SuperSynthesizer Checkpoint

Ariel Hoffman

November 8, 2015

1 Theory

The main objective behind this project was implementing an artificial intelligence that could effectively determine the best synthesizer to use for a given program. In order to do this, I needed to create an artificial intelligence, I needed to determine a good feature set, and I needed to provide it with enough data that it could work on at least preliminary results. Thus far, it has been successful. Below is a diagram of the system overview. I will discuss each of the major components in the rest of this section.

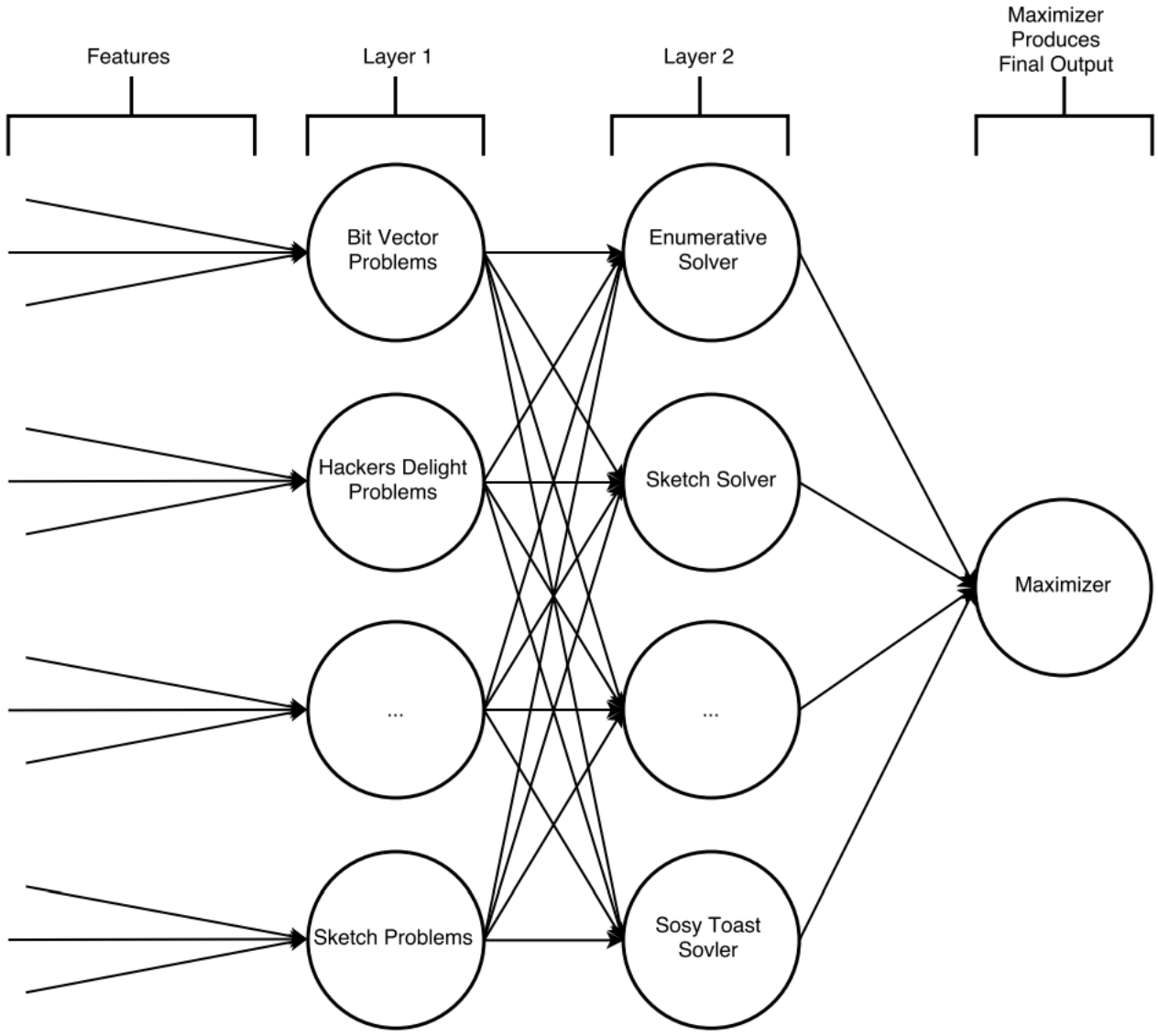

Figure 1: System Overview

1.1 Neural Network Layout

Initially, I had little idea of what things mattered in terms of the ultimate outcome and what things didn't. As a result, I had to rely heavily on work already done by the SyGuS hosts in sorting and evaluating the benchmarks. A sample of the benchmarks was available on github.[1] The benchmarks were, conveniently, presorted into categories that seemed to have impact on which solvers would best suit the solvers. With that knowledge in mind, I began to consider different ways I could make use of this work in the final product. Finally, I decided that I could design the neural network such that the first layer was initially going to simulate the first classifications that the SyGuS creators did for us. Then I decided that the second layer could represent the solver desirability metric. The major design decisions and justifications I made were as follows:

1. I decided that for the prototype I wanted a 2 layer network. A system with more than two layers would have the advantage of being able to classify more problems more accurately once it was fully trained. However one major problem associated with many layer systems is that, because I cannot control the initial conditions as closely, I need to be able to train the algorithm with many, many benchmarks. If I can get access to more benchmarks, I may consider experimenting with more layers, but for now, I wanted to see what would happen on a two layer network.

2. I decided that the initial layer neurons should be associated with the classifications made by the SyGuS moderators. In some neural networks, the first layer and feature design are obvious. For instance, in the case of image recognition, the a neuron in the first layer may just represent a section of the image. However unlike an image, a program relates to everything around it and can therefore not be split up as easily. I figured that because we know the classifications, that would be a good place to start this layer out. Before we move on, though, I just want to note that the layer 1 neurons only have a concrete meaning before the neural network is trained. I need an initial repository of features, weights, and biases to feed into the first layer. However, once the neural net begins its training, all bets are off: the nice concerted definitions of the neurons disappears.

1.2 Feature Design Considerations

The feature design was probably the most interesting part of this project. I initially had the impulse to create the features manually. (I.e. for bit vectors I would write something like - the number of occurances of and, or, xor, etc.) However, I quickly realized two things:

1. Manually determinig good features for each category would be a huge task for a single person. I would have to go through each program category and decide if there were any obvious patterns that I should take advantage of.

2. My manually determined features would naturally create unnecessary bias on the output. Though my manual efforts may do a really good job of making the layer 1 outputs represent the SyGuS classifications, that will not necessarily be the best way to help us achieve the best output. Once the system is trained, the output will depend not on the SyGuS outputs, but rather on whatever classifications the system itself thinks are most relevant to the ultimate outputs.

[1] https://github.com/rishabhs/sygus-comp14

Therefore, I resisted the urge to determine the features with brute force and I explored automated options. Ultimately, because this system is a lot like real writing, I decided that one way I could do it was to create a hash of the words in the system and make that the feature vector.

1.3 How a Neural Net Actually Works

A neural network is really just a cascaded, interconnected feedback system. Each neuron has input features which we'll call f, feature weights, which we'll call w, a bias value, t, and an output value, a. The neural network architecture that I'm using was originally designed in order to support backward propagation. Backward propagation is the process of taking an output error and recursively determining how much the weights and biases throughout the network should be modified in order to support that. As you can imagine, this is much easier to do with a smooth function than a step function. Therefore, I decided to use the sigmoid function to get my output which is consistent with what other people typically use. The sigmoid function is as follows:

$$\sigma(z) = \frac{1}{1 + e^{t-z}} \tag{1}$$

In our case, z is simply the sum of the features and the weights multiplied together for a given neuron.

$$z = \sum f_i w_i \tag{2}$$

And t is the bias we already mentioned. The result of this equation is merely a. This process makes backward propagation much easier because the partial derivatives are easier to find.[2] From a practical perspective, what this means is that before I set my neural net free on the data, I need to create initial values for the weights and biases. Furthermore, they need to be pretty good because, although I have many benchmarks at my disposal, I don't have an infinite number of them.

1.4 Getting Initial Values

I needed to get initial values for each of the feature weights and the biases for the first and second layers. For the first layer, I did this by iterating through all the programs in the different classifications and storing a dictionary of words for each classification. Each dictionary entry contained the number of occurrences of a given word for a given classification. To get the weights, I normalized every word entry. So, for instance, if a word appeared 7 times in a given classification, but 176 times total, I would normalize the entry to be 0.040. I would set that as the initial weight value for that feature. Thus, if a word that occurred two times had a weight of 0.040, the feature would contribute z += 0.040*2. I determined the initial values of the bias by running the layer 1 classification system and storing the values of the maximum z's (across all the layer 1 neurons) and divided that value by 2. I set that result as the bias.

Next, to determine the best initial values for layer 2, I used the data collected during the SyGuS competition and I assumed that the incoming signals were reasonable representations for the features. Then, for each solver, I iterated over the benchmarks in each category it had been run on and I calculated its performance with respect to the minimum performance for that benchmark (1 meaning that it had the minimum score and 0 meaning that it did not complete the run but some

[2] http://neuralnetworksanddeeplearning.com/chap2.html

other benchmark did.) Then I summed up those numbers and divided by the total number of included benchmarks. Using this method, I was able to find a solver performance for each benchmark. This is what I used as the weights for the incoming features. Mathematically it looked as follows:

$$\frac{\sum_i \frac{min(runtime)}{solverRuntime}}{i} = w \tag{3}$$

2 Results

As a preliminary result, I tested how many times layer 1 was accurately guessing about the category and I found the following results:

Category	Number of Programs	Percent Categorized Correctly
Bit Vector Benchmarks	7	74.1%
Let Benchmarks	25	72.0%
Hacker's Delight Benchmarks	57	77.2%
ICFP Benchmarks	50	100.0%
Multiple Functions Benchmarks	8	75.0%
Integer Benchmarks	16	100.0%
Sketch Benchmarks	8	0.0%
Overall	171	81.2%

The only category that really performed poorly was the sketch benchmark category. This is because that category is really a compilation of diverse programs that are grouped together for historic reasons. As a result, I figured that the output of this neuron might not be very indicative of what solvers are likely to work well. That said, for consistency's sake, I left it in because I wasn't sure if the neural network might find something that I hadn't thought of.
After I set up the initial values for the neural network, I finally ran the whole thing. I fed the values forward and, on the first try, the machine was assigning the programs to the correct solvers 90.7% of the time. Overall, these were good numbers for a first attempt, but there were some very real drawbacks to this approach.

3 Discussion and Future Work

At first glance, 90.7% seems pretty good, but when I dug a little deeper, I realized that the neural net had zeroed in on a particular solver which worked for almost every testable case. I say "testable" because some of the cases were not successfully solved by any of the solvers. Therefore, I considered them untestable and I removed them from the training algorithm. However, of the benchmarks that were solved, one of the solvers solved significantly more than the others and solved them in better time than the others. Overall, my artificial intelligence zeroed in on a dominating strategy and stabilized. However, there were occasions when the correct choice would have been the enumerative solver and, because my neural net was so set on the CVC4-1.5-syguscomp2015-v4 solver, it didn't

identify the occasions when the enumerative solver would have been more effective.

In order to improve the performance of the neural net further, I am going to look into doing two things:

1. I'm going to look for more benchmarks. There are allegedly more benchmarks on the starexec community, but the website has been down for a week or two and while it was up I had trouble finding them. If I can't find the rest of the benchmarks soon, I'll look into requesting more.

2. Once I have more benchmarks, it will be more feasible to add more layers. The code was written to be scalable and I believe adding extra layers would be quite reasonable from that perspective.

3. I'm going to do more research into backpropagation so that I can make sure to get every bit of performance improvement I can from the neural network.

When all that is done, I will have a fully functional, well-trained neural network. If it still cannot get that 10% performance boost, I will try removing the dominating solver so that I can evaluate the neural net on a more interesting set of data.