CS 517 Final Project: Using an SMT Solver for Training Feedforward Neural Networks

Charles Ison[*]
Oregon State University

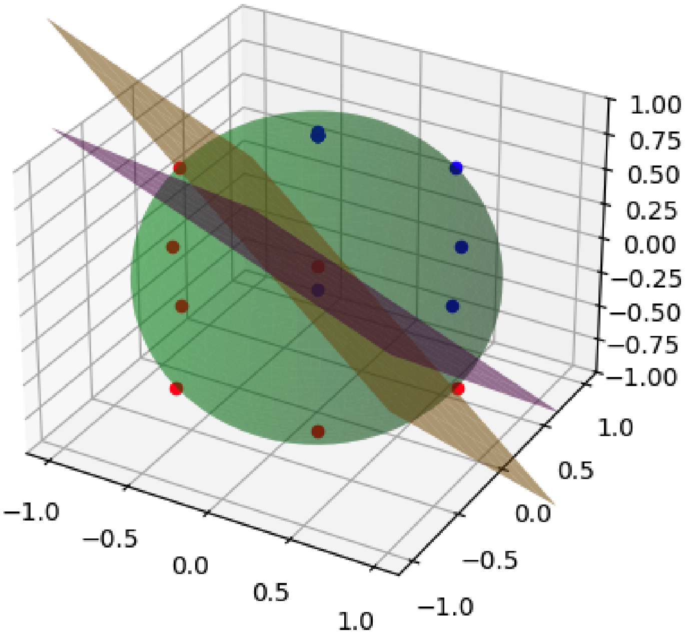

Figure 1: Artificially generated training data for neural networks where points are sampled from a unit sphere and then classified based on position relative to two randomly generated planes.

1 Introduction

Neural networks are typically trained using a gradient descent based algorithm called backpropagation. This technique has found widespread adoption due to its scalability and suprisingly high performance [Lecun et al. 1998], but unforunatly offers no guarantees regarding convergence to an optimal solution. An alternative approach is to directly solve and set the neural network's weights using a Simple Modulo Theory (SMT) solver. This approach is conceptually easier to understand than backpropagation, but unfortunately finding an optimal solution is NP-hard and quickly becomes inefficient as the size of the training data and neural network grows. For this project, an algorithm is proposed to train any feedforward neural network with rectified linear unit (ReLU) activations using an SMT solver. Finally, a performance test is run on neural networks of increasing sizes to compare using backpropagation vs the SMT solver algorithm proposed.

2 Background

Training neural networks using gradient descent first started to see success with the proposal of backpropagation in 1986 [Rumelhart et al. 1986]. Slowly this approach found wider acceptance and started being applied to more complicated problems such as computer vision in 1998 [Lecun et al. 1998]. As GPU advancements and computational power have continued to increase since then, the size and abilities of modern neural networks has also continued to increase. Despite the wide variety of recent advancements, backpropagation has consistently remained the most widely used supervised training algorithm and is supported by popular libraries like PyTorch [Paszke et al. 2019] and TensorFlow [Abadi et al. 2015].

Backpropagation works by first defining some loss function for a model's output, computing the loss based for some input data, finding the gradient of loss with respect to the model's weights, and then updating the weights in a direction that minimizes the loss function. Although this optimization process works remarkably well for its simplicity, there are criticisms that the algorithm can get stuck in local minimums and can be unstable. Figure 2 shows an example visualization of a neural network's ”loss landscape” which the algorithm for backpropagation is effectively searching for a global minimum. From Figure 2, it is easy to see how when using backpropagation, the algorithm could get stuck in a local minimum rather than reaching the ideal global minimum [Li et al. 2017].

Given the instability of backpropagation, an alternative approach could be to directly solve for the weights given a training dataset and an SMT solver. Although this approach does not scale well for larger models or datasets, it demonstrates the NP-hardness of training neural networks and helps justify why some trade-offs are required. Previous work done in this area includes demonstrating that training a simple 3-node neural network is NP-complete [Blum and Rivest 1988] and showing that training neural networks with an ReLU activation function is NP-hard [Boob et al. 2020]. Showing that ReLU training is NP-hard was completed using a reduction

[*] e-mail: isonc@oregonstate.edu

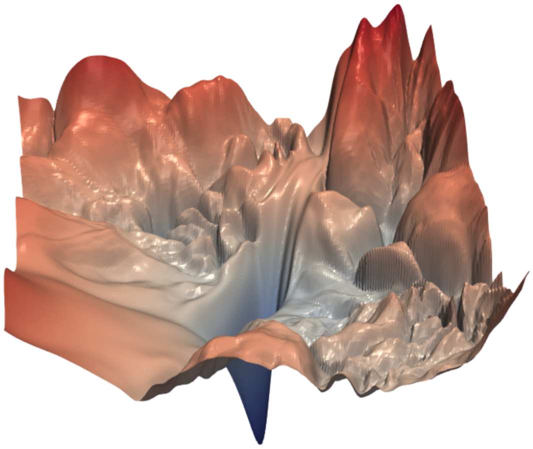

Figure 2: Example ”loss landscape” for a neural network [Li et al. 2017]. A gradient descent optimizer might get stuck in one of the many local minimums rather than reaching the ideal global minimum.

from the hyperplane separability problem, which inspired the training dataset used for this project (see Figure 1).

3 Methods

Previous work demonstrating the NP-hardness of training neural networks has focused on simple feedforward models. For this project, a generic algorithm is proposed that can take any feedforward neural network of arbitrary dimension with ReLU activations and find the optimal weights for a given training dataset using an SMT solver. The input to the SMT solver will be a series of equations that is generated for every individual data point in the training dataset. Therefore, there will only be a satisfiable solution if a configuration of weights exists for the neural network that is correct for every data point in the training dataset.

Feedforward neural networks are the most vanilla implementation of neural networks without special features such as convolutions or recurrence that are common in computer vision or natural language processing models. See Figure 3 for an example two-layer, three node feedforward neural network that is NP-complete to train. [Blum and Rivest 1988]. The proposed algorithm will handle nodes of the following structure:

$$f(x,y,z) = ReLU(w_0x + w_1y + w_2z + b) \tag{1}$$

$$ReLU(x) = max(0,x) \tag{2}$$

Here w_0, w_1, w_2 and b are the weights that need to be ”learned” during the training process. Also, x, y, and z are just an example 3-dimensional input, but any dimensional input is also supported by the algorithm. There are many possible activation functions, but ReLU is the most common due to its ease of calculation and help with preventing vanishing gradients [Tan and Lim 2019]. For this reason along with an existing proof of NP-hardness [Boob et al. 2020], this is the reason ReLU activations will be supported by the proposed SMT training algorithm for any number of nodes and any number of layers in the neural network.

Although this node mental model is typically used to discuss neural networks, the underlying implementation and interface in popular machine learning libraries is typically just a matrix. For example, in the neural network from Figure 3, the first layer that accepts inputs of dimension n and has two nodes would just be represented as the following matrix:

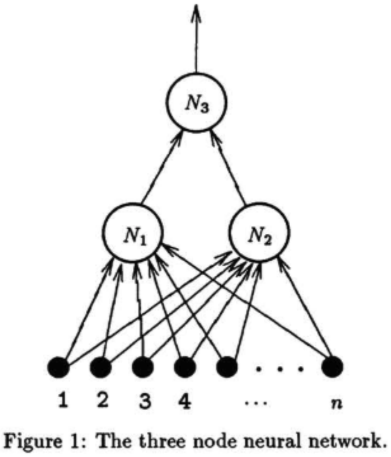

Figure 3: Example two-layer, three-node neural network [Blum and Rivest 1988].

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} & b_0 \\ w_{1,0} & w_{1,1} & \dots & w_{1,n} & b_1 \end{bmatrix} \tag{3}$$

With this in mind, it becomes clear that the algorithm to take a neural network and build the SMT solver's input will need to handle recursive matrix multiplication. Fortunately, matrix multiplications maps relatively intuitively to traditional SMT problems. The interface pySMT [Gario and Micheli 2015] was used for building the SMT equations here and these were then passed into a Z3 solver [de Moura and Bjørner 2008]. pySMT represents each step in an equation as a tree node with two input branches, which worked well for the recursive nature of multiplying layers' matrices. The code for this implementation can be found in the included 'smt_training.py' file and runs in polynomial $O(plmn)$ time where p is the number of data points, l is the number of layers in the network, m is the number matrix rows, and n is the number of matrix columns.

After the matrix multiplcation between each layer's inputs and weights, each value in the resulting matrix needs to pass through the ReLU activation function (Equation 2). Fortunately, pySMT has built-in support for taking the max of two values and adding it as a node in a formula's tree.

Next, the output from the model needs to be mapped back to a corresponding label for loss computation. For classification tasks, a common neural network output is an array of logits where each array index corresponds a potential label's class. Then the array index with the largest output logit corresponds to the neural network's prediction. This allows the model to be trained with backpropagation using cross entropy as a loss function:

$$CE(P^*,P) = -\sum_i P_i^* \log(P_i) \tag{4}$$

where P^* represents the true probability distribution from the labels and P represents the model's predicted probabilities. In order to directly compare the training performance using an SMT solver vs backpropagation, the algorithm proposed needs to also handle the output array of logits. After the algorithm steps described previously, we will have an array of formulas for consideration. Then the array can be consolidated by taking the formula at the corresponding label's index and setting it as greater than each of the other remaining formulas.

For example, if there were three possible classes for the model to predict and the label indicated the formula at index 2 was correct, the the transformation would look as follows:

$$\begin{aligned}&[w_0x_0+b_0, w_1x_1+b_1, w_2x_2+b_2] \rightarrow \\ &\qquad w_2x_2+b_2 > w_0x_0+b_0 \wedge w_2x_2+b_2 > w_1x_1+b_1\end{aligned} \tag{5}$$

The runtime for this transformation is just $O(pc)$ where p is the number of data points and c is the number of classes. Therefore, this transformation step is insignificant for the overall time complexity for the transformation as it is dominated by the matrix multiplication and ReLU steps.

The transformation process described in this algorithm have been implemented to accept any PyTorch [Paszke et al. 2019] feedforward neural network and corresponding training data as input. Then the PyTorch neural network is transformed to a series of equations for each input in the training data, pySMT and Z3 are called to find the ideal weights, and finally the weights are set back on the neural network. A separate testing function is called after the process is complete to ensure the solution results in 100% accuracy. The only stipulations to using the SMT trainer proposed is that the neural network activation functions are only ReLU and there must be list populated on the network that contains each layer's weights (so the algorithms knows how to find them). The goal was to make the implementation as generic as possible, so other interested parties could easily extend the trainer for their work.

Finally, a toy dataset was generated for testing the tool. The dataset was created by sampling points from the surface of a unit sphere and then deciding their classification based on their position relative to two planes:

$$ax+by+cz>0 \tag{6}$$

$$dx+ey+fz>0 \tag{7}$$

where the coefficients for a, b, c, d, e and f were determined using a random number generator. This means the model just has to perform binary classification depending on which side of the plane the point sits, but the SMT training algorithm proposed also supports multiclass classification. See Figure 1 for a visualization of the training data. With this data, the same model was trained using both backpropagation and the SMT solver and the resulting training times were recorded. Backpropagation training was not considered complete until 100% accuracy was achieved on the training dataset. This process was then repeated while linearly increasing the size of the neural network.

4 Results

The algorithm proposed for training neural networks using an SMT solver was able successfully train feedforward networks with an arbitrary number of layers and nodes, but the solver also quickly runs into time limitations. For the SMT solver, the training time increases exponentially as the number of weights in the network increases (Figure 4). At 400 model weights, the SMT solver takes 10 seconds and for any larger number of weights, the solver struggled to consistently return a response. A neural network with 400 weights corresponds to 53,821 equations being passed to the Z3 SMT solver. Interestingly, backpropagation training initially shows the opposite relationship and the runtime decreases as the number of model weights increases (Figure 5). This is most likely caused by backpropagation requiring more training iterations with the fewer weights because there is a narrower range of acceptable values. This is a special case where the neural network is overfitting the toy artificial dataset and, in general, a larger number of weights requires more training time during backpropagation. This was confirmed in Figure 6 where the number weights was increased exponentially (beyond what SMT could support), and then training time for backpropagation does begin to increase.

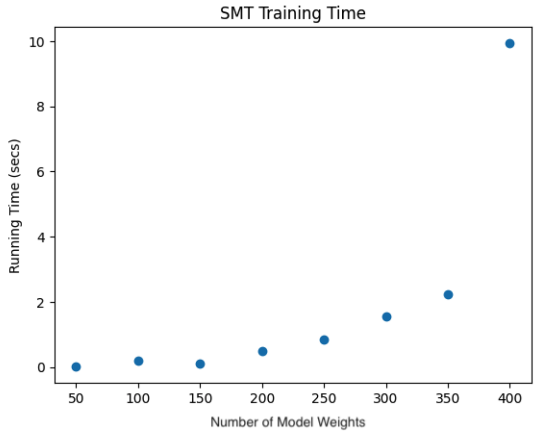

Figure 4: Running time required for SMT training vs number of model weights.

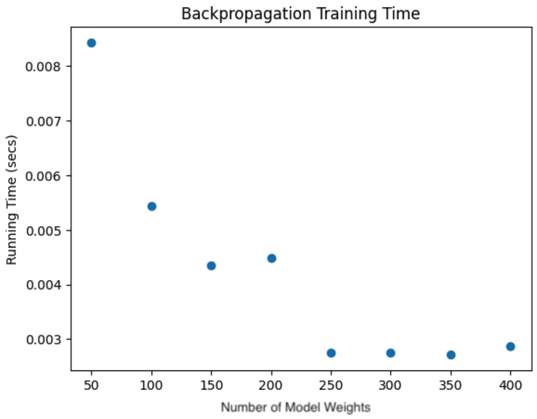

Figure 5: Running time required for backpropagation training vs number of model weights.

5 Conclusion

The training time required for neural networks using backpropagation on real world data can vary drastically based on factors such as hardware, model size, training data size, optimizer, and the initial weights. Considering these many factors and and the potential for non-optimal, locally minimum solutions, training with an SMT solver might seem like a functional alternative for small networks. Unfortunately, for a network with more than 400 weights, using a pySMT interface with Z3 solver appears to struggle to find a solution. While 400 weights might sound substantial, it is not near the scale of most modern neural network architectures. Standard computer vision models able to run on personal laptops can have over 19.4 million weights [He et al. 2015] and large language models such as GPT-3 can have over 175 billion weights [Brown et al. 2020]. Despite the impracticality of using SMT solvers as trainers for neural networks, attempting to train a model with one provides a justification on why the trade-offs for less stable techniques such as backpropagation are necessary. Finally, a related and interestingly line work being done is attempting to verify the behavior of neural networks using SMT solvers [Katz et al. 2017]. This can be important for fields where safety and understanding model behavior is extremely important, such as airborne collision avoidance systems. A natural next step for this project could be shifting to this application area instead of training.

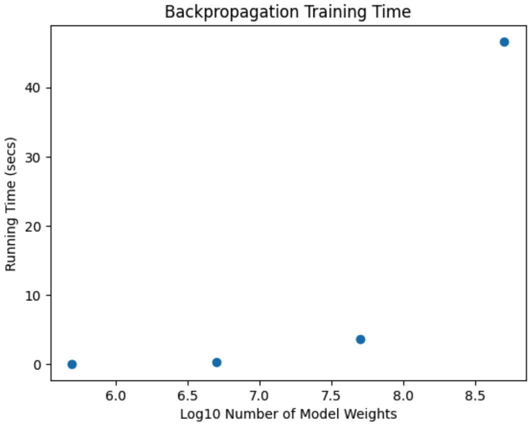

Figure 6: Running time required for backpropagation training as the number weights increases exponentially.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X., 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Blum, A., and Rivest, R. 1988. Training a 3-node neural network is np-complete. In *Advances in Neural Information Processing Systems*, Morgan-Kaufmann, D. Touretzky, Ed., vol. 1.

Boob, D., Dey, S. S., and Lan, G., 2020. Complexity of training relu neural network.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. 2020. Language models are few-shot learners. *CoRR abs/2005.14165*.

de Moura, L., and Bjørner, N. 2008. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, C. R. Ramakrishnan and J. Rehof, Eds., 337–340.

Gario, M., and Micheli, A. 2015. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *SMT Workshop 2015*.

He, K., Zhang, X., Ren, S., and Sun, J. 2015. Deep residual learning for image recognition. *CoRR abs/1512.03385*.

Katz, G., Barrett, C., Dill, D. L., Julian, K., and Kochenderfer, M. J. 2017. Reluplex: An efficient smt solver for verifying deep neural networks. In *Computer Aided Verification*, Springer International Publishing, Cham, R. Majumdar and V. Kunčak, Eds., 97–117.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11, 2278–2324.

Li, H., Xu, Z., Taylor, G., and Goldstein, T. 2017. Visualizing the loss landscape of neural nets. *CoRR abs/1712.09913*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S., 2019. Pytorch: An imperative style, high-performance deep learning library.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. 1986. *Learning internal representations by error propagation*. MIT Press, Cambridge, MA, USA, 318–362.

Tan, H. H., and Lim, K. H. 2019. Vanishing gradient mitigation with deep learning neural network optimization. In *2019 7th International Conference on Smart Computing Communications (ICSCC)*, 1–4.