its ability to predict correct values outside the training data 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 of the training range. More formally, we calculate the metrics on the function values for points sampled from the set $\{x \in \mathbb{R} \mid 5 < |x| < 5 + d\}$, where d is the maximal distance. The effect of the distance on the percentage of close predictions and R^2 can be seen in Figure 6. As we can see from the figure, the model still performs exceptionally well even for larger distances from the sampled domain. This demonstrates that the model generalizes outside the sampled domain and fine-tuning the constants using

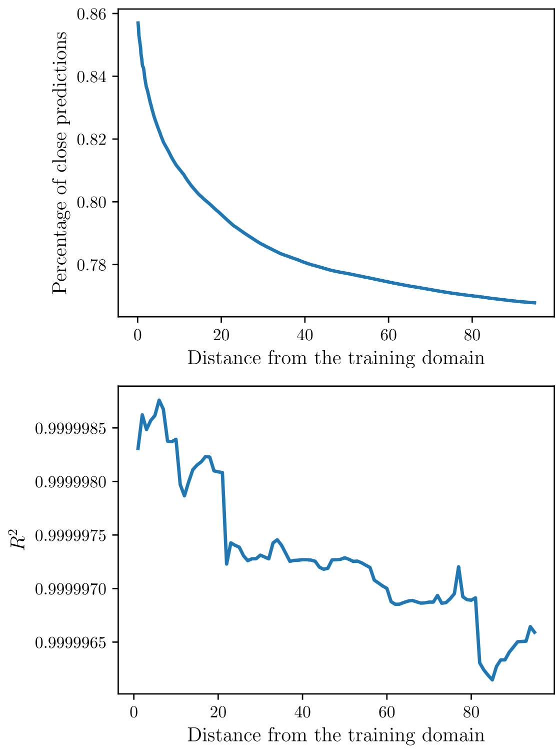

FIGURE 6. The SymFormer's out-of-domain performance.

the local gradient search does not hamper the extrapolation capabilities.

F. STUDY OF CONSTANT ENCODINGS

To analyze the effectiveness of our constants' representation, we conduct a study comparing the representation to other alternative options. In particular, we consider the following variants:

a) ***'No constants* + *BFGS'***: For each constant in the expression, the transformer outputs a special symbol C. At inference, after decoding a final expression, the values of all expression's constants (all symbols C) are found using a posteriori global optimization (Broyden–Fletcher–Goldfarb–Shanno algorithm, BFGS). This setup is the same as in [14].

b) ***'Base encoding'***: The model outputs C for all constants and the regression head is used to predict the constant's values.

c) ***'Base decoding* + *LGS'***: Same training as 'Base encoding', however, at inference, we fine-tune the initial estimates of the constants' values with a local gradient search.

d) ***'Extended encoding'***: Setup described in Section III-D – we split the constants into exponents and mantissas and use special symbols for different constants' exponents. The mantissas are outputted by the regression head. At inference, we do not fine-tune the constants' values after they are decoded.

TABLE 3. Comparison of expression encoding strategies and LGS. SymFormer uses both extended encoding and LGS. We report R^2, RR, and the percentage of close predictions. The base encoding refers to the case when no preprocessing for the constants is employed. BFGS init refers to a situation when the predicted constants serve as a starting point for the BFGS [40]. LGS refers to the case when the gradient search was used to find or improve the constants further.

Model	GS	R^2 ↑	RR ↑	% of close pred. ↑
No constants + BFGS	✓	0.9929	30.40	55.52
Base encoding	✗	0.9979	31.23	50.62
Base encoding + LGS	✓	**1.0000**	53.78	69.09
Extended encoding	✗	0.9995	48.99	70.40
Extended encoding + BFGS init	✓	0.9998	48.37	71.84
SymFormer	✓	**1.0000**	**68.29**	**84.87**

e) ***'Extended encoding* + *BFGS init'***: The training is the same as for the 'Extended encoding', however, at inference, we throw away the initial estimates of the constants' values obtained from the model and run the BFGS from scratch to get the constants' values.

f) ***'SymFormer'***: The final model. We use the exponent-mantissa encoding and at inference, we optimize the values of the constants using an LGS initialized from the decoded values.

From the results in Table 3, we can see that using the constants improves the model's performance in terms of R^2, the recovery rate with tolerance, and also in terms of the percentage of close predictions. One can, therefore, conclude that the performance of SymFormer in comparison to NSRS is better not because of a different dataset or a larger model but because of the use of the constants during training. Furthermore, the last row shows the results for the extended encoding, which uses a local gradient search to further tune the constants. The extended encoding clearly outperforms the base encoding in terms of R^2, the recovery rate with tolerance, and the percentage of close predictions. We believe this to be the case because it is easier for the model to attend to previously generated symbolic tokens than to real values. Therefore, the model can make a more informed decision when predicting the next symbol in the sequence.

V. CONCLUSION

To tackle the problem of symbolic regression, we introduced a novel transformer-based approach called SymFormer that uses a neural network trained on hundreds of millions of formulas. SymFormer is able to efficiently generate a previously unseen formula describing a set of input-output pairs. The model jointly predicts the structure of the formula and the values of all of its constants in a single forward pass of the neural network. A local gradient search is used to tune the constants further. On most of the benchmarks, SymFormer outperforms other state-of-the-art methods not only in terms of R^2 but also in terms of the time required to find the underlying equation. We also validated the importance of the proposed encoding of constants. Finally, by evaluating

SymFormer outside the training range, we demonstrated its remarkable extrapolation capabilities.

Limitations. A limitation of the deep-learning-based architecture is that the number of dimensions and the sampling range are given by the training dataset, and there are no guarantees regarding the behavior outside the dataset's distribution. Even though the inference is fast and efficient, the training uses a lot of compute and takes a considerable time on current hardware.

We have restricted our study to univariate and bivariate functions, most of which did not contain discontinuities or singularities. However, the architecture and training method can easily be extended to include functions of more variables, as well as discontinuous functions. These limitations could be addressed in future work. Furthermore, the method could be extended to model differential equations, as they have a large number of applications in physics and robotics. Analyzing the latent space produced by the encoder may also prove insightful.

REFERENCES

[1] D. Wadekar, L. Thiele, F. Villaescusa-Navarro, J. C. Hill, M. Cranmer, D. N. Spergel, N. Battaglia, D. Anglés-Alcázar, L. Hernquist, and S. Ho, "Augmenting astrophysical scaling relations with machine learning: Application to reducing the sunyaev-zeldovich flux-mass scatter," 2022, *arXiv:2201.01305*.

[2] K. T. Matchev, K. Matcheva, and A. Roman, "Analytical modelling of exoplanet transit specroscopy with dimensional analysis and symbolic regression," 2021, *arXiv:2112.11600*.

[3] J. Kubalík, E. Derner, J. Žegklitz, and R. Babuška, "Symbolic regression methods for reinforcement learning," *IEEE Access*, vol. 9, pp. 139697–139711, 2021.

[4] D. Hein, S. Udluft, and T. A. Runkler, "Interpretable policies for reinforcement learning by genetic programming," 2017, *arXiv:1712.04170*.

[5] C. Wilstrup and J. Kasak, "Symbolic regression outperforms other models for small data sets," 2021, *arXiv:2103.15147*.

[6] M. Schmidt and H. Lipson, "Distilling free-form natural laws from experimental data," *Science*, vol. 324, no. 5923, pp. 81–85, Apr. 2009.

[7] J. Kubalík, E. Derner, and R. Babuška, "Symbolic regression driven by training data and prior knowledge," in *Proc. Genetic Evol. Comput. Conf.*, Jun. 2020, pp. 958–966.

[8] J. R. Koza, *Genetic Programming: On the Programming of Computers By Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[9] N. Staelens, D. Deschrijver, E. Vladislavleva, B. Vermeulen, T. Dhaene, and P. Demeester, "Constructing a no-reference H.264/AVC bitstream-based video quality metric using genetic programming-based symbolic regression," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 23, no. 8, pp. 1322–1333, Aug. 2013.

[10] I. Arnaldo, U.-M. O'Reilly, and K. Veeramachaneni, "Building predictive models via feature synthesis," in *Proc. Annu. Conf. Genetic Evol. Comput.*, New York, NY, USA: Association for Computing Machinery, Jul. 2015, pp. 983–990.

[11] I. Błądek and K. Krawiec, "Solving symbolic regression problems with formal constraints," in *Proc. Genetic Evol. Comput. Conf.*, New York, NY, USA: ACM, Jul. 2019, pp. 977–984.

[12] B. K. Petersen, M. Landajuela, T. N. Mundhenk, C. P. Santiago, S. K. Kim, and J. T. Kim, "Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients," 2019, *arXiv:1912.04871*.

[13] T. Nathan Mundhenk, M. Landajuela, R. Glatt, C. P. Santiago, D. M. Faissol, and B. K. Petersen, "Symbolic regression via neural-guided genetic programming population seeding," 2021, *arXiv:2111.00053*.

[14] L. Biggio, T. Bendinelli, A. Neitz, A. Lucchi, and G. Parascandolo, "Neural symbolic regression that scales," 2021, *arXiv:2106.06427*.

[15] M. Valipour, B. You, M. Panju, and A. Ghodsi, "SymbolicGPT: A generative transformer model for symbolic regression," 2021, *arXiv:2106.14131*.

[16] S. d'Ascoli, P.-A. Kamienny, G. Lample, and F. Charton, "Deep symbolic regression for recurrence prediction," in *Proc. Int. Conf. Mach. Learn.*, 2022, pp. 4520–4536.

[17] G. Dorgo, T. Kulcsar, and J. Abonyi, "Genetic programming-based symbolic regression for goal-oriented dimension reduction," *Chem. Eng. Sci.*, vol. 244, Nov. 2021, Art. no. 116769.

[18] R. Zhang, A. Lensen, and Y. Sun, "Speeding up genetic programming based symbolic regression using gpus," in *Proc. Pacific Rim Int. Conf. Artif. Intell.*, 2022, pp. 519–533.

[19] P. Orzechowski, W. La Cava, and J. H. Moore, "Where are we now? A large benchmark study of recent symbolic regression methods," in *Proc. Genetic Evol. Comput. Conf.*, New York, NY, USA: Association for Computing Machinery, Jul. 2018, pp. 1183–1190, doi: 10.1145/3205455.3205539.

[20] L. Trujillo, L. Muñoz, E. Galván-López, and S. Silva, "Neat genetic programming: Controlling bloat naturally," *Inf. Sci.*, vol. 333, pp. 21–43, Mar. 2016.

[21] M. Kommenda, B. Burlacu, G. Kronberger, and M. Affenzeller, "Parameter identification for symbolic regression using nonlinear least squares," *Genetic Program. Evolvable Mach.*, vol. 21, no. 3, pp. 471–501, Sep. 2020.

[22] L. Trujillo, P. S. Juárez-Smith, P. Legrand, S. Silva, M. Castelli, L. Vanneschi, O. Schütze, and L. Muñoz, "Local search is underused in genetic programming," in *Genetic and Evolutionary Computation*. Cham, Switzerland: Springer, 2018, pp. 119–137.

[23] G. Martius and C. H. Lampert, "Extrapolation and learning equations," 2016, *arXiv:1610.02995*.

[24] S. S. Sahoo, C. H. Lampert, and G. Martius, "Learning equations for extrapolation and control," 2018, *arXiv:1806.07259*.

[25] M. Werner, A. Junginger, P. Hennig, and G. Martius, "Informed equation learning," 2021, *arXiv:2105.06331*.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017, *arXiv:1706.03762*.

[27] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[28] J. Lee, Y. Lee, J. Kim, A. R. Kosiorek, S. Choi, and Y. Whye Teh, "Set transformer: A framework for attention-based permutation-invariant neural networks," 2018, *arXiv:1810.00825*.

[29] P.-A. Kamienny, S. d'Ascoli, G. Lample, and F. Charton, "End-to-end symbolic regression with transformers," 2022, *arXiv:2204.10532*.

[30] J. Lei Ba, J. Ryan Kiros, and G. E. Hinton, "Layer normalization," 2016, *arXiv:1607.06450*.

[31] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, pp. 1929–1958, Sep. 2014.

[32] I. Loshchilov and F. Hutter, "SGDR: Stochastic gradient descent with warm restarts," 2016, *arXiv:1608.03983*.

[33] G. Lample and F. Charton, "Deep learning for symbolic mathematics," 2019, *arXiv:1912.01412*.

[34] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.

[35] S. Glantz and B. Slinker, *Primer of Applied Regression & Analysis of Variance*. New York, NY, USA: McGraw-Hill, 2000. [Online]. Available: https://books.google.cz/books?id=fzV2QgAACAAJ

[36] N. Q. Uy, N. X. Hoai, M. O'Neill, R. I. McKay, and E. Galván-López, "Semantically-based crossover in genetic programming: Application to real-valued symbolic regression," *Genetic Program. Evolvable Mach.*, vol. 12, no. 2, pp. 91–119, Jun. 2011.

[37] K. Krawiec and T. Pawlak, "Approximating geometric crossover by semantic backpropagation," in *Proc. 15th Annu. Conf. Genetic Evol. Comput.*, New York, NY, USA: Association for Computing Machinery, Jul. 2013, pp. 941–948, doi: 10.1145/2463372.2463483.

[38] M. Keijzer, "Improving symbolic regression with interval arithmetic and linear scaling," in *Lecture Notes in Computer Science*. Cham, Switzerland: Springer, 2003, pp. 70–82.

[39] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA, USA: MIT Press, 1994.

[40] R. Fletcher, *Practical Methods of Optimization*. Hoboken, NJ, USA: Wiley, 1987.

MARTIN VASTL received the B.Sc. degree in computer science from Czech Technical University in Prague and the M.Sc. degree in artificial intelligence from Charles University. He currently works in the field of autonomous driving. His research interests include symbolic regression, language modeling, sequence-to-sequence models, and all areas of autonomous driving.

JONÁŠ KULHÁNEK received the B.Sc. degree in computer science from Czech Technical University in Prague and the M.Sc. degree in artificial intelligence from Charles University. He is currently pursuing the Ph.D. degree in computer science, under the supervision of Torsten Sattler. His research focuses mainly on implicit 3-D scene representations and neural rendering, with additional interests in deep reinforcement learning and transformer-based language models.

JIŘÍ KUBALÍK received the M.Sc. degree in computer science and the Ph.D. degree in artificial intelligence and biocybernetics from Czech Technical University (CTU) in Prague, in 1994 and 2001, respectively. He is a Senior Researcher with the Czech Institute of Informatics, Robotics, and Cybernetics, CTU in Prague. His research has mainly focused on various types of evolutionary computation techniques and their applications to hard optimization problems. He is a (co)author of more than 30 papers in this area.

ERIK DERNER received the M.Sc. degree (Hons.) in artificial intelligence and computer vision from Czech Technical University (CTU) in Prague, Czech Republic, and the Ph.D. degree in control engineering and robotics from CTU, in 2022. His research interests include human-centric artificial intelligence, large language models, robotics, sample-efficient model learning, genetic algorithms, and computer vision. The central topics in his research are safety, security, and ethical aspects of generative and conversational artificial intelligence.

ROBERT BABUŠKA (Member, IEEE) received the M.Sc. degree (Hons.) in control engineering from Czech Technical University in Prague, in 1990, and the Ph.D. degree (cum laude) from Delft University of Technology (TU Delft), The Netherlands, in 1997. He was a Faculty Member with Czech Technical University in Prague; and the Electrical Engineering Faculty, TU Delft, where he is currently a Full Professor of intelligent control and robotics with the Department of Cognitive Robotics, Faculty of Mechanical Engineering. In the past, he has made seminal contributions to the field of nonlinear control and identification with the use of fuzzy modeling techniques. His current research interests include reinforcement learning, adaptive and learning robot control, nonlinear system identification, and state estimation. He has involved in the applications of these techniques in various fields, ranging from process control to robotics and aerospace.