CS 221 Final Project
Accelerating Symbolic Regression with Deep Learning

Tyler Hughes, Siddharth Buddhiraju and Rituraj

December 16, 2017

Contents

1 Introduction

During the course of our PhD research, we have wondered if the data generated by numerical simulations or physical experiments resulted from a simple, closed form function of the input. While curve fitting for a parametrized, user-defined model is accomplished by programs such as MATLAB, they are not capable of discovering the models themselves. Therefore, our project aims to efficiently generate simple symbolic expressions that describe the given data. Explicitly, if our input is data set of $(x, y = f(x))$ pairs where the function $f(x)$ is unknown, our output is given by a function $\hat{f}(x)$ that best describes this dataset, along with the optimal values of the corresponding constants in the model. For example, the input might be

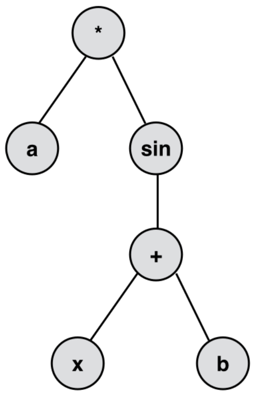

Figure 1: Tree representation of the equation: $f(x) = a * \sin(x + b)$. This tree has a depth of 4 in our model.

$\{(0,1),(1,2)\}$, and our program must generate an output of $\hat{f}(x) = x + c$, $c = 1$ with error $= 0$.

The problem of fitting symbolic expressions to data, also known as 'symbolic regression', has been discussed by several authors [1, 2, 3] for generating expressions that model the data as well as differential equations whose solution is the data. Typically, genetic algorithms with pareto optimality have been used to frame symbolic regression as an optimization problem [4]. More recently, Ref. [5] modified the symbolic regression problem to generate physical inference such as conservation laws or invariants.

However, these methods do not incorporate learning from examples, and consequently, can require a large computation time to discover good symbolic fits to the data. Furthermore, they do not use any features present in the original dataset to aid in the optimization. In our project, we incorporate learning by 1) training a neural network to extract features from the data, and 2) using a recurrent neural network (RNN) decoder to predict symbolic expressions from this encoding. This strategy is able to learn to leverage features from the data in order to find a well-fitting model function without resorting to brute-force searches.

To motivate our approach, we note that in problems involving sequence data, such as machine translation and speech recognition, RNNs implemented with the Long Short Term Memory (LSTM) model have had enormous success [6, 7, 8]. As opposed to a standard RNN, the LSTM is able to retain important state information over several cells, which makes them more suitable for problems where there may not be a direct relationship between adjacent members of a sequence, as in our problem. Thus, in analogy to machine translation problems, where one wishes to encode a sequence of words from one language and decode it into another, we employ LSTMs to decode the encoding of the data points that is provided after fitting. In our approach, we feed to each copy of the LSTM the next symbol in our function expression, with the hope that the LSTM learns to uniquely map a feature vector encoded from the data points (x, y) into the appropriate function up to the constants. Subsequently, the constants are fit using traditional regression.

2 Dataset

Each entry 'i' of our dataset is a generated set of (x, y) pairs corresponding to a function $f_i(x)$. We create $f_i(x)$ randomly from an allowed set A consisting of operators, functions and constants. For example, we may choose this set to be $A = \{(\cdot)+(\cdot), (\cdot)*(\cdot), \sin(\cdot), \cos(\cdot), \mathrm{x}, \mathrm{c}\}$. Here x is the independent variable and c is a constant, which is randomly assigned a value later on.

We first note that any simple mathematical expression can be represented compactly in a tree form. For example, the tree corresponding to $f(x) = a * \sin(x + b)$ is shown in Figure 1. In this representation, operators are nodes with two children (left and right operands), functions (of a single variable) have one

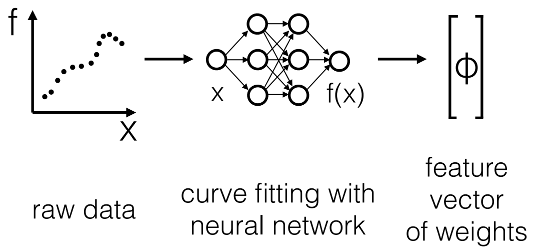

Figure 2: Encoding process. A neural network fits a function $f(x)$ to the raw (x, y) points. The weights and biases in this neural network are flattened into a vector which is to be decoded.

child, and constants/variables are leaf nodes. Thus, we generate $f_i(x)$ by first selecting the class of the root node randomly from $\{\text{operator}, \text{function}, \text{constant}, \text{variable}\}$. Then, we select an element uniformly at random from this class and insert into the tree. Depending on the class, we create 0, 1, or 2 children for this node and repeat this process iteratively. If the tree has reached a certain maximum depth, which we set beforehand, then we force the nodes at this depth to be either constants or variables.

After the equation tree is constructed, we set all of the constants to random values sampled from some range. Then, to get the (x, y) pairs, we sample x uniformly at random in a specified range and evaluate the tree from the leaf nodes up to get $y = f_i(x)$, which we add to the set. We repeat this process several times and (optionally) add some uniformly distributed noise to the y values in order to ensure that each training example is unique. For the discussion in this report, the constants are set to be between -5 and 5, the range of x values between -1 and 1, and our trees to a maximum depth of 4. We chose 50 (x, y) data points per training example.

3 Neural Network Encoder

In order to encode the input data consisting of (x, y) pairs, we fit the data with a neural network as shown in Figure 2. As the neural network trains over the single input data set, its weights and biases are optimized to resolve the different features in the input functions. For example, a straight line of the form $ax + b$ and a wavy function containing $\sin(\cdot)$ would lead to substantially different neural network parameters. In other words, we expect that the parameters learned by the neural network are a reasonably unique indicator (up to the constants) of the underlying mathematical expression and contain encoded information about the characteristics of the initial dataset. Once the function is fit by the neural network, we flatten all of its weights and biases into a vector to be decoded.

4 LSTM Decoder

With the input datasets now encoded by the neural network, we wish to decode these encodings into mathematical expressions describing the dataset. As discussed earlier, we train LSTM networks to decode our expressions. The equation elements that are returned are represented as probability distributions pertaining to one-hot vector encodings of the equation elements. When the function expression consists of only the variables and functions of one variable, our problem is exactly analogous to machine translation. However, with binary operators such as '$+$' and '$\times$', a branching factor is introduced into the problem. We propose two solutions to deal with this problem: in the first, we ignore the branching factor to retain the sequential nature of the LSTM, and in the second, we propose a tree-like LSTM structure. Both LSTM decoders were built in Tensorflow [9].

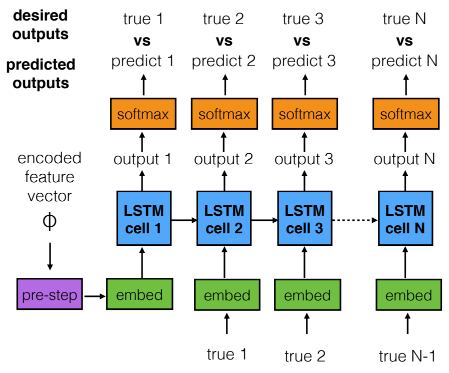

Figure 3: Sequential LSTM model used to decode equations. The encoded feature vector, ϕ is sent to the first LSTM cell (blue) through a dense layer (purple) and then an embedding layer (green) of matrix multiplications and bias additions. An output state is returned and sent through a softmax layer (orange) in order to predict a one-hot for the first equation element. The initial hidden state for this first LSTM cell is a vector of all 0's. The hidden state is sent to the next LSTM cell and the process is repeated with the input being the true one-hot corresponding to the previous stage's equation element. The loss function is evaluated over the predicted one-hots and the true one-hots for each time step.

4.1 Sequential LSTM

In the sequential LSTM model, we retain the sequential nature of machine translation LSTM problem, as shown in Fig. 3. That is, the input to our LSTM is the exact string of symbols used to represent the function. For example, the LSTM input corresponding to $\sin(x + c)$ is the array of one-hot vectors corresponding to [sin, (, x, $+$, c,)]. Therefore, the open- and closed-parentheses and the binary operators are a part of the building block one-hots used to generate the output functions. Consequently, in order to generate meaningful mathematical expressions, the LSTM must also learn the syntax associated with the parentheses and binary operators, i.e., that parentheses occur in pairs and that binary operators must be preceded and succeeded by variables, functions or open-parentheses. However, because the model is sequential, it is faster than a model that introduces branching into the LSTM.

The benefit of speed comes with some trade-offs. Since the sequential model must also learn the syntax, it is expected to occasionally produce expressions that are syntactically incorrect and therefore cannot be evaluated. Further, the space of the building blocks is increased by two due to the parentheses, possibly leading to extra unnecessary parentheses in the expressions without affecting their correctness. In order to overcome these problems, the second model we propose is a tree LSTM that accommodates branching due to binary operators by default.

4.2 Tree LSTM

As we discussed in the section describing the dataset generation, the simple mathematical expressions we are working with can be compactly represented as trees. In this form, there is no need for dedicated syntactical helpers, such as the parentheses, because the mathematical structure and order of operations is implicit in the ordering of the tree. Thus, it should be possible to build our decoder as a system that can return mathematical trees of the same form as our dataset.

To do this, we built an LSTM model that sends its output to two separate LSTMs, as shown in Figure 4. We send the hidden states of the LSTM cells directly to each of the child cells, but the output states must go through further processing to differentiate between left and right children. To do this, we have output states go through a single dense layer before being fed into their corresponding child LSTM inputs. Similar to the sequence model, the parameters in each LSTM cell, embedding cell, and output cell are shared.

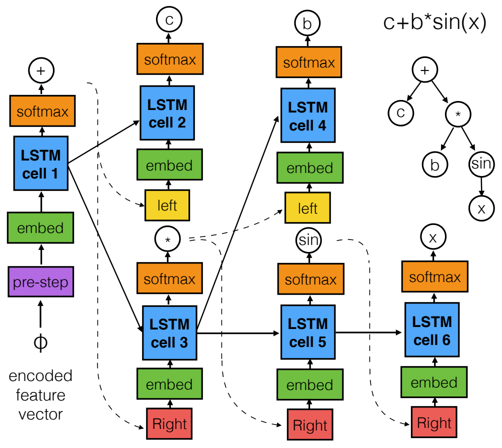

Figure 4: Tree LSTM model used to decode equation trees. As in the sequence model from Fig. 3, we initially input our feature vector, ϕ to the first LSTM cell in the same manner. The predicted one-hot is returned from the softmax layer and evaluated for equation class (operator, function, constant/variable). Depending on the result, we create 2, 1 or 0 child LSTM cells, respectively, with the same hidden state as the previous cell. The output is sent through a different dense layer depending on if it is headed to a left child (yellow) or right child (red). This allows the model to differentiate between equations that have two possible orderings, such as x+c and c+x. The hope is that an implicit ordering is established. This process is repeated until all child nodes are of the class constant or variable, or a maximum allowed depth is reached.

While this model always returns syntactically correct trees, there are complications in implementing it in Tensorflow because the setup of such a tree takes a sizable amount of time. Therefore, if we need to construct a new tree for each training example, this method is prohibitively slow. To get around this issue, we have our model predict full binary trees for each decoding step, with special '<eoe>' markers in the nodes that are empty. Then, we later prune the tree of these nodes. This allows us to reuse the same binary tree structure for all of our training examples.

5 Results

In Table 1, we show the prediction accuracy of both our sequential LSTM decoder and our tree LSTM decoder over a data set of 1500 examples, i.e., 1500 mathematical functions each with fifty (x, y) points. Depth 1 functions such as $y = x$ and $y = c$ were excluded from the data set since the decoder predicted them correctly with 100% accuracy.

For a depth of 2, the sequential model is almost perfect since the class of unique functions that can be generated is rather small, and learning the syntax on this small class is rather easy. However, as we reach a depth of 3 and 4, the accuracy of both models drops to around 50% or below. It is worth noting that mathematical expressions at this depth may become incredibly complicated, and almost impossible to write down based on just the data plot - take $\tanh(\sin(x) + \tanh(x))$, for example. Therefore, our accuracy is still quite high when compared to a human baseline. In Figure 5, we show the change in the training loss divided by the number of examples as the LSTM was trained.

In the present form, it appears that the sequential model performs better than the tree model. For example, for depth 2, the sequential model has a training accuracy of 93.4% while the tree model achieves only 69%. However, as mentioned, the tree model does not require special treatment to handle syntactical correctness. It is possible that the tree model requires many more epochs to converge than the sequential model, resulting in the poorer performance at 300 epochs. Another potential source of error in the tree

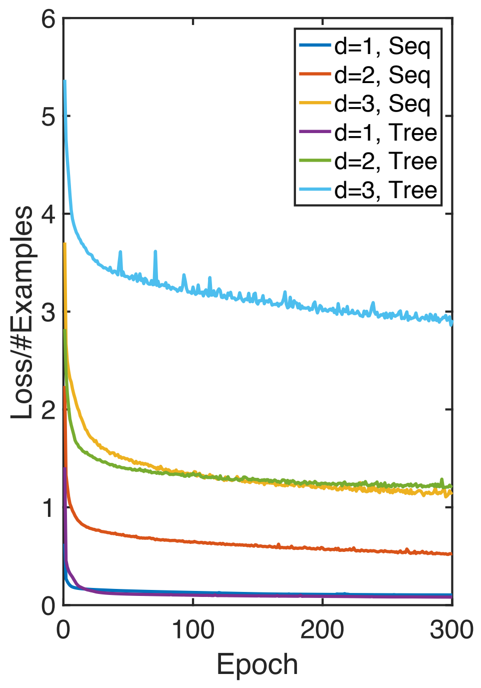

Figure 5: Training loss divided by the number of examples as the LSTM decoder was trained for the sequential as well as tree models.

	Sequence		Tree	
Depth	Train Accuracy	Test Accuracy	Train Accuracy	Test Accuracy
2	93.4%	98.7%	69%	78%
3	56.0%	56.0%	33%	32%
4	27.7%	28.0%	22%	21%

Table 1: Accuracy on both the sequential and tree LSTM model for 1425 training examples and 75 test examples after **300 epochs** for $\mathbf{L^2}$ loss. Depth 1 was excluded since both models predicted the correct functions, viz. $y = x$ and $y = c$, with 100% accuracy.

	Sequence LSTM			
Depth	Train Accuracy		Test Accuracy	
	cross-ent	L^2	cross-ent	L^2
2	91.9%	92.8%	92.0%	92.0%
3	45.2%	45.2%	38.7%	37.3%
4	12.8%	17.7%	10.7%	14.7%

Table 2: Comparison of accuracy on the sequential LSTM model for 1425 training examples and 75 test examples after 150 epochs between the **cross entropy with logits** loss and $\mathbf{L^2}$ loss.

model may be the connection between parent LSTM to left and right child nodes. Right now, the outputs go through separate single dense layers before feeding to left and right child nodes, but perhaps more complexity is needed to allow the model to gain better accuracy.

Because our test and train accuracies are relatively similar to each other, we anticipate that our model is doing a decent job generalizing to new examples. The low accuracy result may be an indicator of low bias in our model, which suggests that adding more complexity to our model could be desirable. Furthermore, especially at larger max depths, there are more than one reasonable solutions to the curve fitting problem. For example, a function $\sin(x)$ could be reasonably mistaken for x in some small range. These results only check for equation correctness, but do not incorporate fitting error. These concerns will be elaborated on in the following section.

6 Analysis

6.1 Implementation and Hyperparameters

We now discuss some details of the methods we used to implement this model, including the hyperparameters chosen, and give some analysis explaining our final choices.

6.1.1 Encoder network

In our experiments, we found that a neural network with one hidden layer of size 4, worked well to fit the functions while not introducing too many free or unused parameters. We used $\tanh(\cdot)$ activation functions in the hidden layer and a linear activation for the final layer. To prevent over-fitting and also force the network to find unique solutions to the fitting problem, we added L2 regularization with a strength of 0.01. The training of this network was done with gradient descent using a learning rate of 0.1 and 3000 epochs. The neural network was implemented from scratch in Python.

6.1.2 LSTM network

To evaluate loss between the true and predicted one-hot vectors, we tried L1 loss, L2 loss, and softmax cross-entropy loss, finding that L2 worked the best followed by cross-entropy. In Table 2, we compare cross-entropy vs. L2 loss on the sequential model. L1 loss performed very poorly, with little reduction in training error over time when compared against L2. We used an Adam optimizer with learning rate of 0.001 and 1500 epochs. Dropout regularization was tried with drop rate of 0.2 and 0.5, but this was not found to improve performance.

Interestingly, the LSTM hidden state size was a crucial hyperparameter that dramatically affected the performance of our model. For very low values (1-10), our model was not able to learn enough from the training examples, and we had resulting poor performance. On the other hand, for large values (50-200), our training became erratic and it seemed as if the LSTMs may have had too many degrees of freedom. We settled on a state size between 20 and 40, as this middle ground seemed to give the best results overall.

6.2 Function Evaluation

6.2.1 Undetermined constants

Our model is able to correcly classify functions involving undetermined constants, such as $f(x) = \sin(x + c)$, even though each of these training examples were supplied with differently assigned values of c. This shows that the model learns to generalize to undetermined constants from specific examples, which signals a significant improvement to brute-force search methods, which have to search over possible constant assignments in addition to underlying functions.

6.2.2 Finding the constants

After the model is run, we wrote a script to determine the constant values in the predicted function using traditional regression. For this we define a squared loss integral, similar to the training error in linear regression, as

$$E(\mathbf{c}) = \int dx (f(x) - \hat{f}_{\mathbf{c}}(x))^2, \tag{1}$$

where $\mathbf{c}$ is a vector containing all the constants to be determined. Then, we start from an arbitrary initial guess for $\mathbf{c}$ and keep updating it using gradient descent $\mathbf{c} := \mathbf{c} - \eta \nabla_{\mathbf{c}} E(\mathbf{c})$. We start with a relatively large value of η and keep decreasing it as the error decreases. Although there are more efficient methods, gradient descent suffices for a small number of constants (true for low tree depths) and converges relatively fast.

6.2.3 Syntax and heuristics

Since the sequential LSTM model must also learn the syntax, i.e., the appropriate use of parentheses, binary operators and arguments to functions, it occasionally produced invalid expressions. In order to overcome this problem, we attempted to include heuristics that increase the loss when a nonsensical expression was generated by the LSTM. Suppose the first prediction from the LSTM was tanh, then the only acceptable succeeding output could be the open parenthesis '('. Similarly, for each LSTM prediction, a vector of 1's and 0's can encode the allowed succeeding prediction. Therefore, a bi-gram heuristic could be added to the loss of the form

$$\text{Loss} := \text{Loss} + \lambda \sum_{j=1:\text{len(equation)}} \text{pred}(j-1)^{\text{T}} \mathbb{C}\ \text{pred}(j) \tag{2}$$

where λ is a manually set prefactor, pred(j) is the one-hot vector of the $j-$th prediction, and the matrix $\mathbb{C}$ is 0 for disallowed successors, and 1 for allowed successors. While adding this heuristic function certainly produced syntactically correct expressions, the error between the desired equation and the predicted equation increased, possibly because the heuristic was bi-gram, i.e., only compared successive predictions. Furthermore, this heuristic was quite sensitive to the amplitude λ.

6.2.4 Training examples

Finally, it was eventually found that upon increasing the number of training examples from 200 to 1500, the LSTM learned the syntax substantially better, and therefore, the occurrence of invalid expressions decreased. Therefore, in the final form of our code, no heuristics have been included.

7 Future Work

While our model performed reasonably well in predicting equations, we anticipate there are several directions for future work that would be potential improvements.

7.1 Auto-encoder

In this iteration, we specify directly that we want our feature vector to be composed of the weights and biases in our neural network fit. The encoder and decoder are, therefore, trained separately. We believe that a more fruitful approach would involve training these two components at the same time. For example, one could imagine an encoding network consisting of a series of 1D convolutional neural networks (CNNs), where we send the outputs of the convolutions to the LSTM network and simultaneously train the kernels. An alternative approach could involve using an encoding RNN to take the (x,y) points directly and use the final hidden state as the encoded feature vector. We anticipate that our success was limited by the features that our networks could extract from the original dataset. Also, since not all parameters of the neural network were of the same importance in fitting the dataset, a lot of unnecessary data was included in the feature vectors.

7.2 Incorporating function fitting metrics into training

Another limitation of our model was the fact that we evaluated our training steps based on how closely our model predicted the equation compared to the true equation. In reality, what we more care about is how well the predicted function fits the original dataset. For example, one could imagine several functions that may fit the dataset equally well. In this case, it does not make sense to penalize the model, on training, if it does not guess the 'correct' function, when there are many acceptable answers. This functionality could, in principle, be implemented by performing a fit during train time and then returning a loss that depends on the error of the fit. However, it is not clear whether this operation is differentiable and, therefore, whether we may implement and optimize this model in Tensorflow or related packages. Therefore, we left curve fitting as a post-processing analysis step for the scope of this project.

7.3 Pareto-optimality and function simplicity metrics

As mentioned in a few previous studies involving symbolic regression [2, 4], one would ultimately want a system that returns equations that are both good approximations to the data and as simple as possible. This metric can be framed as a Pareto optimality condition [4, 10]. For example, we could have our model output a set of equations that are Pareto optimal, in so far as each member of the set has no counterpart that is both more simple and more accurate than it. The user could then choose which equation best suits his or her needs depending on the application. For instance, if the project expects a simple model to describe the data, then a simple equation can be selected, and vice versa. Alternatively, a parameter can be set that controls the relative importance of simplicity and accuracy. Then, the decoding step may use this parameter to guide it's predictions.

8 Conclusion

We have shown that symbolic regression can be framed as an encoder-decoder problem with reasonable success using LSTM recurrent neural networks. By leveraging the features and structure in our input data, we hope that this approach can be fruitful in advancing symbolic regression for many applications.

References

[1] Josh Bongard and Hod Lipson. Automated reverse engineering of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 104(24):9943–9948, 2007.

[2] John Duffy and Jim Engle-Warnick. Using symbolic regression to infer strategies from experimental data. In *Evolutionary computation in Economics and Finance*, pages 61–82. Springer, 2002.

[3] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[4] Wouter Minnebo, Sean Stijven, and Katya Vladislavleva. Empowering knowledge computing with variable selection, 2011.

[5] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009.

[6] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[7] Christopher Olah. Understanding lstm networks. *GITHUB blog, posted on August*, 27:2015, 2015.

[8] Andrej Karpathy. The Unreasonable Effectiveness of Recurrent Neural Networks. http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

[9] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[10] Ekaterina J Vladislavleva, Guido F Smits, and Dick Den Hertog. Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *IEEE Transactions on Evolutionary Computation*, 13(2):333–349, 2009.