Adapting Symbolic Linear Relaxation to the Sigmoid Activation Function in Neurify

Andrew Burford
Department of Computer Science
Stony Brook University
Stony Brook, NY, USA

***Abstract*—In this paper we modify the symbolic linear relaxation algorithm used in the Neurify verification system to support the sigmoid activation function. We use our modified version of Neurify to evaluate a neural network used to tune the block device readahead parameter in operating systems. This neural network represents the very recent application of machine learning models to tune operating system parameters, which is a new area that can benefit from verification tools. We also evaluate the readahead neural network using other state of the art property verification tools. We show that our modified Neurify implementation yields comparable performance to other neural network verifiers.**

I. Introduction

Many recent advancements in the training and development of neural networks paired with the continued improvement of computational processing power has lead to a vast and diverse range of machine learning applications. Many promising applications exist in safety critical systems where an incorrect decision can have severe consequences. It is therefore necessary to develop methods for verifying that neural networks will produce acceptable decisions for an entire range of possible inputs. For systems with very complex input spaces such as image data used for image classification, a common property to verify is *local robustness*, which ensures that the set of inputs reachable by a small perturbation from a given input all produce the same output. Even for applications outside of safety critical systems, verification of this property can increase our trust in relying on neural networks to behave as expected.

One very recent application of neural networks is to tune operating system level parameters [1]. Traditionally, simple heuristics with well defined behavior and provable guarantees are used for tuning OS parameters. In order to provide a viable replacement for such algorithms, kernel level neural networks must provide reliability and robustness as well. Due to the novelty of this application, there is not a clear choice for which activation function to use between layers. In the work of Akgun et al., a classification neural network with a sigmoid activation function was used to tune the block device readahead parameter. Ideally, any neural network verification technique should support as diverse a range of activation functions as possible. Moving towards this goal, in this paper we adapt an existing verification method used in the Neurify software to handle the sigmoid activation function. We then evaluate several properties of the readahead neural network for a set of hand picked input spaces. We also evaluate this network using other verification tools and compare their efficiency and results.

II. Related Work

An input that is very close to another input but classified differently by a neural network is sometimes referred to as an adversarial input. Initial approaches to dealing with the issue of eliminating adversarial inputs and therefore increasing local robustness involved modifications to the training of the neural network [3]. The robustness was then measured by the success rate of an algorithm attempting to find adversarial examples for a given network. However, this type of approach is far from a guarantee that adversarial examples do not exist. It is possible that the network is just overfitting the types of examples generated by the adversarial generation algorithm. There are also no guarantees about the ability of the adversarial generation algorithm to exhaustively search for adversarial inputs.

There exist a wide variety of tools and techniques that can fully verify the local robustness of a neural network. Many verification tools focus primarily on neural networks with the piece-wise linear ReLU activation function since it is easier to model than completely non-linear functions such as sigmoid. In fact, there exist algorithms capable of computing the exact output set of a ReLU network. For example, W. Xiang et al. [12] present the ExactReach algorithm which operates by splitting up the input set into the union of a set of convex polytopes. However, the number of polytopes grows exponentially with both the dimension of each layer and with the number of layers so this does not scale well. Many solvers deal with this by overapproximating the reachable set and using divide and conquer to search for inputs that violate robustness. The Neurify [9] software that we modify in this paper uses a technique called symbolic linear relaxation to over approximate the output set and then a technique called direct constraint refinement to search for inputs violating robustness. In this paper, we extend the implementation of Neurify's techniques to work with the non-linear sigmoid activation function. These techniques are built upon or otherwise related to a range of methods one can use to over approximate and search.

Symbolic linear relaxation builds upon symbolic interval analysis which in turn builds upon arithmetic interval analysis. Arithmetic interval analysis is used by W. Xiang et al. [11]

in their tool MaxSens. It consists of coming up with concrete upper and lower bounds for each node in a layer by multiplying the upper and lower bounds of the previous layer by the corresponding weights for each node. This corresponds with propagating a hyperreptangle through the network, which the ReLU function can be easily applied to by truncating the parts of the rectangle with any negative dimensions. Tighter bounds are achieved by breaking up the input space into many small pieces. S. Wang et al. [10] introduced symbolic interval analysis in their tool ReluVal. Instead of using concrete bounds for each node, they compute symbolic bounds which are expressed as linear combinations of the values of previous nodes. If the bound for a node straddles the non-linear portion of the ReLU function, it is concretized to actual numerical upper and lower bounds (where the lower bound will be zero). Neurify also uses symbolic bounds for each variable, but instead of concretizing the bounds of nodes with a negative lower bound and positive upper bound, they encode two linear constraints to overapproximate the output of the ReLU function. The inequalities for these constraints are expressed symbolically just like the upper and lower bounds.

G. Singh et al. [8] used similar symbolic interval expressions in the DeepPoly tool. Each node has a symbolic interval referencing the values of previous nodes. However, their algorithm uses an entirely different abstract domain which consists of a single upper polyhedral constraint, a single lower polyhedral constraint, and auxiliary concrete upper and lower bounds for each variable. This is coupled with a set of abstract transformers to propagate an input set through the network.

Robustness verification can also be approached as an optimization problem, and some similar techniques can be used. For example, Bunel et al. [2] present an optimization algorithm which uses branch and bound to search through the input space. This approach involves encoding the network as a set of linear constraints with integer decision variables used to represent the piece-wise linearity of ReLU activation functions. This can be fed into any Mixed Integer Linear Programming solver to find the minimum perturbation from an input point which does not fall in the output domain. The branch and bound technique is used when partitioning the input domain into different sets and propagating each set. If the overapproximation of a partition is contained within the desired output set, it can be ignored. Otherwise, the algorithm iteratively partitions into smaller input spaces until a tight enough bound is reached. This is essentially the same process used in ReluVal to split the input space using iterative refinement. However, Neurify actually improves the splitting operation in ReluVal's branch and bound search by splitting only the nodes in hidden layers where non-linearity occurs instead of splitting the input domain.

Another approach that involves optimization is the ReluPlex [5] algorithm. This approach builds on top of the simplex algorithm by maintaining an assignment of variables that may violate some of the nonlinear ReLU constraints. It then prioritizes fixing the violated ReLU constraints as it optimizes the variables. ReluPlex was built to specifically handle the ReLU function, but G. Katz et al. [6] extended ReluPlex with the creation of Marabou which works any piecewise linear function. Marabou also increases efficiency by adding support for a divide and conquer solving mode similar to the framework introduced in Bunel's branch and bound paper. Other optimization algorithms can actually deal with any arbitrary activation function, not just piecewise linear. For example, the Duality [4] tool solves the Lagrangian dual problem where the non-linear constraints are added to the objective function. This allows the lagrangian of the objective function to guide non-linear optimization for any kind of activation function. For fully linear optimization, the solution to the dual problem will equal the solution to the primal problem. For the Lagrangian dual used in Duality, the solution to the dual problem represents an upper bound to the maximum perturbation, similar to the reachability overapproximation algorithms. The issue with these optimization approaches is that the high non-linearity in the function represented by the neural network limits the efficiency of the algorithm.

III. Adapting Neurify

The two main pieces of the Neurify algorithm are symbolic linear relaxation and direct constraint refinement. Symbolic linear relaxation is a technique for propagating the input set through the network, and direct constraint confinement is a technique for splitting the input set into different subsets so as to minimize overapproximation. In the original paper, the ReLU activation function was exclusively considered for both parts of the algorithm. First, we review the general idea behind symbolic linear relaxation, and then we describe the necessary changes to extend it to the sigmoid activation function. We then describe the implications that a different activation function have on direct constraint refinement.

A. Symbolic Linear Relaxation

Symbolic linear relaxation is a relatively small modification to symbolic interval propagation. Symbolic interval propagation is the technique of representing the upper and lower bounds for each node in a neural network as a linear combination of vectors in the input space. This way, we keep track of dependencies between the nodes in each layer as we propagate the upper and lower bounds. As a small example, consider a network with two layers and two nodes in each layer. The weights matrix between the layers is $\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$ and the bias vector is the zero vector. The ReLU function is applied after the weights matrix. The nodes in the first layer are labeled x_i where $x_1 \in [-1, 2]$ and $x_2 \in [0, 3]$. This corresponds to a hyperrectangle input space. After propagating through the weights matrix but before applying the ReLU function, we label the nodes y_i. Each y_i has a linear combination of x_i variables to represent an upper bound and another linear combination of x_i variables to represent a lower bound. Since we haven't had to introduce any overapproximation yet, these upper and lower bounds are the same. Note that you can interpret each bound as a zonotope where the generator vectors

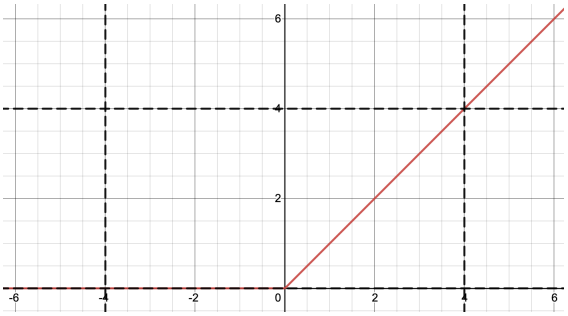

Fig. 1. This rectangle represents the overapproximation that happens when a node is concretized during symbolic interval propagation

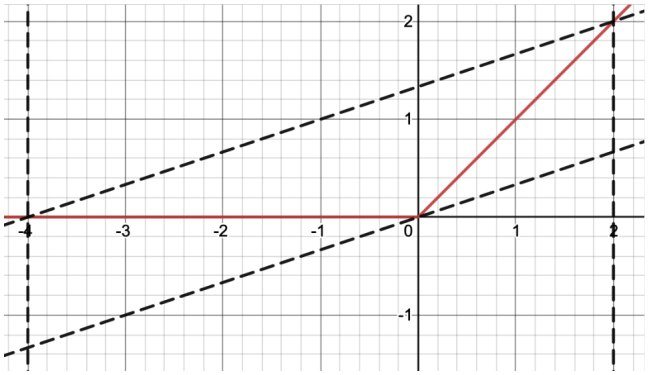

Fig. 2. This is a visualization of the parallelogram relaxation used by Neurify to overapproximate the ReLU function. The upper bound line $y = \frac{u}{u-l}(x-l)$ is used to form the upper bound zonotope and the lower bound line $y = \frac{u}{u-l}x$ is used for the lower bound zonotope.

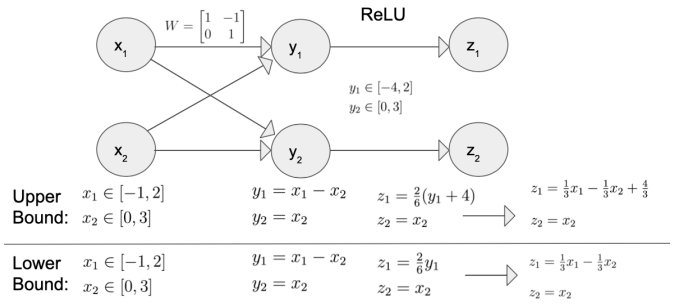

Fig. 3. This is a visualization of an example neural network with symbolic interval propagation used to propagate the input space through a weights matrix and ReLU function.

are the columns of the matrix of coefficients and the free variables are the input variables. For example, the upper bound for y_i is $y_1 = x_1 - x_2$ and $y_2 = x_2$ so the matrix of coefficients is $\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$ (the same as the weights matrix).

In order to propagate through the ReLU function we handle the upper bound and lower bound zonotopes separately. We first compute concretized upper and lower bounds for each node in the zonotope. This is a simple optimization to compute because the free variables are just the hyperrectangle input space. So for the upper bound of y_1 we compute the upper bound by taking the maximum of $x_1 = 2$ and subtracting the minimum of $x_2 = 0$, leaving us with 2. Similarly, we take the minimum of $x_1 = -1$ and subtract the maximum of $x_2 = 3$ to get a lower bound of $y_1 = -4$. These concretized bounds of $y_1 \in [-4, 2]$ mean that y_1 straddles the non-convex portion of the ReLU function so we must overapproximate. Standard symbolic interval propagation overapproximates by saving these bounds as the output of the ReLU function, dropping the dependencies of this node on the previous layer. This is shown pictorally in Figure 1.

Symbolic linear relaxation tries to improve this by instead using a parallelogram relaxation of the ReLU function as depicted in Figure 2. This is where the upper and lower bound zonotopes will differ. For the upper bound zonotope, we compute the upper bound constraint of y_1 using the formula $y = \frac{u}{u-l}(x-l)$ where u and l represent the concretized upper and lower bounds respectively. If we represent the output of the ReLU function as z_i, we end up with the equation $z_1 = \frac{2}{6}(y_1 + 4)$ for our upper bound zonotope. The lower bound can be computed with the equation $y = \frac{u}{u-l}x$ so the equation for the lower bound zonotope is $z_1 = \frac{2}{6}y_1$. y_1 can be expanded so that z_1 remains a linear combination of x_i variables. See Figure 3 for the final upper and lower bound zonotopes. Note that if the concretized bounds do not straddle the non-convex portion of the ReLU function, no overapproximation is necessary. For example, z_2 remains unchanged after the ReLU function because the lower bound was ≥ 0. If the upper bound is ≤ 0, the variable can be set to zero which is equivalent to setting its row in the coefficient matrix to all zeroes.

Also note that if the input space were an H-polytope instead of a hyperrectangle, we would be maintaining upper and lower bound star sets instead of zonotopes. The actual reachable set at any given step can be thought of as the convex hull of the upper and lower bound star sets. This is visualized in Figure 4.

B. Extension to Sigmoid

Symbolic linear relaxation works because it computes a single line for the upper bound and a single line for the lower bound of each node. In order to compute such linear bounds for any interval on the sigmoid function, we must break things up into cases.

In the first case, both upper and lower bounds are greater than zero. Here, the derivative of the sigmoid function is monotonically decreasing, so the lower bound can be a line connecting the two output points of the sigmoid function. The upper bound is the same line but shifted up to be tangent to the sigmoid function. This case is displayed in Figure 5. Similar logic can be followed for computing upper and lower bounds when both bounds are less than zero.

For the case where the lower bound is less than zero but the upper bound is greater than zero, we must further subdivide into two cases. Consider just the lower bound because the upper bound can be computed symmetrically. In the first case, the slope between the two output points is greater than the

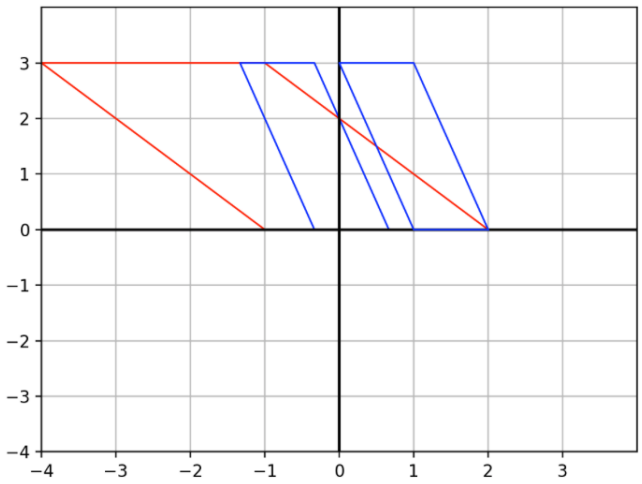

Fig. 4. This is a visualization of the output of the example neural network in Figure 3. The red paralellogram represents the output after the first linear layer and the blue parallelograms represent the upper and lower bound zonotopes after propagating through the ReLU function. The actual reachable set can be interpreted as the convex hull of these two zonotopes.

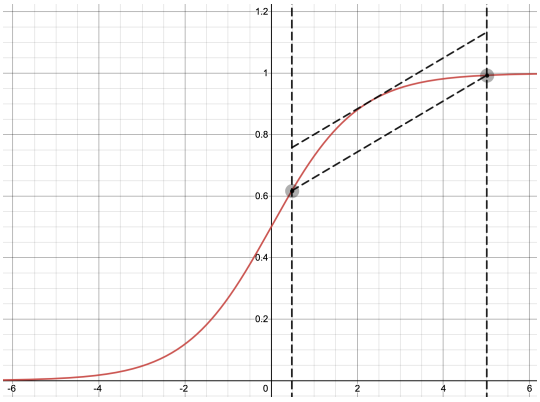

Fig. 5. Symbolic Linear Relaxation applied to the sigmoid function

derivative at the lower bound. Here we set the lower bound line to have slope equal to this derivative and pass through the lower bound output point. This choice allows our overapproximation to remain a strict subset of the overapproximation that results from symbolic interval propagation. If the slope of this derivative is greater than the slope between the two output points, we set the lower bound line to the line connecting the two output points. See Figure 6 and 7 for examples of both cases.

C. Implications for Direct Contraint Refinement

Direct constraint refinement is a novel approach introduced in Neurify for splitting up the input set into smaller sets that are verified independently. In order to choose how to split the input space, direct constraint refinement looks at nodes within the network that cause overapproximation by straddling the non-convex portion of the ReLU function. Within this set of nodes, it chooses the node with the largest output gradient. This gradient is a measure of how much effect this node has

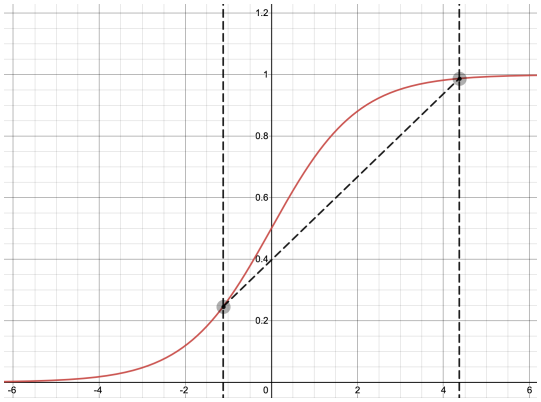

Fig. 6. Symbolic Linear Relaxation applied to the sigmoid function

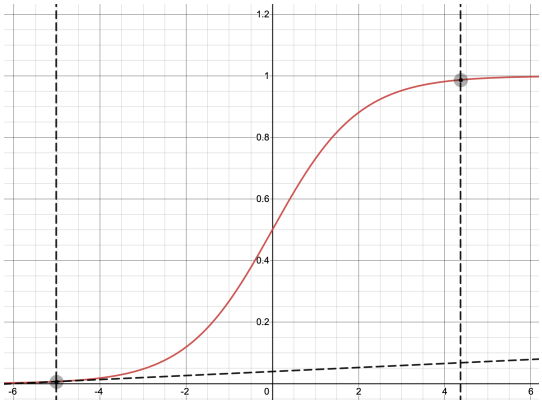

Fig. 7. Symbolic Linear Relaxation applied to the sigmoid function

on the output. It is computed during propagation through the network and for symbolic linear relaxation, it is simply equal to the slope of the overapproximated bounds or the slope of the ReLU function. So for any node whose concretized bounds required overapproximation of the ReLU function, the gradient is $\frac{u}{u-l}$.

Once a node is selected, a half space constraint is added to the input space. On one side of this halfspace, the chosen node should have a concretized upper bound ≤ 0. On the other side of the halfspace, the chosen node should have a concretized lower bound > 0. In both cases, no overapproximation of the ReLU function is required. This means that if it performs enough splitting, the analysis will approach exact analysis as fewer and fewer nodes require any overapproximation. Neurify recursively performs this splitting on each smaller input set until it is able to verify the property or find a counter-example.

For the sigmoid activation function, every single node has some amount of overapproximation as long as the upper and lower bounds are not completely tight since every piece of the function is non-linear. This means we must consider all nodes as potential nodes to split on, not just the ones that straddle zero. We still use the same influence analysis to select a node since the influence analysis is independent of the type of activation function. The gradient of a node is just equal to the slope of the line used to overapproximate

the sigmoid function. Once we have a selected node, we use the same technique as Neurify to split the input space in two, but instead of using zero as a boundary point, we use the midpoint of the upper bound and lower bound. For example, consider a node with bounds $y \in [0, 4]$. The bounds of $sigmoid(y)$ are $\in [0.5, 0.982]$. Its gradient will be the slope of the line connecting the two points since this is the slope used for overapproximation ($\frac{sigmoid(u)-sigmoid(l)}{u-l}$). If the linear combination used to represent y is $y = x_1 + 2x_2 - x_3 + 1$, then the following half space constraint would be added to the input space to split on this node: $y \leq 2$. This would be represented as $x_1 + 2x_2 - x_3 - 1 \geq 0$ or equivalently $\begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix} x - 1 \leq 0$. The $y > 2$ constraint would be added to form the other input space.

IV. Evaluation of Readahead Network

We evaluate our modified Neurify algorithm by using it to verify properties about a neural network used to select an optimal readahead value. A well known problem in filesystem and operating system tuning is the assignment of an optimal readahead value. When a page from a file is read by an application, the readahead value tells the operating system how many additional pages past this one to also request from disk and store in the page cache. Since many applications request data sequentially, reading additional data past what was requested by the application can improve performance since future page requests can likely be loaded straight from cache.

The standard Linux kernel sets the readahead to a static value that works pretty well across a wide range of workloads. The readahead network we will verify properties of is a feed forward classification network with sigmoid activation functions. It classifies the application currently running as one of four different types of workloads and sets the readahead value to the optimal value for that workload. The optimal value for each workload type was experimentally determined by running examples of each workload with different readahead values.

A. Choice of Input Space

The readahead neural network could be evaluated for local robustness given a handful of ground truth input data points. However, an understanding of the trends in the input data can lead one to hypothesize about a range of possible properties one might want this network to satisfy. More complex properties with larger input spaces also create better benchmarks for comparing the Neurify verification algorithm with other network verification algorithms. Below, we discuss the input features of this network, and we choose two interesting properties to verify about it.

There are five features calculated every second by the readahead neural network based on values gathered from tracepoints placed in the kernel. These tracepoints deal with page accesses for files in the filesystem. Each tracepoint involves a page index number within a file. The five features fed into the network are the number of tracepoints called, the mean page index, the standard deviation of page indices, the mean absolute difference between page indices of consecutive tracepoint calls, and the current readahead value.

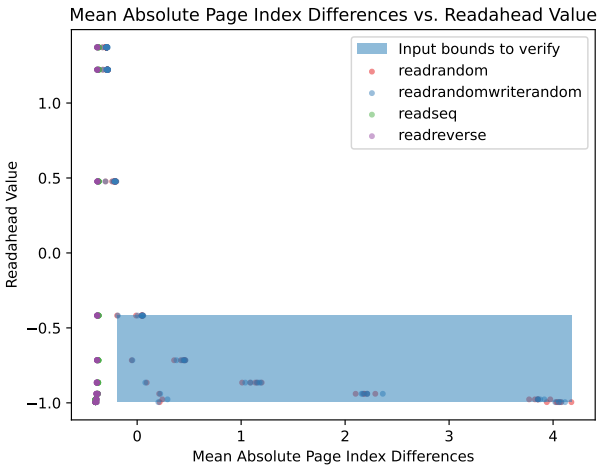

Fig. 8. The blue rectangle represents the bounds of a hyperrectangle for which all interior points should be classified as either readrandom or readrandomwriterandom. Note that the units on both axes are Z-scores based on the distribution of the entire training data set.

There exist complex, non-linear relationships between combinations of these variables, hence the necessity of utilizing machine learning for classification. However, we can categorize the output classes in pairs with similar features. One pair of workload classes with similar features are read sequential and read reverse. These classes are intended to represent an application that reads all data in a file sequentially in ascending page index order and an application that reads all data sequentially in reverse order, respectively. The other pair of workload classes are readrandom and readrandomwriterandom. The readrandom workload randomly picks offsets in a file to read from and readrandomwriterandom does the same thing except it also writes data to random points in a file.

By inspecting histograms and scatterplots of combinations of the input data feature variables, we identified two properties that the network should exhibit. In a scatter plot of readahead and mean absolute page index differences, there is a clear divide between clusters of readsequential and readreverse data points and clusters of readrandom and readrandomwriterandom data points as the readahead value decreases. In Figure 8, we highlight a rectangle representing the input space that should be classified as either readrandom or readrandomwriterandom and another rectangle that should be classified as readsequential or readreverse. For the other dimensions of this input space, we simply choose the minimum and maximum values for the data points in these classes because the classification should stay within these pairs of classes regardless of the other input features.

Another scatterplot with some clear clustering is the plot

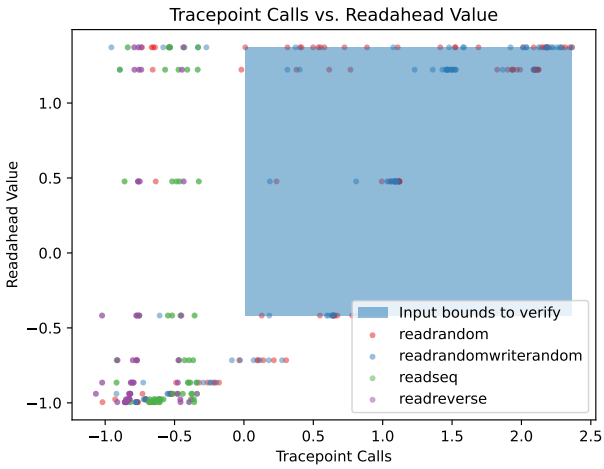

Fig. 9. The blue rectangle represents the bounds of a hyperrectangle for which all interior points should be classified as either readrandom or readrandomwriterandom. Note that the units on both axes are Z-scores based on the distribution of the entire training data set.

of the number of tracepoint calls vs. the readahead value. As readahead increases, the number of tracepoint calls for readrandom and readrandomwriterandom workloads steadily increases. In Figure 9, we highlight a rectangle for each workload pair which should be classified as that pair.

In a classification network, the output layer consists of one node for each class and the class with the largest value is selected as the output class. The constraint that either one of two output classes have the largest value is a non-convex output space, so we chose the tighter constraint that both workloads in one pair are ranked higher than both workloads in the other pair. If $x \in \mathbb{R}^4$ is the output vector, we can represent the constraint that the first pair of nodes be ranked higher than the last two using an H-Polytope like so:

$$\begin{bmatrix} -1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} x \leq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

B. Results

The hyperrectangle proposed above to encapsulate all readrandom and readrandomwriterandom data within a range of mean absolute page index differences does not uphold the property that the random pair of workloads are both ranked higher than the readseq and readreverse pair of workloads. We also tried verifying a smaller hyperrectangle, where the radii in all dimensions except mean absolute page index differences and readahead are set to zero, and an average is used for the center of the hyperrectangle in other dimensions. This also violates the property.

The hyperrectangle around a cluster of readrandom and readrandomwriterandom data points with a similar number of tracepoint calls also violates the property. Even when projecting the other dimensions to zero as explained above, the property remains violated.

V. Comparison To Other Verification Algorithms

We used an implementation of the MaxSens [11] algorithm provided by C. Liu [7] to verify the same properties.

Tool	Mean Diffs (s)	Tracepoint Calls (s)
Neurify	150.4	154.2
MaxSens	22.3	32.2

The tracepoint calls property took 3 iterations of the Neurify algorithm while the mean diffs property took just 1 iteration. Given the small difference in time taken to find a counterexample for these properties, this would suggest that the direct constraint refinement is effective at breaking up the domain into subsets that are much easier to calculate. This could also mean there is a high overhead for the first iteration of the algorithm.

Overall, our modified Neurify is definitely slower than MaxSense but it is still within an order of magnitude of the performance.

VI. Limitations and Future Work

Symbolic linear relaxation requires providing a single upper and lower bound linear constraint for a given activation function, so in theory this could be extended to any activation function. In addition, the implementation of our extension to the Neurify algorithm is based off an implementation provided by C. Liu [7] which was intended for pedagogical purposes. This means that the choice of language and code design could be significantly redesigned for speed rather than human readability and clarity.

Another type of optimization employed recently in neural network verification algorithms is optimizing the choice of linear bounds to minimize future overapproximation. Our upper and lower bounds for the sigmoid function attempt to minimize overapproximation, but alternative choices could potentially prevent even more overapproximation later on in future layers. This requires a more advanced analysis, but in theory it could readily be applied to the Neurify algorithm.

There is also the question of how to split up the domain during direct constraint refinement. For the ReLU activation function, the choice is easy because you simply split around boundary of the two linear segments of the function. However, there is not an obvious best choice for the sigmoid function. We simply chose bounds that allow our modifications to provide strictly smaller overapproximations than the concretized bounds that ReluVal would provide.

While this paper helps apply the field of neural network verification to the recent application of machine learning to tuning operating systems parameters, more work can be done to formalize better properties to verify and to modify the network so that these properties are actually upheld. We attempted to prove properties that one pair of classes ranked above another pair of classes, but ideally we wanted to prove that either class in one pair was the highest ranked class. If Neurify were extended to check for results in a non-convex output space or at least the union of convex output spaces then we could verify more general properties.

VII. Conclusion

In this paper, we introduce a modification to the Neurify algorithm to allow it to support the sigmoid activation function. In order to test our implementation, we evaluate several properties of a feed forward neural network used in the recent application of machine learning to tune operating system parameters. We found that the properties we wanted to verify were violated. This was then confirmed using alternative neural network verification algorithms that already support the sigmoid activation function. Specifically, we compared the performance of our modified Neurify implementation to an implementation of the MaxSens algorithm. Our implementation was significantly slower in finding counter examples to our input properties, but only by one order of magnitude.

References

[1] I. U. Akgun, A. S. Aydin, A. Shaikh, L. Velikov, and E. Zadok. A machine learning framework to improve storage system performance. HotStorage '21, page 94–102, New York, NY, USA, 2021. Association for Computing Machinery.

[2] R. Bunel, I. Turkaslan, P. H. S. Torr, P. Kohli, and M. P. Kumar. Piecewise linear neural network verification: A comparative study. *CoRR*, abs/1711.00455, 2017.

[3] N. Carlini and D. A. Wagner. Towards evaluating the robustness of neural networks. *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57, 2017.

[4] K. Dvijotham, R. Stanforth, S. Gowal, T. A. Mann, and P. Kohli. A dual approach to scalable verification of deep networks. *CoRR*, abs/1803.06567, 2018.

[5] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. *CoRR*, abs/1702.01135, 2017.

[6] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. L. Dill, M. J. Kochenderfer, and C. Barrett. The marabou framework for verification and analysis of deep neural networks. pages 443–452, 2019.

[7] C. Liu, T. Arnon, C. Lazarus, C. W. Barrett, and M. J. Kochenderfer. Algorithms for verifying deep neural networks. *CoRR*, abs/1903.06758, 2019.

[8] G. Singh, T. Gehr, M. Püschel, and M. Vechev. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.*, 3(POPL), Jan. 2019.

[9] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Efficient formal safety analysis of neural networks. *CoRR*, abs/1809.08098, 2018.

[10] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal security analysis of neural networks using symbolic intervals. *CoRR*, abs/1804.10829, 2018.

[11] W. Xiang, H. Tran, and T. T. Johnson. Output reachable set estimation and verification for multi-layer neural networks. *CoRR*, abs/1708.03322, 2017.

[12] W. Xiang, H. Tran, and T. T. Johnson. Reachable set computation and safety verification for neural networks with relu activations. *CoRR*, abs/1712.08163, 2017.