Interpretable Machine Learning

Andrius Ovsianas

Supervisor: Prof. M. van der Schaar

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Master of Philosophy

Girton College August 2020

I would like to dedicate this thesis to my loving parents and sister.

Declaration

I, Andrius Ovsianas of Girton College, being a candidate for the MPhil in Machine Learning and Machine Intelligence, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Word count: 10490

Existing Software: The software used in this project is implemented in Python using standard numerical computation libraries such as `torch`, `numpy`, `scipy`, `sympy`. Apart from such packages, we used `gplearn` to implement Genetic Programming, and a bit of software provided by Lample and Charton (2019), which was used to produce figure A.1b.

Andrius Ovsianas
August 2020

Acknowledgements

First, I would like to thank my supervisor Prof. Mihaela van der Scharr for introducing me to this project and providing crucial feedback.

Furthermore, I would like to thank Dr. Ahmed Alaa for our weekly calls, ideas, support and inspritaion throughout the course of the project.

I would also like to thank Francisco Vargas for the numerous stimulating conversations.

Finally, I would like to thank all of my wonderful friends in Swirles. Your precious company has kept me laughing throughout, what one could only describe as, these unprecedented times.

Abstract

The amount of digital information in the world is growing rapidly, and, as a consequence, the analysis of that data is being increasingly automated. Modern Machine Learning models are employed in numerous industries to assist humans in making decisions, or even completely automate certain decision-making. While models such as Deep Neural Networks or Gradient Boosted Trees achieve human-like performance in multiple problems, how they do their predictions is generally not understood. This is a barrier for wider applications of such models in areas where high-stakes decisions are made and the trust in automated systems is paramount.

Interpretability research in Machine Learning is concerned with enhancing trustworthiness of predictive models. Conventionally, researchers have approached this problem by either creating interpretable models in the first place, or proposing ways of extracting insight from already trained black-box models. In contrast, this thesis proposes an alternative idea – building a **model of interpretable models** with deep neural networks. Already the literature surrounding Neural Processes suggests that it is possible to train neural networks that mimic Gaussian Processes – models of functions. Why not extend this idea to interpretable functions?

This work is concerned with symbolic regression – the problem of finding concise mathematical expressions that describe the given data. Previously, symbolic regression was largely approached through slow evolutionary algorithms, rendering this type of models unpopular. We present an alternative approach – building a neural-based architecture which produces symbolic expressions as its outputs. In this framework, solving an instance of symbolic regression corresponds to a single forward pass in our network. Our experiments on synthetic data suggest that, at least in the univariate setting, this approach is superior to a regular Genetic Programming approach in terms of both accuracy and speed.

Table of contents

List of figures

Chapter 1

Introduction

The past decade has been significant for the field of artificial intelligence. Improvements in computational hardware have enabled the use of large models containing millions of parameters. In turn, this led to the sprouting of novel deep neural architectures every year, beating the previous state-of-the-art methods in numerous tasks. However, as these models become larger and more sophisticated, it is more challenging to understand their inner workings. This might not be an issue for applications where only the prediction accuracy is important. However, to be able to deploy them in sensitive areas, where the understanding of the predictive models is crucial, or even regulated (e.g. finance, medicine, law), we must be able to explain why specific predictions are made.

Interpretability of Machine Learning models is a growing field. Many methods have been proposed to explain deep models' predictions or make them interpretable in the first place. The nature of their interpretations ranges from giving examples for particular predictions to discovering the actual functional dependencies between features and targets. As much of deep learning, most of the recent research on interpretability is image-centric, based on attributing importance to different features. Few of the methods focus on capturing more complex relationships. To this end, works such as Rudin (2019) urge researchers to create interpretable models instead of explaining black boxes for high-stakes decisions post-hoc. In this thesis, we propose an alternative approach, which exploits the immense modelling power of deep neural architectures and yet outputs intelligible models. In particular, we suggest developing **models of interpretable models** and provide a specific proof-of-concept approach.

This work is concerned specifically with symbolic regression – constructing compact symbolic equations that fit the given data. Historically, researchers have approached symbolic regression through Genetic Programming algorithms. These have now been studied for a few decades, but are still not in the statistician's toolbox. There have also appeared a few neural architectures trying to tackle symbolic regression over the past few years; however, none have been particularly successful. This is partly due to the fact that expressions are discrete tree structures, and thus it is difficult to frame symbolic regression as optimizing a differentiable loss function.

In this thesis, we present a new approach towards symbolic regression through neural networks – Neural Symbolic Regression. Although currently only successfully applicable for single variable regression, our method solves several important problems: a) the model is trained end-to-end via gradient descent and requires no further optimization other than decoding techniques b) unlike all of the other methods, it is incredibly fast during test time (solving symbolic regression corresponds to a single forward pass in the network). We provide results on synthetic data and suggest ways of extending this to multiple variables.

1.1 Thesis outline

The thesis is organized as follows.

In Chapter 2, we give an overview of the research on Interpretability. We describe the recent works in this area and explain why building interpretable models is crucial.

Chapter 3 focuses solely on the problem of symbolic segression and reviews related work. It rigorously formulates the problem and discusses the possible solutions, such as the standard evolutionary algorithms and the most recent attempts, some of which involve neural-based architectures.

Chapter 4 presents the details of our methodology. In particular, in order to solve symbolic regression through supervised learning we need an annotated set of symbolic expressions – we describe how such a set can be built. Furthermore, we propose a specific neural architecture that can learn to solve symbolic regression from this dataset.

Chapter 5 discusses the results on several synthetic datasets. We show that compared to Genetic Programming, for single variable regression, Neural Symbolic Regression achieves superior performance in case of both speed and accuracy.

And finally, in Chapter 6, we conclude and propose possible future work.

Chapter 2

Interpretability

The field of interpretability is somewhat chaotic. It is partly due to the fact that the notion of *what is interpretable* is incredibly vague. Much of the research takes the interpretability of their approaches for granted, without providing precise, quantifiable evidence that the proposed method is useful for human understanding of why certain predictions were made. In turn, some seminal papers in interpretability take the form of essays describing possible taxonomies to categorize different approaches and their evaluations Doshi-Velez and Kim (2017); Gilpin et al. (2018); Lipton (2016). In addition, they expose increasingly critical and somewhat controversial ideas regarding the conventionally held beliefs Lipton (2016); Rudin (2019).

In this section, we review what the research in interpretability is trying to achieve and show the most common approaches and their shortcomings.

2.1 What is interpretability

In general, interpretability is understood as the ability to explain in terms understandable to a human Doshi-Velez and Kim (2017); Gilpin et al. (2018). Often the terms *explainability* or *transparency* are used instead, but the meaning might differ slightly. For example, in Gilpin et al. (2018) explainability is a combination of two factors: interpretability, i.e. how comprehensible an explanation is, and completeness, i.e. how much useful information about the model it contains. Note that this notion groups all models, whether intrinsically interpretable or not, under the same category. In contrast, other literature seems to refer to explainable ML as a methodology to explain only black-box models Rudin (2019).

The notion of interpretability is domain-specific and depends on the purpose of the interpretable component in the first place Weller (2017). We might need interpretability to enhance the overall user experience and user trust in our technology, to enable easy debugging or development for the engineer, or to make the society comfortable with the usage of such technology in high-stakes areas. Different needs require different forms of interpretability:

while to enhance user experience it might be enough to provide visual examples, to ensure the trust of the society in a statistical model we might need much more elaborate and convincing explanations.

2.2 Motivation

The use of machine learning models for automated decision making is spreading to sensitive high-stakes areas such as credit rating prediction Lipton (2016), healthcare decision support Pekkala et al. (2017); Weng et al. (2017), parole and bail decisions Tan et al. (2018). While automation is desirable, we have to be careful and ensure that the designed systems are robust and operate well. Already, some fundamental general regulations for algorithmic decisions are being put in place Carvalho et al. (2019) and entities at the international level are starting to recognize that such models have to be audited accordinglyCarvalho et al. (2019). Inevitably, the future models used for highly sensitive applications are going to be interpretable.

Why is interpretability important? Simply put, machine learning algorithms are faulty. While in product recommender systems the cost of making a mistake is minuscule, an error in financial decisions can be life-changing, or an error in healthcare can be fatal. In these areas, the levels of trust in the system must be incredibly high in order for them to be deployed. On the other hand, understanding how a system works enables us to improve it.

Given that our models produce accurate predictions, how can they be faulty, and why do we mistrust them?

One reason is that in many applications, there is a discrepancy between the optimization objective and the true objectives the users might have Doshi-Velez and Kim (2017); Lipton (2016). While the models are trained to maximize prediction accuracy, there might be other goals that are more difficult to specify, such as privacy, racial or gender-based bias. For example, in the community of discrimination-aware machine learning Hardt et al. (2016); Zliobaite (2015), the goal is to learn unbiased-models from biased data. Clearly, there are two competing objectives: we **do** want to have an accurate model, but not at the cost of discriminating based on certain features. While the first goal is easy to specify, rigorously defining discrimination is difficult (several competing notions of it can be found amongst the literature).

Another explanation could be that datasets are not representative of the real world, or the problem is non-stationary. This is especially the case with product recommender or advertisement systems where the dynamics of the environment are always changing.

I like to draw the parallel between model interpretability and source code readability. It is not for no reason that most open-source software projects have specific style guidelines and strict code review. Well written code is desirable because it is easy to debug, maintain, develop and explain. If it suddenly does not perform as expected, at least it is possible to figure out why and make sure that it does not happen again. Similarly, data analytics is an iterative process that builds upon interpretations of previous iterations and therefore gains from interpretability and flexibility.

An excellent example of this can be seen in Caruana et al. (2015), where the authors describe a dataset containing data about patients that have pneumonia and analyze models that predict the probability of death. The data indicated that people with asthma have a lower probability of dying, which is mainly because these patients were treated more aggressively. A high accuracy neural network has been developed; however, it has learned the same tendency and therefore, could not be used for clinical trials. In contrast, the new interpretable GA^{2}M model catches that tendency as well, but the term corresponding to asthma can be easily adjusted according to domain-specific knowledge.

2.3 Taxonomies

As mentioned, several papers have proposed ways of categorizing methods in interpretability Doshi-Velez and Kim (2017); Gilpin et al. (2018); Weller (2017). We go through some of them.

Models are usually either *intrinsically* interpretable or are made interpretable by some *post hoc* analysis. Although usually the former is considered to provide more transparent systems, it is worth noting that it depends on the model size as well - linear regression with hundreds of parameters is hardly interpretable. Models that are in general considered to be intrinsically transparent include linear models, generalized additive models (GAMs), GAMs with interactions (GA^{2}Ms), rule sets, decision trees and symbolic expressions. We discuss methods of interpretability for the models of the second type below.

Interpretability methods can be grouped according to whether they are *global* or *local*. Global explanations provide information about how the model makes decisions in general, while local explanations tend to give insight into why a certain prediction was made. For example, it might be difficult to describe concisely how a credit rating model works, but it is likely easier to give a reason for why a particular application was rejected.

Methods of interpretation can be either *model-specific* or *model-agnostic*. Model-specific interpretations are restricted to a particular class of models and utilize that class' properties, while model-agnostic interpretations can be applied to any models and can work with access only to the predictions of the black-box model.

Explanations can also be categorized according to the degree of transparency that they provide. The highest degree of transparency empowers the user to simulate the whole algorithm by themselves; for example, a linear model with few parameters. In contrast, lower degrees of transparency might only enable them to think about the model in terms of its decomposable units (e.g. hidden layers, representations in DNNs).

2.4 Methods

2.4.1 Model-agnostic methods

As mentioned above, model-agnostic methods provide interpretations without any assumptions on the model internals. Mostly they give useful statistical feature summaries or feature importance scores that the user can then visualize Molnar (2019). These methods range from plots representing relationships between feature and target values to local or global surrogate models.

Statistical summaries

For example, Partial Dependence Plots (PDPs) Friedman (2001) are produced by varying one or two features and marginalizing out the rest. This yields a *global* description of how those features impact the target. More precisely, it helps us determine if, on average, the chosen features impact the prediction monotonically, or maybe quadratically. In contrast, for Individual Conditional Expectation plots (ICEs) Goldstein et al. (2013), one feature is chosen and varied in a particular range for every sample of the dataset. This produces a set of trajectories, showing how a feature impacts predictions for every individual sample. Therefore it gives a *local* interpretation, maybe providing us with some insight towards the heterogeneity of the feature. These two methods are simple and operate under an unlikely assumption that the investigated feature is uncorrelated with the rest. Accumulated Local Effects plots (ALEs) solve that problem for PDPs by computing conditional distributions; however, an analogous solution for ICEs does not exist because of the nature of the approach.

LIME

Local Interpretable Model-Agnostic explanations Ribeiro et al. (2016), or LIME, is a local surrogate approach that approximates a given black-box $f : \mathbb{R}^d \to \mathbb{R}$ with an interpretable model around a selected data point $\mathbf{x} \in \mathbb{R}^d$. The approach of LIME is the following.

We first choose an interpretable data representation. Usually, it is chosen to be a binary vector $\mathbf{x}' \in \{0,1\}^{d'}$ indicating the presence of some high-level interpretable features. For example, in computer vision applications, it is usually chosen to be a vector encoding the presence of certain superpixels rather than the raw pixel values. In natural language processing applications, it tends to be a vector illustrating the presence of specific words, rather than their embeddings or the sentence itself. Formally, the mapping between the simplified and original input spaces is defined as the function $h_{\mathbf{x}}(\mathbf{x}')$. For example, in the superpixel case, if all entries of the interpretable representation are 1, the mapping recovers the image, i.e. $h_{\mathbf{x}}(\mathbf{1}) = \mathbf{x}$.

We then fit the interpretable model $g : \{0,1\}^{d'} \to \mathbb{R}$. Because we want the surrogate model to explain f well locally, we sample data points $\mathbf{z}$ around $\mathbf{x}$ and random encodings

$\mathbf{z}' \in \{0,1\}^{d'}$, and compute $f(h_{\mathbf{z}}(\mathbf{z}'))$. The pairs $\{(\mathbf{z}', f(h_{\mathbf{z}}(\mathbf{z}'))\}$ are then used to train g. We can now inspect g to understand the local behaviour of f and calculate statistics such as local feature importance.

While LIME has attracted a lot of attention, it does carry several downsides. First, it has been observed to provide very unstable explanations for points very close to each other. Second, the sampling strategy around $\mathbf{x}$ makes a big difference towards the final results; however, there is no clear way how to choose it. Finally, while it is a local surrogate method, it is not clear how that alone is advantageous if the only reason it is currently used for is feature importance. In fact, in Lundberg and Lee (2017), LIME is described as an additive feature attribution method.

SHAP

Another idea in feature importance is that of Shapley values Lipovetsky and Conklin (2001). Shapley values are commonly used in cooperative game theory to represent the input of each individual in the game towards the total reward. To formally define Shapley values for our purposes, consider a black-box model $f : \mathbb{R}^d \to \mathbb{R}$ and an instance $\mathbf{x}$ for which we want to explain the prediction $f(\mathbf{x})$. Suppose that our set of features is defined by F. We define a function $e : \mathcal{P}(F) \to \mathbb{R}$ such that for a subset $S \subset F$, the value $e(S)$ is equal to the prediction $f(\mathbf{x})$ if the model f were trained solely on the features S. The Shapley value for the feature i is then given by

$$\phi_i = \sum_{S \subset F \setminus \{i\}} \frac{|S|!(|F| - |S| - 1)!}{|F|!} \left[e(S \cup \{i\}) - e(S)\right]$$

It essentially says the following: if we build our set of features by including one feature at a time in random order and assign the change in the prediction that a feature causes to its importance, on average the importance assigned to feature i will be ϕ_i.

Although in the model-agnostic setting we do not have access to retraining the model with different sets of features, there are ways to calculate ϕ_i using Monte-Carlo approximations by sampling random values for the features not in the subset S.

An incredibly important result was shown in Lundberg and Lee (2017). The authors noticed that LIME, Shapley values and other feature importance approaches fall under the category of additive feature attribution methods. These methods assign importance ϕ_i to variables such that the local explanation model g satisfies

$$g(\mathbf{z}') = \phi_0 + \sum_{i=1}^{d} \phi_i z'_i$$

The authors proposed three desirable properties that an additive feature attribution method should satisfy – accuracy at the given point, 0 weights on missing values, and consistency – and proved that Shapley values are a unique solution. In addition to these results, Lundberg and Lee (2017) proposed efficient methods of computing Shapley values in the model-agnostic setting and proposed a couple of methods for the model-specific setting.

2.4.2 Model-dependent methods

In contrast to model-agnostic methods, model-specific approaches assume the class of black-box models they are trying to explain. As such explanations are largely desired for neural networks, most of the approaches are designed for neural architectures Gilpin et al. (2018). For the following discussion, suppose that $f : \mathbb{R}^d \to \mathbb{R}$ is a neural network model and let $\mathbf{x}$ denote its inputs, in most cases an image.

Saliency maps and importance attribution

Saliency maps is a technique to attribute importance to different features in the image for a particular prediction of a deep neural network. They result in images showing which areas of the input image are the most relevant. One of the earliest and most basic approaches towards constructing local saliency maps is to compute the gradients of network outputs with respect to the input image Simonyan et al. (2013), i.e.

$$\frac{\partial f}{\partial \mathbf{x}}$$

This is simple to compute using backpropagation. Another simple way to construct saliency maps is to repeatedly occlude parts of a given image and record what score the network outputs for the class of interest Zeiler and Fergus (2013). These scores can them be plotted as a heatmap to show which parts of the input image are the most crucial in predicting a particular class.

In addition to local saliency maps, one can build a global class model of a particular class Simonyan et al. (2013). The idea is to find an input image that maximizes the score of the given class predicted by the neural network. This can be achieved by simply employing gradient descent. The provided images are useful for understanding what patterns activate those classes. This can also be extended to hidden neurons in order to qualitatively analyze what they are individually responsible for.

The downside of the approaches above is that they are prone to neuron saturation, in that the saturated neurons backpropagate gradients close to 0. A more principled approach towards importance attribution in neural networks was proposed in Shrikumar et al. (2017); Sundararajan et al. (2017). Both papers describe the notion of the reference input $\mathbf{x}'$ – it is seen as a neutral input that we want to measure our importance against and usually it is chosen to be the black image. To measure the importance of a given input $\mathbf{x}$ these papers raise the following question: what does the input $\mathbf{x}$ have, that the baseline $\mathbf{x}'$ does not, that causes the neural network to output a different prediction $f(\mathbf{x})$?

To answer this question, Sundararajan et al. (2017) propose an approach called the Integrated Gradients. The idea is the following: imagine the baseline input $\mathbf{x}$ being continually transformed to $\mathbf{x}'$ on the straight path between them. At every point in that path the gradient of the output with respect to the input is contributing to the final difference in the prediction between $f(\mathbf{x})$ and $f(\mathbf{x}')$. Therefore, all of the gradients along that path should be accumulated, or integrated. That is, the importance is attributed by calculating the path integral:

$$\text{IntegratedGradient}(\mathbf{x}')_i = \int_{[\mathbf{x}',\mathbf{x}]} \frac{\partial f(\mathbf{s})}{\partial s_i} \mathrm{d}\mathbf{s}$$

Likewise, Shrikumar et al. (2017) define a similar approach, DeepLIFT, that computes "discrete" gradients. These can be calculated much faster, in a manner similar to a single backpropagation. However, the implementation of DeepLIFT is much more convoluted, whereas Integrated Gradients can be computed with several lines of code.

Testing with Concept Activation Vectors

A promising approach towards understanding neural networks is presented in Kim et al. (2017). The philosophy of the paper is the following. Conventional saliency maps provide insight into how sensitive to particular inputs a neural network is in predicting a certain class, or, more succinctly, pixel-level sensitivity. Instead the authors propose a method to detect concept-level sensitivity. For example, given a concept "feather" and a class "bird", they attempt to answer the question: how sensitive is our network to feathers when predicting the score of the class "bird"?

In the paper, concepts are rigorously defined as Concept Activation Vectors (CAVs). Concept Activation Vectors are computed as follows. The user selects a set P_C of examples that illustrate a concept, e.g. various pictures with feathers, and a random set of other images N. Then they select a hidden layer h_l in the neural network. A linear classifier is trained on the activations $h_l(\mathbf{x})$ to predict which inputs belong to P_C and which inputs belong to N. CAV associated with feathers is then defined as the normal $\mathbf{v}_C$ to the decision hyperplane learned by the linear classifier. Note that by definition, in the space of h_l activations, $\mathbf{v}_C$ points towards the concept images P_C.

To compute the sensitivity of the class "birds" towards the concept "feather", the authors propose to compute the directional derivative of the class logits:

$$\nabla f_{l,\text{bird}}(h_l(\mathbf{x})) \cdot \mathbf{v}_C$$

Here the gradient is of the output score of the class bird with respect to the activations in the l-th layer.

Concept activation vectors can now be used to debug neural networks. For example, if our bird detection neural network is working well but not quite perfectly, we can debug it to see which concepts it is sensitive to, e.g. wings, feathers, beaks, and determine what is causing the problem. In addition, we can test if a trained neural network exhibits any racial or gender-based bias by creating CAVs for the secured group and testing its sensitivity.

2.4.3 Explanations

Much of the interpretability research focuses on methods like the ones described above – means to understand complicated black-box models. Most of them offer interpretability through attributing importance to different features. However, that is not sufficient to truly understand deep models.

A rather controversial opinion is presented in Rudin (2019). The author urges the machine learning community to stop explaining black boxes and start building interpretable models in the first place. They argue that explanations produced by saliency maps or attention mechanisms are not faithful to the true model and can often be misleading. Indeed, I agree with such sentiment – it is much more desirable to have transparent models. While one of the main goals of explainability seems to be to enhance the decision-making of humans equipped with black-box models, there seems to be a lack of human-grounded experiments. Out of the works mentioned above only LIME and SHAP have performed them. However, even those are proxy experiments and do not measure the true utility of feature attribution methods. In fact, some papers that have conducted experiments involving humans concluded that feature attribution based explanations have no significant impact Carton et al. (2020); Lai and Tan (2019); Weerts et al. (2019). On the other hand, it was also observed that explanations, regardless of their effectiveness, increase user trust in the black-box model. This suggests that deploying such systems might artificially increase user trust and could be deemed unethical.

On the other hand, the paper does not mention any new promising work on explainability, such as the reviewed Concept Activation Vectors. In addition, it misses the point presented in Gilpin et al. (2018). In a way, an intrinsically interpretable model can also be seen as accompanied by an explanation – the model itself. While this is the most faithful explanation, it might lack interpretability. We can imagine a configuration space of all possible combinations of faithfulness, interpretability and accuracy. Given a particular task, each point in that space induces a certain utility. It is reasonable to assume that the utility would be maximized when all three parameters are. However, the author takes a step further and suggests that much higher utility is achieved at points where faithfulness is maximized and ignores the fact that the utility is task-dependent and could be achieved at different points for different tasks. For example, in tasks where the user has high expertise, the utility function might be different compared to the utility function of tasks where the user is a novice.

Chapter 3

Symbolic Regression

In this chapter, we take a more in-depth look at symbolic regression - the task of learning symbolic relations between the known feature variables and the unknown target variable. It is an old problem that has been studied extensively but previously has mostly been tackled with evolutionary algorithms. Much of the research is based on improving their computational complexity, accuracy or interpretability of the expressions produced. There have also been a few attempts at trying to incorporate neural architectures to crack symbolic regression. Below, we provide a precise problem formulation and describe some of the selected approaches and their properties.

3.1 Problem Formulation

Let $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ be a dataset, where $\mathbf{x}_i = (x_1^{(i)}, \ldots, x_d^{(i)}) \in [0,1]^d$ is a d-dimensional feature vector lying in the unit cube. We assume that the features $\mathbf{x}_i$ and the target values y_i in the dataset are both *continuous* measurements, rather than *categorical* labels.

To solve one instance of the symbolic regression problem is to find a symbolic function $f : [0,1]^d \to \mathbb{R}$ that fits the given dataset D. In this context, a symbolic function is a function that can be written succinctly as a mathematical expression, for example $f(x) = e^{\cos(5x)}$ or $f(x_1, x_2) = x_1 + \sqrt{2x_2}$. The objective of symbolic regression is two-fold:

1. We want $f(x)$ to be as accurate as possible in predicting the data D. That is, we strive to minimize the mean squared error

$$\mathbb{E}_{(\mathbf{x}_i, y_i) \sim D}[(y_i - f(\mathbf{x}_i))^2]$$

2. We want the produced expression $f(x)$ to be *understandable* to a human user.

While the first objective is clear, the second one is rather vague and user-dependant. Typically, it is understood as producing expressions that are parsimonious, containing just a

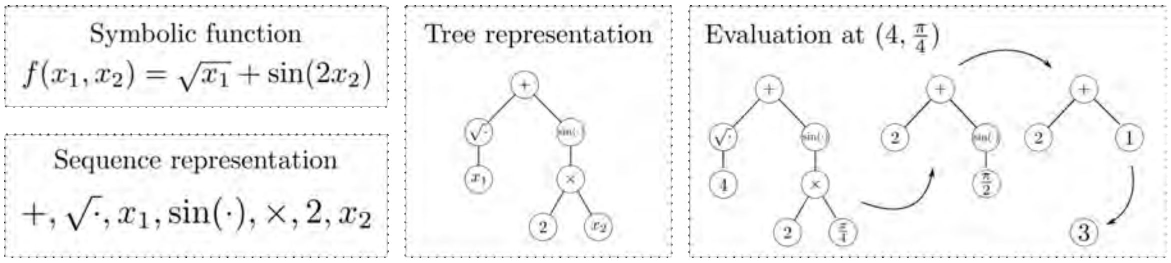

Symbolic function
$f(x_1, x_2) = \sqrt{x_1} + \sin(2x_2)$
Sequence representation
$+, \sqrt{\cdot}, x_1, \sin(\cdot), \times, 2, x_2$
Tree representation
Evaluation at $(4, \frac{\pi}{4})$

Fig. 3.1 Representations of symbolic expressions and their evaluation

few operators. However, this criterion does not take into account which operators the human user understands more easily than others. An ideal symbolic regression solving system, in addition to the dataset D, would also admit as an input a set of operators S that should make up the output expression.

More formally, symbolic expressions are commonly viewed as tree structures (see figure 3.1). Leaf nodes represent constants or variables, while the inner ones represent the operators. Parent nodes and their children correspond to operators and their operands. In this setting, expressions are evaluated bottom-up: first, we set the values of the variables at the leaf nodes, then compute the values of the parent nodes, and continue this way until we reach the root.

Symbolic expressions can also be represented by sequences, where each token is either an operator, variable or a constant. To do so, one can simply perform a preorder-traversal, otherwise known as Polish notation, on the corresponding tree. It acts recursively, by first writing down the label of the parent node and then flattening the children subtrees starting from the left-most one and working its way towards the right.

In general, given a preorder-traversal, it is impossible to determine the original tree that produced it. However, symbolic expressions correspond to a special type of trees that have subtle restrictions - the number of children of each node is determined by its label, or more precisely by the arity of the operator. This is enough information to be able to recover a symbolic expression from its sequential representation. Therefore, such representation of symbolic expressions is well-defined.

3.2 Related work

3.2.1 Genetic Programming

The most common approach towards symbolic regression is that of Genetic Programming (Koza (1992)). In general, the idea behind GP is to start with a set of randomly generated models (*individuals*), i.e. the first generation, and improve them via natural selection, hoping that models in final generations will be accurate. Each generation, every individual is evaluated on the task at hand, and only a fraction of them are kept according to their *fitness* (performance on the training data). To fill in the gap for the discarded individuals, new

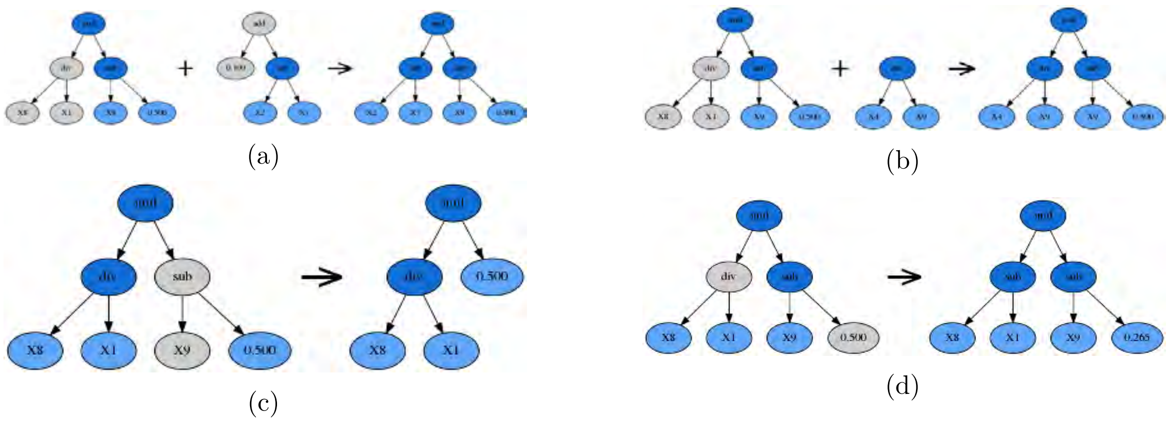


Fig. 3.2 Default mutations in `gplearn`.
a) **Crossover**: given two trees from the population, replaces a subtree of one tree with a subtree from another. b) **Subtree mutation**: given a tree from the population and replaces its subtree with a random tree. c) **Hoist mutation**: given a tree from the population, chooses a subtree, and replaces it with one of that subtree's subtree. d) **Point mutation**: given a tree from the population, chooses a few nodes and replaces them. Images taken from https://gplearn.readthedocs.io/en/stable/intro.html.

individuals are created by *mutating* the best performers and performing *crossovers* among them. These offspring and their parents comprise the population of the next generation, and the process continues.

In symbolic regression, the mutations and crossovers are performed on the tree structure of the expressions. To mutate a tree, one can, for example, randomly choose a subtree (or a node) and substitute it with a completely new random subtree (or a node). To perform a crossover between two symbolic expressions, one can substitute a subtree of one expression with a subtree from another. Figure 3.2 provides examples of the mutations available in the python package `gplearn`.

In classical symbolic regression applications, the fitness of a symbolic expression is defined simply as the prediction error. However, then the produced expressions are likely to be complex and uninterpretable. In the program generation community, this has been a bigger issue, which was referred to as bloating – the increase of individuals' sizes without the increase in fitness. To this end, some research suggests incorporating a complexity term when computing the fitness. The complexity term is usually a function of the size of the expression (e.g. expressional complexity, Searson (2014)) or measures the degree to which the expressions are non-linear (e.g. the order of the Chebyshev's polynomial that fits the given expression, see Vladislavleva et al. (2009)).

There are a few shortcomings to the Genetic Programming approach. First, it is incredibly slow due to its computational complexity: because mutations are quite complex and affect the inner tree structure of expressions, every generation each individual has to be evaluated on the whole dataset D from scratch. Furthermore, the approach is not very stable and prone to producing not well-defined expressions. For example, `gplearn` implementation of symbolic regression sidesteps this by defining the logarithm for negative arguments to be 0, or division to evaluate to 1 if the denominator is very low.

Evolutionary Feature Synthesis

An elegant approach to reduce the computational complexity of Genetic Programming for symbolic regression is called Evolutionary Feature Synthesis (EFS), Arnaldo et al. (2015). The idea is the following: rather than performing natural selection on models, EFS performs it on features that a linear model uses for predictions. Specifically, each generation the population is comprised of basis functions $\{h_m(\mathbf{x})\}_{m=1}^{p}$ that are together used as one linear model $f(\mathbf{x}) = \sum_{m=1}^{p} \beta_m h_m(\mathbf{x})$. The evolution starts with a population of trivial features $h_m(\mathbf{x}) = x_m$. Each generation, a new linear model is trained on the given data using the features in the current population. Linear regression is regularized by L^1 weight penalty to introduce sparsity and produce *feature scores*. The features with the highest scores are combined using predefined unary and binary functions to introduce new features in the population. Finally, a second linear model is trained, and the features with the highest scores are kept in the subsequent generation. EFS is faster than classical GP: new individuals are obtained by quick mutations and can be evaluated using vectorial operations.

3.2.2 Neural architectures

Recently several works have appeard incorporating neural networks in symbolic regression, wildly varying in their approach.

In Li et al. (2019), the authors use asymptotic constraints (i.e. the leading polynomial powers of a function) as an input to an RNN decoder. The decoder is trained to produce expressions that conform to these constraints and then used in Monte-Carlo Tree search to sample expressions and find the ones that fit the data best. The downside of this approach is that it is not clear how to extend it to the multivariate case and the expressions generated are restricted to rational functions.

In Martius and Lampert (2016), the authors suggest using a neural network architecture which uses sines, cosines and multiplication units for activations. The network is then trained on the given data as usual but regularized with L^1 loss to make the weights sparse and the model interpretable. In their following work Sahoo et al. (2018), the authors incorporate division as an operation in the final layer and include L^0 regularization, by forcing all sufficiently small neural network weights to 0. The disadvantage of this approach is that only certain operators can be incorporated into the pipeline.

GrammarVAE

Perhaps the most similar to ours is the GrammarVAE approach, Kusner et al. (2017). The authors train a variational encoder-decoder architecture to learn disentangled representations for symbolic expressions.

Firstly, they train a VAE that encodes and decodes symbolic expressions. Since the paper approaches modelling molecules as well as expressions, they are viewed more generally as being produced by Context-Free Grammars. In turn, a given expression is "flattened" by writing down one-hot-encodings of the production rules that generate that expression. Note that for symbolic expressions this is a bit overcomplicated and one could simply use preorder-traversal (see section 3.1).

As usual in variational inference, the encoder takes the sequence $s_1, \ldots, s_n$ of one-hot vectors and produces a variational distribution $q_\phi\left(\mathbf{z} \middle| s_1, \ldots, s_n\right)$ for the latent variable. Given a representation $\mathbf{z}$ the decoder part of the network uses a recurrent neural network to produce probabilities $p_\theta\left(s_i \middle| \mathbf{z}\right)$ for each of the production rules. While training, these are used in the cross-entropy loss function. During decoding, to ensure that a produced sequence is syntactically correct, before predicting a token, its logits are masked such that invalid production rules would have probability zero.

Having trained the VAE, the authors use the learned expression representations $\mathbf{z}$ to perform symbolic regression. For a particular instance of the problem, they train a Gaussian Process to, given a representation $\mathbf{z}$, predict the mean squared error of the associated expression. Finally, using this GP, they employ Bayesian Optimization to find the expression that fits the data best.

Notice that the learned representations $\mathbf{z}$ only hold the *syntactic* information about the tree structure of the symbolic expression and not the *semantic* information about its functional properties. In turn, a separate Gaussian Process model has to be trained for each instance of the symbolic regression problem. This is what makes the approach slow.

Finally, note that the decoding procedure in GrammarVAE is problematic. The logits produced are conditional on the representation $\mathbf{z}$ and not the previous production rules. More precisely, this means that once a production rule s_1 is sampled from $p_\theta\left(s_1 \middle| \mathbf{z}\right)$, the next rule is sampled from $p_\theta\left(s_2 \middle| \mathbf{z}\right)$ and not $p_\theta\left(s_2 \middle| s_1, \mathbf{z}\right)$. Not only is this a choice in the decoding procedure, but it is also enforced by their choice of the network architecture: the inputs to recurrent neural network are identical copies of $\mathbf{z}$ rather than the logits of the production rules. We solve this problem by initializing the hidden state of the RNN as $\mathbf{z}$ and providing the logits of the rules as inputs.

3.2.3 Symbolic Regression through Meijer-G functions

A recent novel approach towards symbolic regression is presented in Ahmed M. Alaa (2019). The work is based on the properties of Meijer-G functions, better known in the mathematics community. Meijer-G functions are univariate functions defined on $\mathbb{C}$, which were designed to capture many of the known complicated functions as its special cases. It is parameterized by two vectors $\mathbf{a} \in \mathbb{C}^p$, $\mathbf{b} \in \mathbb{C}^q$, two integers m, n and defined through a complex integral over a path L:

$$G_{p,q}^{m,n}\left(\begin{matrix}\mathbf{a}\\ \mathbf{b}\end{matrix}\middle| z\right) = \frac{1}{2\pi i}\int_L \frac{\prod_{j=1}^{m}\Gamma(b_j - s)\prod_{j=1}^{n}\Gamma(1-a_j+s)}{\prod_{j=m+1}^{q}\Gamma(1-b_j+s)\prod_{j=n+1}^{p}\Gamma(a_j-s)} z^s \mathrm{d}s$$

There are three choices for the path L (their descriptions are quite tedious), depending on which the conditions for convergence of the integral change, but the values for all three agree.

The reason Meijer-G functions are attractive is because of the algorithm described in Roach (1996) – given the function parameters, it produces a corresponding symbolic expression. It is implemented in standard libraries, such as `sympy`. To perform multivariate symbolic regression, the paper suggests using utilizing Kolmogorov's superposition theorem:

$$g(\mathbf{x}) = g(x_1, \ldots, x_n) = \sum_{i=0}^{r} g_i^{\text{out}}\left(\sum_{j=1}^{d} g_{ij}^{\text{in}}(x_j)\right)$$

In this setting Meijer-G functions can be used as basis functions $g_{i,j}^{\text{in}}$ and g_i^{out}, and the whole model can be trained end-to-end via gradient descent.

Although initially this project was designed to build on top of this research, several drawbacks of this approach were discovered, which we found difficult to overcome:

- The mentioned algorithm Roach (1996) can find interpretable symbolic representations only for very few configurations of $\mathbf{a}$ and $\mathbf{b}$. This is because it is based on a table consisting of known expressions of the integral for specific values of the parameters. To find a symbolic expression for parameters not in the table, the algorithm uses differential shift operators, which relate Meijer-G functions with one another, for example[1]:

$$\left(\frac{z}{a_i}\frac{\mathrm{d}}{\mathrm{d}z} + 1\right) {}_pF_q\left(\begin{matrix}\mathbf{a}\\ \mathbf{b}\end{matrix}\middle| z\right) = {}_pF_q\left(\begin{matrix}\mathbf{a}+\mathbf{e}_i\\ \mathbf{b}\end{matrix}\middle| z\right)$$

  Because of the nature of these shift operators, the algorithm can find symbolic expressions only to those parameters that are reachable by moving around in the parameter space using these shift operators.

  In addition, even if the algorithm does find a formula, there is no guarantee that it is interpretable. For example, take a look at one of the expressions in the table of `sympy`'s `hyperexpand`[2]:

[1] more precisely, the algorithm operates by converting the Meijer-G function to a sum of generalized hypergeometric functions ${}_pF_q$ and then finds symbolic expressions for them

[2] here $S(x)$ and $C(x)$ are Fresnel functions

$$ {}_1F_2\left(\begin{matrix}1\\ \frac{3}{4},\frac{5}{4}\end{matrix}\middle| z\right) = \frac{\sqrt{\pi}\left(i\sinh(2\sqrt{z})S\left(\frac{2\sqrt{4}ze^{\frac{i\pi}{4}}}{\sqrt{\pi}}\right)+\cosh(2\sqrt{z})C\left(\frac{2\sqrt{4}ze^{\frac{i\pi}{4}}}{\sqrt{\pi}}\right)\right)e^{-\frac{i\pi}{4}}}{2\sqrt[4]{z}} $$

- There is no known closed-form expression for the gradient of the Meijer-G function with respect to its parameters. This means the only way to generally incorporate them into gradient-based end-to-end systems is to use finite-difference approximations.
- The Meijer-G function framework is not very flexible, and it is not designed for learning. While neural networks offer many possible architectural changes, Meijer-G function representation is fixed, and it is not clear how one could modify it. In addition, for a specific parametrization of functions (e.g. ANNs or polynomials) to do effective learning, it is not enough for them to contain highly flexible functions, as shown in the case of polynomial regression. The parametrization must also offer suitable loss surfaces for gradient descent in many different tasks, as it was repeatedly empirically shown with ANNs. It is not clear that Meijer-G functions provide such a parametrization.

Although this approach is exciting and new, we decided not to use it in this project and instead try and design a neural-based architecture for symbolic regression.

3.2.4 Symbolic Mathematics through Deep Learning

Although not on symbolic regression, the work of Lample and Charton (2019) is very elegant. The authors approach the task of indefinite integration through the means of deep learning. The paper draws inspiration from the fact that humans mostly perform integration not by using mathematical properties of the integrand, but by inspecting how the symbolic expression for that integrand looks like symbolically (e.g. integration by parts and change of variables). This suggests that the task could be phrased as "*translate* the integrand $f(x)$ to $F(x) = \int f(x)$", and solved by current state-of-the-art NLP techniques. This is essentially what the approach in the paper is – the authors train a transformer architecture to translate integrands to their integrals. Symbolic expressions are written down in their sequential representation and then translated by a seq2seq model to their integrals. For example, the function $f(x) = x^2 + x^{-2}$ is viewed as the sequence `[+, pow, x, 2, pow, x, -2]`, and integrating $\int f(x) = \frac{1}{3}x^3 - \frac{1}{x} + C$ is viewed as translating that sequence into `[+, -,` $\times$`, 1/3, pow, x, 3, pow, x, -1, C]`.

The reason I mention this work is because the method we designed in this paper is inspired by taking the task of translating symbolic expressions to symbolic expressions a step further – translating datasets to symbolic expressions.

3.3 Interpretability of Symbolic Expressions

Even though in this work (and most of SR research) we assume that compact symbolic expressions are intrinsically interpretable, it is important to raise the question - what are their limitations? It is difficult to answer this question; however, we outline a few obvious properties.

Consider the following function[3]:

$$f(x) = \exp\left\{\cosh\left(\frac{x + 2x^2 + \cos(x)}{3 + \sin(x^3)}\right)\right\}$$

It is a concise expression and many properties of it are known (smoothness, positiveness), but just by looking at it, some questions are difficult to answer: is it monotone? Is it unimodal? Is it convex? Symbolic expressions are interpretable only as much as the amount of information we can get by trivial analysis.

Moreover, some datasets might be too discrete for there to exist a concise equation. Symbolic Regression probably will not produce interpretable expressions for data that is generated by a decision tree.

Finally, symbolic expressions are not suitable for categorical or high-dimensional data, and just by themselves do not provide any uncertainty estimates.

[3]This example is taken from one of the early works of Bayesian Probabilistic Numerics Diaconis (1988), where it's presented for different reasons

Chapter 4

Methodology

This chapter goes through the details of our model. Untraditionally we describe our training data prior to the model, as it allows us to easily show the underlying difficulties of the problem.

4.1 The Big Picture

To solve symbolic regression, we propose a deep architecture – Neural Symbolic Regression (NSR). Our approach builds on an idea that none of the reviewed works incorporate – learning representations of data that can be decoded to expressions. In particular, GrammarVAE and the discussed integration architecture Lample and Charton (2019) both encode and decode symbolic expressions. Consequently, the learned representations only contain information about the *syntactic* information of the tree structure of the symbolic expression.

In contrast, NSR is a deterministic encoder-decoder architecture that admits datasets as its input and outputs the corresponding symbolic expressions in their sequential representation. In turn, our network learns and exploits the relationship between the *semantic* information about the curvature of a function and the *syntactic* information about its symbolic representation.

NSR is a **model of models**. This has two consequences:

1. In this setting, to utilize supervised learning and train the architecture with gradient descent end-to-end we need a collection of training examples – datasets $\{(\mathbf{x}_i, y_i)\}_{i=1}^{m}$ annotated with the symbolic function $f(\mathbf{x})$. That is, our training dataset is a **dataset of datasets**. To get rid of this ambiguity, in this paper, we only refer to it as the training set. Any other mention of a dataset refers to sets of the form $\{(\mathbf{x}_i, y_i)\}_{i=1}^{m}$.

   To obtain the training set, one has to generate symbolic expressions (annotations) and evaluate them to produce datasets (inputs). This is not a straightforward task and is further discussed in section 4.2.

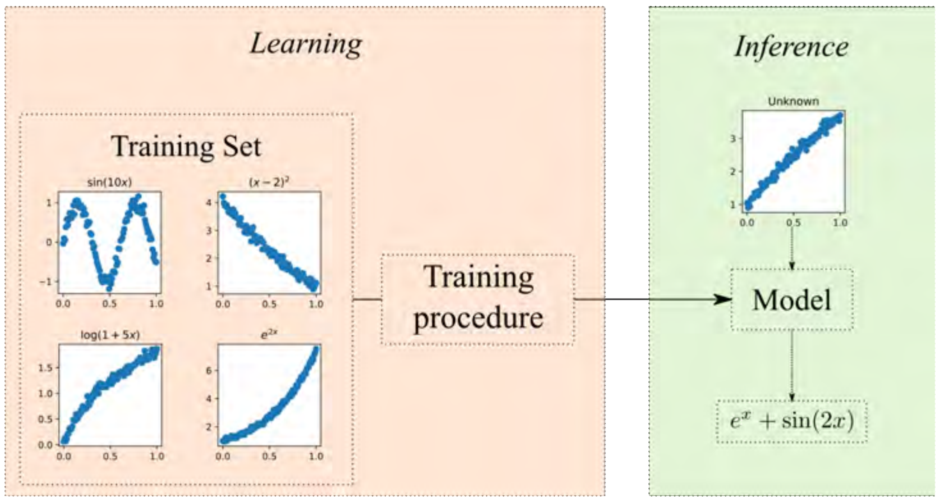


Fig. 4.1 The pipeline of the NSR model

2. Inference in NSR corresponds to inferring a model – that is, solving symbolic regression is a single forward pass in the network, which is incredibly fast compared to previous approaches.

In this chapter, we focus on univariate symbolic expressions, where $\mathbf{x}_i = x_i \in [0, 1]$. This is clearly a limitation and extending this architecture to multiple dimensions is difficult. We describe our attempts in Chapter 5.

The big picture of our approach is illustrated in figure 4.1. First, we generate a training set that will be used in our learning procedure. Then we use it to train our NSR model using gradient descent in an end-to-end fashion. Finally, we evaluate NSR on numerous datasets and, more generally, use it for symbolic regression. In the next few sections, we explain these steps in detail and the following chapter presents the results.

Our model is partly inspired by the series of recent papers on Neural Processes, where different techniques were developed to build models on datasets Garnelo et al. (2018a,b); Gordon et al. (2019); Kim et al. (2019). In turn, our architecture shares a lot of the structure with the one used for Conditional Neural Processes Garnelo et al. (2018a) (see figure 4.2).

4.2 Generating the Training Set

To be able to train NSR, we need training examples consisting of symbolic expressions and their evaluations. During test time, we want NSR to be accurate on a large class of functions, and therefore our training set must contain such examples.

Generating them is quite complicated and tedious. For example, in their work Lample and Charton (2019) point out an issue: naively sampling expressions favours trees that are more likely to be deep than broad or left-leaning rather than right-leaning. For their purposes, to produce samples of expressions, they propose a dynamic programming approach, which is not biased in such ways. In our case, even though we might not care about the structural properties of the sampled symbolic expressions, we need expressions that are well-defined and exhibit a rich variety of features once evaluated.

As we were seeking for a proof of concept method, we decided to put this issue aside and devised a deterministic ad-hoc algorithm to build our training set. The full description of it is provided in the Appendix. Essentially, it starts with simple trees and builds new ones incrementally, only keeping those that produce graphs different enough from the ones already in the training set.

To generate the training set our algorithm solves the following problems:

- **Well-defined functions on [0, 1]**. Random sampling can produce expressions that are not well-defined, such as $\sqrt{-1 - x^2}$. Note that, this is not an issue for either GrammarVAE Kusner et al. (2017) or the integration architecture Lample and Charton (2019), since their encoder-decoder architectures encode expressions rather than data.

- **Rich features**. Random sampling (or the algorithm described in Lample and Charton (2019)) generally produces functions that are either very flat and monotonic or have incredibly high variance or values. Figure A.1 shows compares functions sampled from our training set and the ones generated randomly.

- **Explosive values**. Importantly, note that it is challenging to produce simple **discrete** expressions that attain values in a relatively reasonable range, for example [-100, 100] (partly this is because of the polynomial and exponential family functions).

  One might suggest normalizing them: rather than including $e^{e^{10x}}$ to our training set, include $\frac{1}{M}e^{e^{10x}}$ (where M is its maximal value). However, that would mean that our model would have to be able to predict incredibly large constants very accurately (not to mention that it has a finite vocabulary of tokens).

  Instead, we take a more general approach to deal with this issue. Rather than normalizing expressions, we normalize their values before they are given to the model. That is, our dataset might contain functions like $e^{e^{10x}}$[1], but the data $\{(x_i, y_i)\}_{i=1}^{m}$ that our model observes is obtained by:

[1]although likely this expression causes numerical overflow and would be discarded

$$\begin{aligned} x_i &\sim U[0,1] \\ z_i &= e^{e^{10x_i}} \\ \mu &= \text{mean}(\{z_i\}) \\ \sigma^2 &= \text{var}(\{z_i\}) \\ y_i &= \frac{z_i - \mu}{\sigma} \end{aligned}$$

On the test data, after predicting an expression, we will evaluate it on several points and normalize in the same way.

This means that our model will not be burdened by arbitrary scaling and the task could be seen instead as "given a dataset produce an expression with similar curvature".

- **Simple expressions**. Finally, for interpretability reasons, we want our network to produce simple expressions. To make sure that the training set contains such expressions, our algorithm, therefore, starts with elementary functions like $f(x) = x$ and $f(x) = 0$ and includes new ones only if their graph is unique enough.

The resulting dataset contains 60000 symbolic expressions. They were generated using 21 tokens: the variable `x`, constants `1-10`, unary operators `[sin, cos, exp, ln, pow2, pow3, negate]` and binary operators `[mul, add, sub]`. We randomly picked a subset of 5000 expressions that were used to validate the generalization of the model.

4.3 Model

4.3.1 Definition

NSR is a probabilistic discriminative model that predicts the sequencial representation $s_1, \ldots, s_n$ of a function $f(x)$, given a dataset $\{(x_i, y_i)\}_{i=1}^m$ that was generated by $f(x)$, i.e.

$$p_\theta\left(s_1, \ldots, s_n \middle| \{(x_i, y_i)\}_{i=1}^m\right) = \prod_{j=1}^{n} p_\theta\left(s_j \middle| s_1, \ldots, s_{j-1}, \{(x_i, y_i)\}_{i=1}^m\right)$$

In this section, $s_1, \ldots, s_n$ are regarded as one-hot vectors encoding the operator, constant or the variable that it represents. The architecture of our model is presented in figure 4.2. It is divided into 3 stages:

1. **Encoding datapoints**. Each data point (x_i, y_i) is encoded into a representation r_i via a simple multilayer perceptron:

$$r_i = \text{MLP}_p \quad i = 1, \ldots, m$$

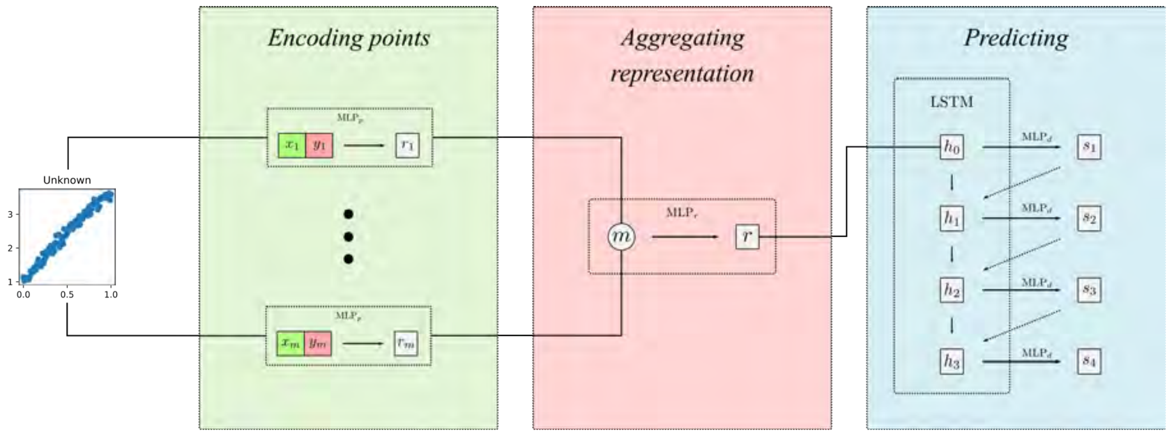


Fig. 4.2 Model architecture

2. **Computing dataset representation**. The datapoint representations r_i are aggregated using the average function and then given to another multilayer perceptron:

$$r = \text{MLP}_r(\text{mean}(\{r_i\}_{i=1}^m))$$

3. **Predicting the sequential representation**. We initialize the hidden state of the LSTM and predict the first token:

$$\mathbf{h}_0 = r$$
$$\mathbf{c}_0 = 0$$
$$p_\theta\left(s_1 \middle| \{(x_i, y_i)\}_{i=1}^m\right) = \text{Softmax}(\text{MLP}_d(\mathbf{h}_0))$$

Given s_{j-1} and $\mathbf{h}_{j-2}, \mathbf{c}_{j-2}$ we can then compute the next hidden state and the conditional distribution of s_j:

$$\mathbf{h}_{j-1}, \mathbf{c}_{j-1} = \text{LSTM}(s_{j-1}; \mathbf{h}_{j-2}, \mathbf{c}_{j-2}) \tag{4.1}$$
$$p_\theta\left(s_j \middle| s_1, \ldots, s_{j-1}, \{(x_i, y_i)\}_{i=1}^m\right) = \text{Softmax}(\text{MLP}_d(\mathbf{h}_{j-1}))$$

Notice that in 4.1 we require the label of s_{j-1}. During training we use *teacher forcing*, where this label is provided as the gold truth value of s_{j-1}. At test time, this label is given by beam search, which selects several most likely tokens (see section 4.3.2).

We optimize the model parameters by maximizing the likelihood of observing the sequence $s_1, \ldots, s_n$ given the data $\{(x_i, y_i)\}_{i=1}^m$:

$$\mathcal{L}(\theta) = \mathbb{E}_{E \sim \mathcal{T}}\left[-\log p_\theta\left(s_1, \ldots, s_n \middle| \{(x_i, y_i)\}_{i=1}^m\right)\right]$$

Here $\mathcal{T}$ is our training set, E is a randomly sampled expression from $\mathcal{T}$. $s_1, \dots, s_n$ is the sequential representation of E and $\{(x_i, y_i)\}_{i=1}^m$ is a dataset produced using E as descibed in section 4.2.

4.3.2 Decoding procedure

During test time, there are several choices to decode the expressions. One option is to sample one token at a time from the conditional distributions that our model provides us. Another is to take the most likely token at each step. In our work, we take the conventional approach of beam search. Beam search attempts to find the most likely sequence according to our model p_θ without performing an exhaustive tree search. Instead, it starts with an empty sequence and considers all possible next symbols, keeping a fixed number of those that have the highest likelihood. At every next iteration, the algorithm expands every sequence of tokens at hand but only keeps a limited number of those, that have the highest log probability. Once a certain depth is reached the most likely sequence is picked. Note that the choice of the decoding strategy is independent of the model itself and any of the above could be used.

Our implementation of beam search differs from the conventional approach by a couple of details. First, in our case, not all sequences correspond to valid expressions. In turn, our version of beam search tracks the additional number of tokens that have to be generated to complete the expression (Kusner et al. (2017) describe a similar decoding strategy using stacks). Second, once the beam search generates all candidate symbolic expressions, we pick the candidate not by its log probability, but by how well it fits the given data, that is, its MSE score.

Algorithm 1 Decoding algorithm

```
    Input: A dataset $\{(x_i, y_i)\}_{i=1}^m$, beam width $W$ and depth $D$
    Output: An expression $E$
 1: initialize an empty list of beam waves $L$
 2: add a wave with a single empty token to $L$
 3: for $i = 1, \dots, D$ do
 4:     initialize an empty list of candidate sequences $S$
 5:     for $j = 1, \dots, L[i-1]$.size do
 6:         $s_1, \dots, s_{j-1} \leftarrow L[i-1][j]$
 7:         for $s_j \in$ TOKENS do
 8:             $p \leftarrow p\left(s_1, \dots, s_j \middle| \{(x_i, y_i)\}_{i=1}^m\right)$
 9:             add the tuple $(p, (s_1, \dots, s_j))$ to $S$
10:         end for
11:     end for
12:     pick $W$ sequences with highest probability from $S$ and append them to $L$
13: end for
14: $E \leftarrow$ an expression in $L$ with the best MSE error for $\{(x_i, y_i)\}_{i=1}^m$
```

Chapter 5

Results

In this chapter we describe the results. We start with univariate experiments and compare our model to regular Genetic Programming. Both approaches are evaluated on synthetic data. We then describe our attempts at extending the framework of Neural Symbolic Regression to the multivariate feature setting.

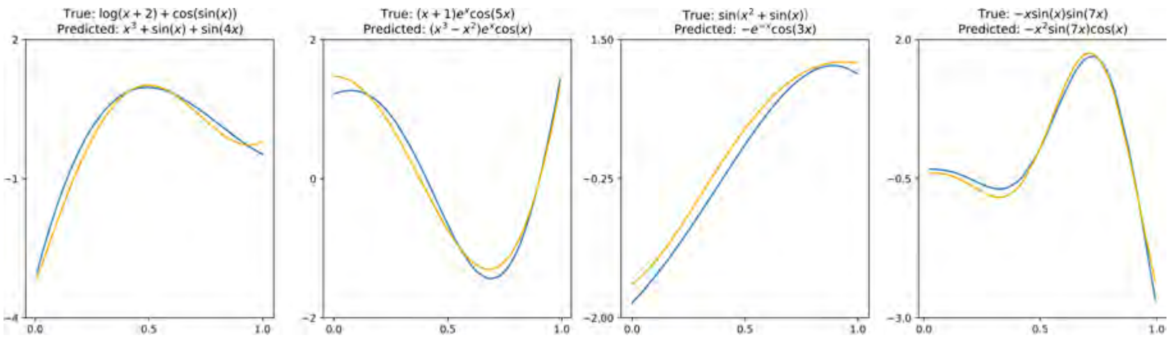
5.1 Details

We start by training our model. We chose the hyperparameters arbitrarily without carefully adjusting them. All multilayer perceptrons use ReLu activation functions except for the last layer. MLP_p and MLP_r both contain of 4 layers, and MLP_d has 3. Each layer has 256 hidden units, except for the input layer of MLP_p, which has 2, and the output layer of MLP_d, which has 21. The LSTM module contains two layers with 256 dimensional hidden state and 21 dimensional inputs. We use Adam optimizer with an initial learning rate of 0.001 and batches of size 16. We trained the architecture for 15 epochs. On a laptop this takes a few hours. During test time, while decoding we use a beam width of 10 and maximum depth of 15.

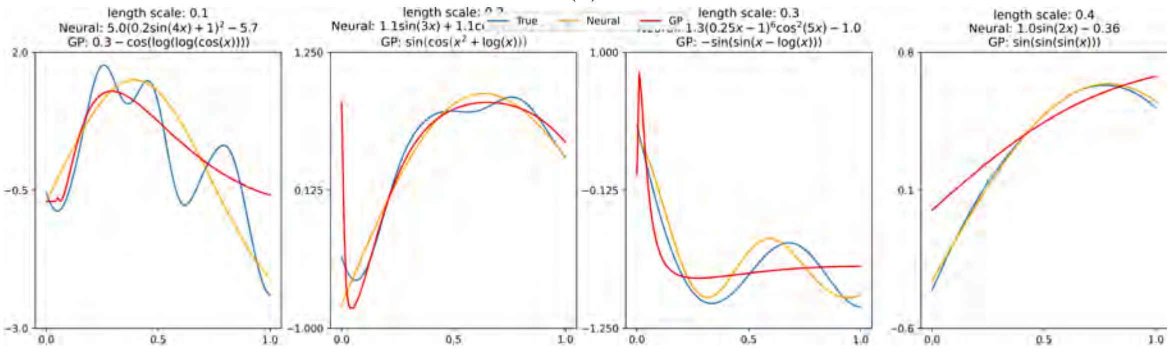
5.2 Univariate experiments

Figure 5.1a contains expressions that the trained model predicted on the validation part of our training set. Note that although the recovered expressions are not identical, some of them do share the same terms or factors. However, that's not that important - multiple expressions are functionally identical. What's intriguing is that our model is accurate. It manages to extract enough information from the dataset to be able to construct rather complicated expressions with high prediction accuracy.

To see how well it generalizes we evaluate our model on Gaussian Process samples. We use the most standard Gaussian Process with square exponential kernel, where the variance is set to 1 and length scale is varied between 0.1 and 0.5. Our generated expressions and



(a)



(b)

length scale	Neural		GP	
	MSE	Time	MSE	TIME
0.1	0.21(±0.15)	0.33(±0.1)	0.19(±0.06)	78.54(±5.85)
0.2	0.03(±0.03)	0.33(±0.04)	0.08(±0.07)	84.62(±6.67)
0.3	0.005(±0.005)	0.35(±0.04)	0.04(±0.03)	77.73(±5.78)
0.4	0.002(±0.003)	0.32(±0.04)	0.014(±0.012)	93.19(±17.5)
0.5	0.0013(±0.0016)	0.42(±0.12)	0.016(±0.018)	108.17(±9.59)

(c)

Fig. 5.1 Predicted expressions and their evaluations on a) validation data of the training set b) Gaussian Process samples. Note that the graphs are obtained by sampling from the expressions and normalizing as described in section 4.2. c) Performance comparison between our Neural model and a Genetic Programming algorithm on functions sampled from a Gaussian prior.

predictions are shown in figure 5.1b. Table 5.1c shows the MSE of our model on different length scales. Unsurprisingly, the model performs better on larger length scales.

5.2.1 Comparison to Genetic Programming

We compared our univariate approach to symbolic regression via standard Genetic Programming. We used an off the shelf implementation provided in the python package `gplearn` and, as before, evaluated how well the two approaches fit the data sampled from a Gaussian prior with varying length scales. The Genetic Algorithm was provided with the same set of operators as was our model trained on (refer to section 4.2). Both methods were given datasets of size 100. Figure 5.1b provides a visual comparison of the fitted curves, while table 5.1c gives time complexity and accuracy details.

Note 3 important details:

1. Although both approaches are similarly inaccurate for the length scale of 0.1, NSR is much more accurate than Genetic Programming on the bigger length scales.
2. NSR consumes **orders of magnitude** less time compared to Genetic Programming.
3. Our model produces smooth curves. In contrast, GP often outputs curves that are spiky. This is because of the way the algorithm deals with function domains, e.g. in `gplearn` the logarithm with a negative argument evaluates to 0, the division operator outputs 1.0 for very small denominators.

It must be mentioned that the hyper parameters for the GP were not optimized, however, that does require experience and it is not clear what hyper parameters should be chosen to produce the best comparison. However, this just highlights one of the advantages of NSR - once the model is trained, there are no hyper parameters to optimize.

5.2.2 Training with noise

Although NSR produces accurate expressions, it can get quite unstable if the data is noisy. This makes it prone to errors on real datasets. To counter that, we train a separate model where the training data is subject to $\mathcal{N}(0,1)$ noise. An example of it's robustness compared to the originally trained network is provided in the figure ??.

As discussed below, we will use our univariate model to fit residuals in an additive model. For this reason, fitting the network just on the Gaussian noise is not completely sufficient – residuals are not necessarily distributed according to the Gaussian distribution, especially if the underlying datapoints don't have any noise. The question of how to train our model such that it is robust to various noise distributions is crucial. However, we leave that for future work.

5.3 Multivariate experiments

5.3.1 Additive models

Our first step in trying to tackle multivariate symbolic regression is to consider additive models:

$$y(\mathbf{x}) = \sum_{i=1}^{d} f_i(x_i) \tag{5.1}$$

Since we already have a model that fits univariate functions, we can utilize it here. However, note that we can't use gradient descent. Instead, we employ a simple version of backfitting. We start with all f_i being zero. Every iteration, we consider updating each of the f_is but choose the one which reduces the mean squared error the most. To update f_i we compute the residues:

$$r(\mathbf{x}) = y(\mathbf{x}) - \sum_{\substack{j=1 \\ j\neq i}}^{d} f_j(x_j)$$

and use NSR to predict the symbolic expression for the dataset $\{(x_i^{(n)}, r(\mathbf{x}^{(n)})\}_{n=1}^{N}$. We qualitatively evaluate this approach on the following synthetic data:

- Samples from the validation data of our training set: we sample expressions f, g and evaluate the additive model on $h(x, y) = f(x) + g(y)$.
- Additive samples from GP: like above, but we sample f, g from a Gaussian prior.
- Samples from GP: sampling h from a 2D Gaussian process.

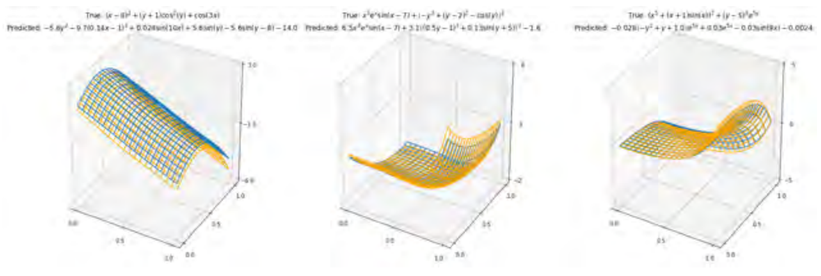
The results are given in figure 5.2. As expected, on the first two cases, our model does fairly well, but functions taken from a Gaussian prior where the two dimensions are not independent causes some trouble because of our modelling assumptions in equation 5.1.

Swiss Jura dataset

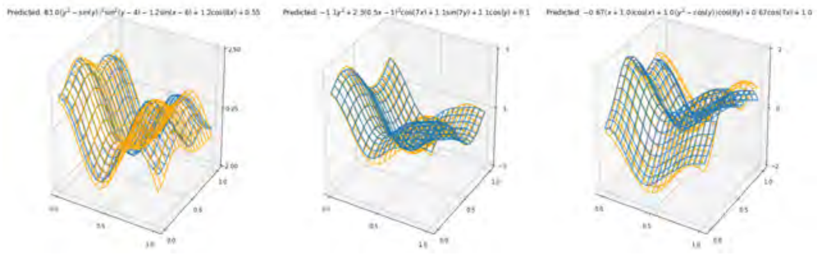
The Swiss Jura dataset contains a few hundred soil samples of several pollutants from around the Swiss region of Jura Mountains. The training set (259 points) contains concentration measurements of Nickel, Zinc and Cadmium, but the test set (100 points) is missing information about Cadmium. The goal in this dataset is to predict the concentration of Cadmium given it's spatial coordinates and Nickel and Zinc concentration. Often this dataset is used to evaluate new models in Gaussian Process literature. Although, I haven't evaluated any benchmark algorithms myself, the scores are reported in terms of mean absolute error (MAE). Standard GP methods achieve MAE of around 0.55, while advanced ones score around 0.4.

We evaluated our additive model on this data and achieved an MAE of 0.47 on both the training and test data. However, the equation that the model recovers depends only on the concentration of Nickel c:

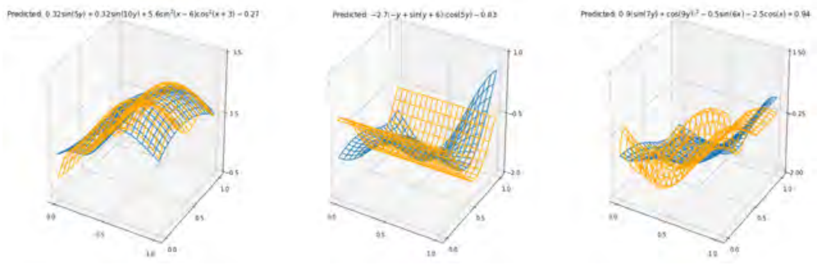
$$0.19 \left(c^3 + e^{e^c \sin{(c)}}\right)^2 \sin^2{(3c)} + 0.42$$



(a)



(b)



(c)

Fig. 5.2 Additive model evaluation on the three cases described in 5.3.1

5.3.2 Functions with two variables

Another obvious way to extend our model to the multivariate setting is to train NSR with symbolic expressions over multiple variables. Note that there is one issue: we need to generate a new trainng set, which would contain symbolic expressions over several variables. Even though we can use the same algorithm as described in section 4.2, we are now subjected to the curse of dimensionality. It was already a challenge to generate symbolic expressions that are versatile and curvy, but now it is much more difficult: each rotation of a linear function gives us a new function. In contrast, in the 1 dimensional case, because of scaling, we only had 2 linear functions x and $-x$. Another downside of this approach is that it is limited to regression on a fixed number of features.

We try this approach with 2 dimensional features. We generate a dataset containing 22000 expressions, 2000 of which were used for validation. Example predictions on those

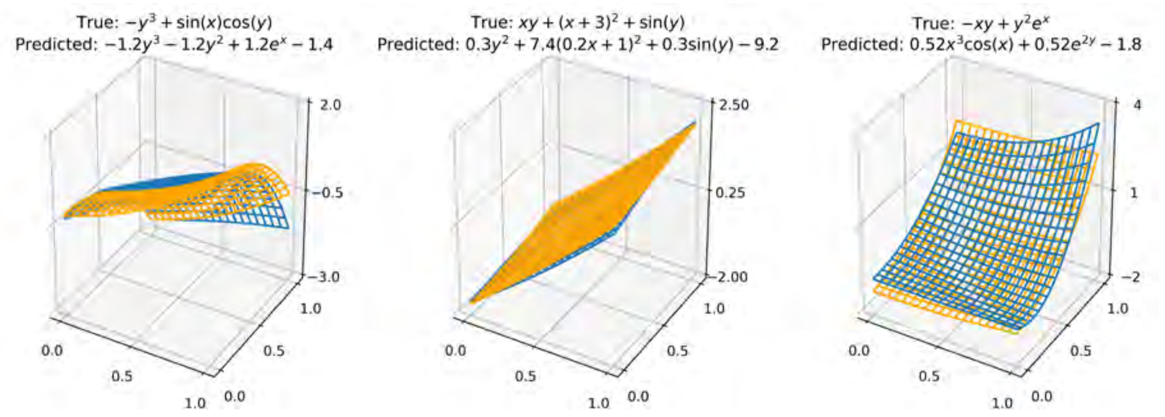


Fig. 5.3 Predicted expressions and their evaluations on the validation part of the training set with two variables.

expressions are provided in figure 5.3. Although our model still performs well and generates accurate expressions, the functions in the dataset are rather flat.

We also evaluated this approach on the Jura Swiss dataset, however since we could only use 2 variables we tried all pairs of features and picked the best one – Nickel concentration c_n and Zinc concentration c_z. We obtained the following expression for the concentration of Cadmium c_c:

$$c_c = -0.98c_n^3 - 0.98c_n + 0.98c_z + 0.98\left(c_n + c_z\right)^2 + 0.41$$

which scored an MAE of 0.42 on the test set and 0.49 on the training set.

5.3.3 Additive model with interactions

Having trained a seperate model for bivariate symbolic regression, we can now extend our additive model to also admit interactions between terms:

$$y(\mathbf{x}) = \sum_{i=1}^{d} f_i(x_i) + \sum_{i,j} g_{i,j}(x_i, x_j) \tag{5.2}$$

Note that 5.1 and 5.2 look like a GAM and a GA2M model, respectively. The only missing part is the link function, i.e. GA2M is specified as:

$$h^{-1}(y(\mathbf{x})) = \sum_{i=1}^{d} f_i(x_i) + \sum_{i,j} g_{i,j}(x_i, x_j)$$

However, we don't have a way to generate invertable functions h, and therefore, even if we would fit this model with NSR, it would not be possible to make predictions from it.

Unfortunately, this model hasn't provided us with better results on real datasets. We believe this is due to the limiting nature of the additive models and the fact that our dataset contains overly simplistic functions.

Chapter 6

Overview

6.1 Future work

Our approach to symbolic modelling is original and works extremely well on univariate data. However, there's still a long way to go to make the Neural Symbolic Regression method robust and practical. These are the possible directions for future work:

- Discovering ways to perform multivariate symbolic regression. Currently, none of our methods perform well on any real datasets. Making this model work on at least a few would give it some credibility and hope.
- In the current framework, the only way we incorporate the selection of simple expressions is when generating our training set. In contrast, Genetive Programming is quite flexible and there are multiple ways to force it to produce parsimonious expressions. Although, in the univariate case we haven't observed any problems with this, it might be a limiting factor, especially with more variables.
- A possible immediate project could be to investigate other architectures for the encoder and decoder in NSR, for example by incorporating attention the way it is in Attentive Neural Processes Kim et al. (2019), or decoding with trees, the way it is done in Doubly Recurrent Neural Networks Alvarez-Melis and Jaakkola (2016).
- Our proposed loss function for the end-to-end gradient descent training is based on "syntactic" cross-entropy loss. A loss function based on, for example, the MSE on the data would be much more significant and allow many more possibilities for architectural design. A possible way to achieve this is to train the network with a non-differentiable loss function via Reinforcement Learning.
- Our approach is not suited for classification problems, because categorical labels don't correspond to smooth functions. In order for this method to be useful, we have to solve this problem.

- Currently, our model contains only a few operators. If other operators were to be used, we would have to generate a new dataset and train the model from scratch. It would be an interesting project to see if it is possible to train such an architecture with a large number of operators, at the same time letting the user submit a binary vector indicating which of them they expect to observe.

Appart from symbolic regression, a few other thoughts for research we had are:

- Trying this architecture with other interpretable models, like rule lists or decision trees. Both of these structures already have an advantage - they can deal with categorical data. Maybe learning representations would also appear to be easier.

- Trying to adapt such structural learning to problems other than interpretability. For example, we view our approach as a strict improvement over GrammarVAE. One application of GrammarVAE is trying to find molecules that satisfy certain properties. We suspect that our approach could improve on the results in that area, because it is much quicker. Another potential application could be program generation.

6.2 Conclusion

In this project we focused on creating a neural architecture that is capable of performing symbolic regression. Our approach was unique – we learned representations of symbolic expressions, which contained not only syntactic information but also semantic information about the curvature of the expressions.

This has enabled us to do incredibly fast inference that was more accurate than a standard Genetic Programming approach. Admittedly, we haven't been able to find a satisfying method to do multivariate regression. However, this work serves only as proof of concept. We hope that it leads to quality research that makes learning interpretable models easier.

References

Ahmed M. Alaa, M. v. d. S. (2019). Demystifying black-box models with symbolic metamodels. In *Neural Information Processing Systems*.

Alvarez-Melis, D. and Jaakkola, T. S. (2016). Tree-structured decoding with doubly-recurrent neural networks.

Arnaldo, I., O'Reilly, U.-M., and Veeramachaneni, K. (2015). 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. Association for Computing Machinery.

Carton, S., Mei, Q., and Resnick, P. (2020). Feature-based explanations don't help people detect misclassifications of online toxicity. *Proceedings of the International AAAI Conference on Web and Social Media*, 14(1):95–106.

Caruana, R., Lou, Y., Gehrke, J., Koch, P., Sturm, M., and Elhadad, N. (2015). Intelligible models for healthcare: Predicting pneumonia risk and hospital 30-day readmission. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 1721–1730, New York, NY, USA. Association for Computing Machinery.

Carvalho, D., Pereira, E., and Cardoso, J. (2019). Machine learning interpretability: A survey on methods and metrics. *Electronics*, 8:832.

Diaconis, P. (1988). Bayesian numerical analysis. *Statistical decision theory and related topics IV*, 1:163–175.

Doshi-Velez, F. and Kim, B. (2017). Towards a rigorous science of interpretable machine learning.

Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232.

Garnelo, M., Rosenbaum, D., Maddison, C. J., Ramalho, T., Saxton, D., Shanahan, M., Teh, Y. W., Rezende, D. J., and Eslami, S. M. A. (2018a). Conditional neural processes.

Garnelo, M., Schwarz, J., Rosenbaum, D., Viola, F., Rezende, D. J., Eslami, S. M. A., and Teh, Y. W. (2018b). Neural processes.

Gilpin, L. H., Bau, D., Yuan, B. Z., Bajwa, A., Specter, M., and Kagal, L. (2018). Explaining explanations: An overview of interpretability of machine learning.

Goldstein, A., Kapelner, A., Bleich, J., and Pitkin, E. (2013). Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation.

Gordon, J., Bruinsma, W. P., Foong, A. Y. K., Requeima, J., Dubois, Y., and Turner, R. E. (2019). Convolutional conditional neural processes.

Hardt, M., Price, E., and Srebro, N. (2016). Equality of opportunity in supervised learning.

Kim, B., Wattenberg, M., Gilmer, J., Cai, C., Wexler, J., Viegas, F., and Sayres, R. (2017). Interpretability beyond feature attribution: Quantitative testing with concept activation vectors (tcav).

Kim, H., Mnih, A., Schwarz, J., Garnelo, M., Eslami, S. M. A., Rosenbaum, D., Vinyals, O., and Teh, Y. W. (2019). Attentive neural processes. *CoRR*, abs/1901.05761.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA.

Kusner, M. J., Paige, B., and Hernández-Lobato, J. M. (2017). Grammar variational autoencoder. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1945–1954, International Convention Centre, Sydney, Australia. PMLR.

Lai, V. and Tan, C. (2019). On human predictions with explanations and predictions of machine learning models. *Proceedings of the Conference on Fairness, Accountability, and Transparency - FAT* '19.*

Lample, G. and Charton, F. (2019). Deep learning for symbolic mathematics.

Li, L., Fan, M., Singh, R., and Riley, P. (2019). Neural-guided symbolic regression with semantic prior. *CoRR*, abs/1901.07714.

Lipovetsky, S. and Conklin, M. (2001). Analysis of regression in game theory approach. *Applied Stochastic Models in Business and Industry*, 17(4):319–330.

Lipton, Z. C. (2016). The mythos of model interpretability.

Lundberg, S. and Lee, S.-I. (2017). A unified approach to interpreting model predictions.

Martius, G. and Lampert, C. H. (2016). Extrapolation and learning equations.

Molnar, C. (2019). *Interpretable Machine Learning.* https://christophm.github.io/interpretable-ml-book/.

Pekkala, T., Hall, A., Lötjönen, J., Mattila, J., Soininen, H., Ngandu, T., Laatikainen, T., Kivipelto, M., and Solomon, A. (2017). Development of a late-life dementia prediction index with supervised machine learning in the population-based caide study. *Journal of Alzheimer's disease : JAD*, 55(3):1055–1067. 27802228[pmid].

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "why should i trust you?": Explaining the predictions of any classifier.

Roach, K. (1996). Hypergeometric function representations. In *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation*, ISSAC '96, page 301–308, New York, NY, USA. Association for Computing Machinery.

Rudin, C. (2019). Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1(5):206–215.

Sahoo, S. S., Lampert, C. H., and Martius, G. (2018). Learning equations for extrapolation and control.

Searson, D. P. (2014). Gptips 2: an open-source software platform for symbolic data mining.

Shrikumar, A., Greenside, P., and Kundaje, A. (2017). Learning important features through propagating activation differences.

Simonyan, K., Vedaldi, A., and Zisserman, A. (2013). Deep inside convolutional networks: Visualising image classification models and saliency maps.

Sundararajan, M., Taly, A., and Yan, Q. (2017). Axiomatic attribution for deep networks.

Tan, S., Caruana, R., Hooker, G., and Lou, Y. (2018). Distill-and-compare. *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society.*

Vladislavleva, E. K., Smits, G., and den Hertog, D. (2009). Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *Evolutionary Computation, IEEE Transactions on*, 13:333 – 349.

Weerts, H. J. P., van Ipenburg, W., and Pechenizkiy, M. (2019). A human-grounded evaluation of shap for alert processing.

Weller, A. (2017). Transparency: Motivations and challenges.

Weng, S. F., Reps, J., Kai, J., Garibaldi, J. M., and Qureshi, N. (2017). Can machine-learning improve cardiovascular risk prediction using routine clinical data? *PLOS ONE*, 12(4):1–14.

Zeiler, M. D. and Fergus, R. (2013). Visualizing and understanding convolutional networks.

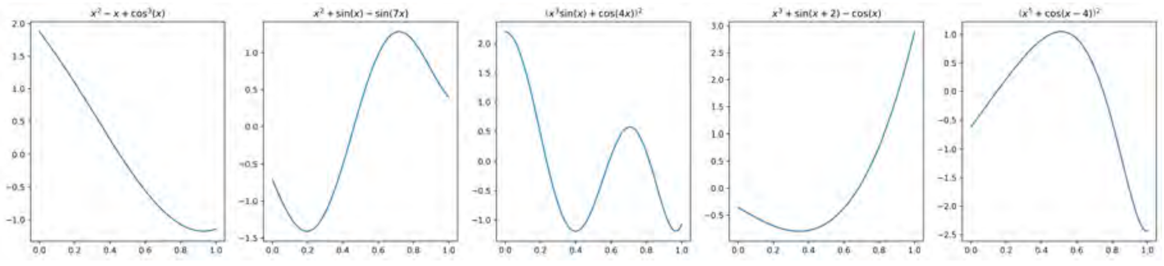
Zliobaite, I. (2015). On the relation between accuracy and fairness in binary classification.

Appendix A

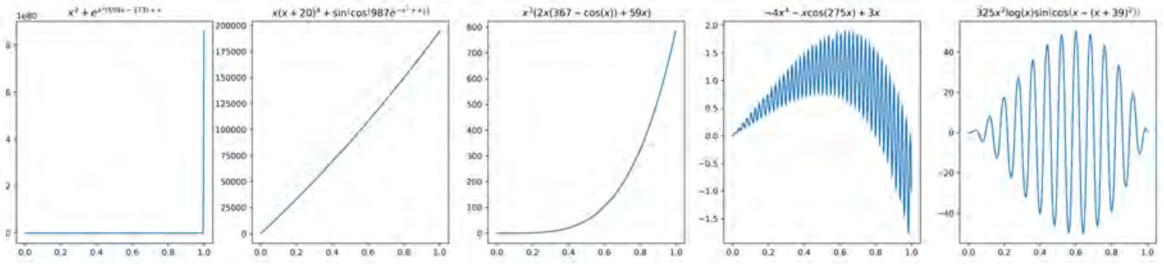
Data generation

Here we describe our data generation process. As mentioned, it is an ad-hoc method that tries to overcome the issues listed in section 4.2.

The approach is to grow the dataset starting with simple expressions: `x, 1, 2, ..., 10`. Each step we try to apply operators on already known trees and see if that produces a function that is different from the ones we already have. To compare the functions, we first normalize them to have zero mean and unit variance on the interval $[0, 1]$ and then measure their similarity using the supremum norm. To make sure that the produced expressions are well-defined we track their values. Any expressions that produce numerical errors are discarded.



(a)



(b)

Fig. A.1 Graphs and expressions produced by a) sampling from our dataset b) sampling from the algorithm in Lample and Charton (2019)

The pseudocode for the algorithm is provided in algorithm 2 on the next page. $\hat{f}$ there stands for a normalized version of the function f. The final dataset contains 60k expressions, 55k of which are used for training and the rest for validation. A few samples are provided in the figure below comparing them to the ones produced by the approach in Lample and Charton (2019). Notice not only at the curvature but also at the simplicity of the expressions that we generate.

Algorithm 2 Dataset generation

```
    Input: $\epsilon, N$, a set of unary and binary operators $U, B$, a set of constants $C$
    Output: Dataset $D$ containing symbolic expressions
 1: initialize $D \leftarrow [x, 1]$
 2: initialize $S \leftarrow [x, 1, \ldots, 10]$
 3: for $i = 1, \ldots, N$ do
 4:     $S' \leftarrow \emptyset$
 5:     for $\mathrm{op}_1 \in U$ do
 6:         for $g \in S$ do
 7:             $f(x) \leftarrow \mathrm{op}_1(g(x))$
 8:             $S' \leftarrow S' \cup \{f\}$
 9:             if $\|\hat{f} - \hat{h}\|_{\sup} \geq \epsilon \ \forall h \in D$ then
10:                 $D \leftarrow D \cup \{f\}$
11:             end if
12:         end for
13:     end for
14:     for $\mathrm{op}_2 \in B$ do
15:         for $g_1 \in S$ do
16:             for $g_2 \in D \cup C$ do
17:                 $f(x) \leftarrow \mathrm{op}_2(g_1(x), g_2(x))$
18:                 $S' \leftarrow S' \cup \{f\}$
19:                 if $\|\hat{f} - \hat{h}\|_{\sup} \geq \epsilon \ \forall h \in D$ then
20:                     $D \leftarrow D \cup \{f\}$
21:                 end if
22:             end for
23:         end for
24:     end for
25:     $S \leftarrow S'$
26: end for
27: return $D$
```