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

Charles Ison*
Oregon State University

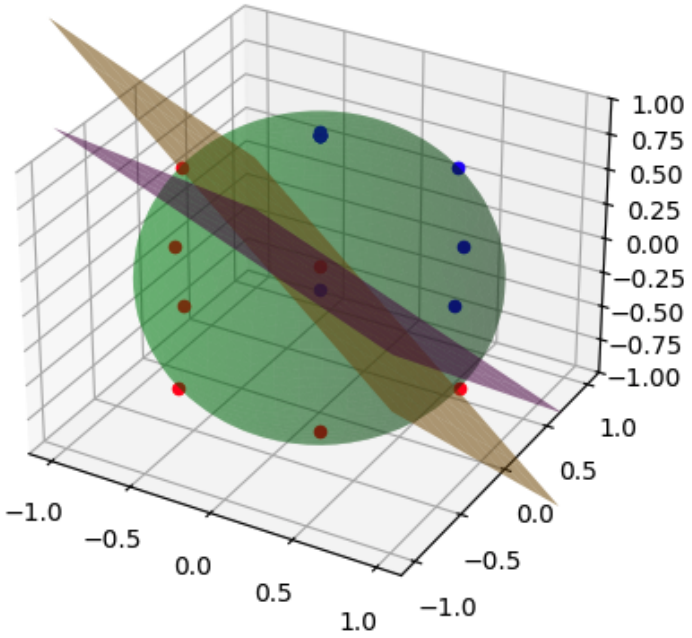

Figure 1: Artificially generated model training data with points sampled from a unit sphere and then classified based on position relative to two randomly generated planes.

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 an Simple Modulo Theory (SMT) solver (assuming some optimal solution exists for the training data). 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 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, an empirical comparison of the performance for backprogagation vs the SMT solver training is given.

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].

Intuitively, backpropagation works by first defining some loss function for a model's output, computing the loss based on some training data input, computing the gradient of the model's weights using the chain rule, and then updating the weights in a direction that minimizes the loss function. Although historically this optimization process has worked remarkably well, 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" where it is easy to see how the algorithm could get stuck in a local minimum rather than reaching the global minimum [Li et al. 2017].

Given the instability of backpropagation, an alternative approach would be to directly solve for the weights given a training dataset and an SMT solver. Although this approach does not scale well to larger, more complicated models, it is a fun exploration of the NP-hardness of training neural networks. 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 is completed using a reduction from the hyperplane separability problem which has inspired the training dataset used for this project.

*e-mail: isonc@oregonstate.edu

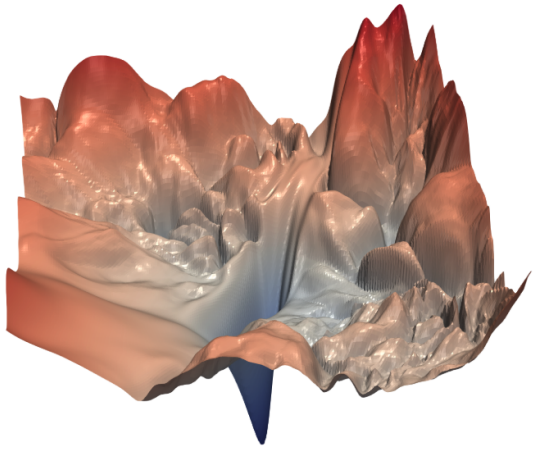

Figure 2: Example "loss landscape" for neural network [Li et al. 2017]. It is clear a gradient descent optimizer might get stuck in a local minimum rather than reaching the global minimum.

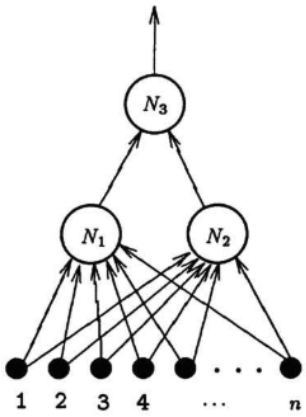

Figure 1: The three node neural network.

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

3 Methods

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

For a specific node in the network, the calculation would be defined as follows for a three dimensional input:

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

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

Here w_0, w_1, w_2 and b are the weights that need to be "learned" during the training process. 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, 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 example neural network seen in Figure 3, the first layer that accepts inputs of dimension n and has two nodes would just be represented as the following matrix:

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

With this in mind, it becomes clear that the algorithm to generate the SMT solver's input will need to handle recursive matrix multiplication. Fortunately, this maps relatively intuitively to traditional SMT problems and the tree interface of pySMT [Gario and Micheli 2015] made developing the recursive formula straightforward (**reference code here**). pySMT is a formula building interface to many SAT/SMT solvers and under the hood Z3 was used for this project [de Moura and Bjørner 2008].

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

Next the output from the model needs to be mapped back to the 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 times 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 currently have an array of formulas for consideration. Then each of these formulas can be combined by setting the formula at the label's index as greater than each of the other remaining formulas.

For example, if there were three possible classes for the model to predict and label indicated class 3 was correct, the the transforma-

tion would look as follows:

$$[w_0x_0 + b_0, w_1x_1 + b_1, w_2x_2 + b_2] \rightarrow$$
$$w_2x_2 + b_2 > w_0x_0 + b_0 \wedge w_2x_2 + b_2 > w_1x_1 + b_1 \quad (5)$$

Next, the proposed algorithm takes any Pytorch [Paszke et al. 2019] implementation of a feedforward neural network as input, performs the formula transformation discussed, calls pySMT to find the ideal weights, and sets the weights back on the neural network. Finally, a separate testing function is called to ensure the satisfiable solution result in 100% accuracy. The only stipulations to using the SMT trainer proposed is that the neural network activation functions are only ReLU and that a list must be 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 algorithm in order to ensure each data point is satisfiable. 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:

$$Ax + By + Cz > 0 \quad (6)$$
$$Dx + Ey + Fz > 0 \quad (7)$$

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.

4 Results

The algorithm proposed for training neural networks using an SMT solver is able successfully train any feedforward network of an arbitrary number of layers and nodes, but the solver also quickly runs into time limitations. In order to compare these SMT training time against backpropagation, a simple test model was created and trained using both algorithms. Much like the work from [Blum and Rivest 1988], a two layer neural network was used, but the number of internal nodes linearly was fluctuated. This allowed for the comparison of the number of weights in the network vs the required training time using SMT and backpropagation.

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 number of weights larger the solver struggled to consistently return a response. Interesting, 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 thew 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. Even for the toy dataset, if the number of weights is increased exponentially substantially passed what SMT could support, then the training time for backpropagation does begin to increase (Figure 6).

5 Conclusion

The training time required for neural networks using backpropagation on real world data can vary drastically based on factors such as

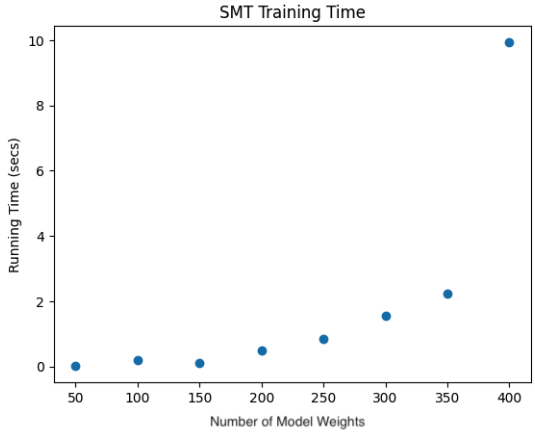

Figure 4: Running time required for SMT training.

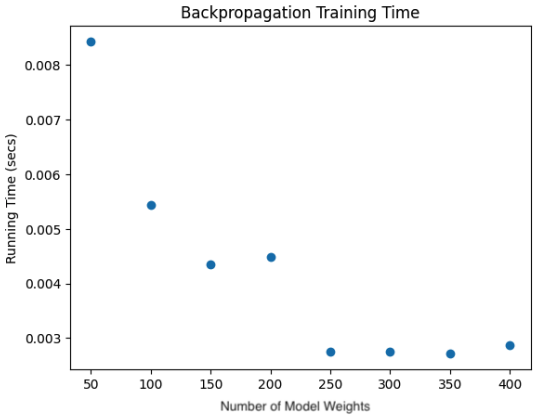

Figure 5: Running time required for backpropagation training.

hardware, model size, training data size, optimizer, and the initial weights. Considering these 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 any 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 just neural network training.

References

ABADI, M., AGARWAL, A., BARHAM, P., BREVDO, E., CHEN, Z., CITRO, C., CORRADO, G. S., DAVIS, A., DEAN, J.,

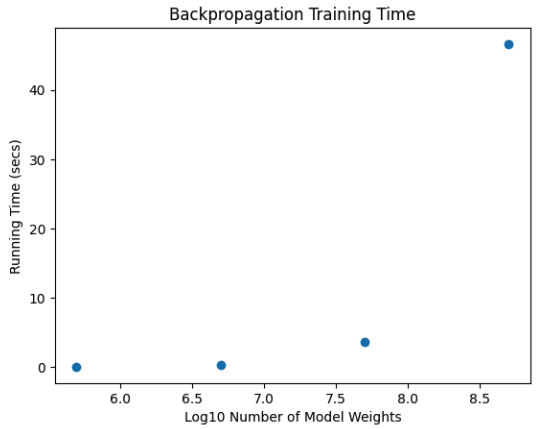

Figure 6: Running time required for backpropagation training as the number weights increase exponentially.

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.