A Solver + Gradient Descent Training Algorithm for Deep Neural Networks

Dhananjay Ashok[1] , **Vineel Nagisetty**[2] , **Christopher Srinivasa**[2] and **Vijay Ganesh**[3]
[1]University of Toronto,
[2]Borealis AI,
[3] University of Waterloo
dhananjay.ashok@mail.utoronto.ca
{vineel.nagisetty, christopher.srinivasa}@borealisai.com
vijay.ganesh@uwaterloo.ca

Abstract

We present a novel hybrid algorithm for training Deep Neural Networks that combines the state-of-the-art Gradient Descent (GD) method with a Mixed Integer Linear Programming (MILP) solver, outperforming GD and variants in terms of accuracy, as well as resource and data efficiency for both regression and classification tasks. Our GD+Solver hybrid algorithm, called **GDSolver**, works as follows: given a DNN D as input, **GDSolver** invokes GD to partially train D until it gets stuck in a local minima, at which point **GDSolver** invokes an MILP solver to exhaustively search a region of the loss landscape around the weight assignments of D's final layer parameters with the goal of *tunnelling through and escaping the local minima*. The process is repeated until desired accuracy is achieved. In our experiments, we find that **GDSolver** not only scales well to additional data and very large model sizes, but also outperforms all other competing methods in terms of rates of convergence and data efficiency. For regression tasks, **GDSolver** produced models that, on average, had 31.5% lower MSE in 48% less time, and for classification tasks on MNIST and CIFAR10, **GDSolver** was able to achieve the highest accuracy over all competing methods, using only 50% of the training data that GD baselines required.

1 Introduction

Over the last few years, considerable amount of research has gone into algorithms for training Deep Neural Networks (DNNs), and yet, Gradient Descent (GD) and its variants remain the dominant approach for DNN training [Ruder, 2016]. The primary reason for this state of affairs is that GD-based training methods can easily handle a large variety of DNN architectures and are highly scalable in training very large DNN, achieving high accuracy with relatively little computational effort.

Having said that, despite their incredible success, GD-based methods [1] do suffer from a few significant weaknesses. First, GD and variants fundamentally lack the ability to distinguish between local and global minima, and hence may get stuck in local minima resulting in sub-optimal performance, generalization. Second, there are scenarios where GD and variants suffer from poor data efficiency, i.e., the amount of data needed to get reasonable accuracy can be very high. Finally, in recent years researchers have been able to show that DNNs suffer security, trust and robustness issues, e.g., adversarial attacks [Papernot *et al.*, 2016], and that training DNNs to be adherent to certain constraints is highly desirable [Verma *et al.*, 2019]. Unfortunately, GD and its variants can neither provide any guarantees, nor can they straightforwardly handle highly non-differentiable constraints that typically arise in the context of security and reliability specifications.

All these weaknesses suggest that there is considerable room for improvement, and there is an urgent need for researching new classes of DNN training algorithms. Recognizing the above-mentioned issues with GD and its variants, researchers have proposed Mixed Integer Linear Programming (MILP) solver-based training methods [Icarte *et al.*, 2019], among others. Such methods have the advantage that they can guarantee optimality, can alert users to infeasible problems, and handle highly non-differentiable constraints such as the ones that arise in security specifications that can potentially be added to the set of optimization constraints [Gupte *et al.*, 2013]. Unfortunately, solver-based methods suffer from significant problems of over-fitting to training data and very poor scalability vis-a-vis the size of the network being trained.

While there have been attempts to augment GD-based methods with optimizers (e.g., Adam) and learning rate scheduling techniques to overcome the oft-repeated problem of getting stuck in local minima, they do suffer from being heuristic in nature, i.e., they do not provide any gurantees that they have reached a global minima. Perhaps more importantly from a practical point of view, such additional optimizers also suffer from relatively poorer data efficiency.

To address these issues, we provide a new hybrid training algorithm for DNNs, called **GDSolver**, based on a combination of GD and an MILP solver (specifically we use the

[1]While a variety of GD methods are available today, we focus on methods that offer the best accuracy, are most scalable, and the most widely used as of this writing.

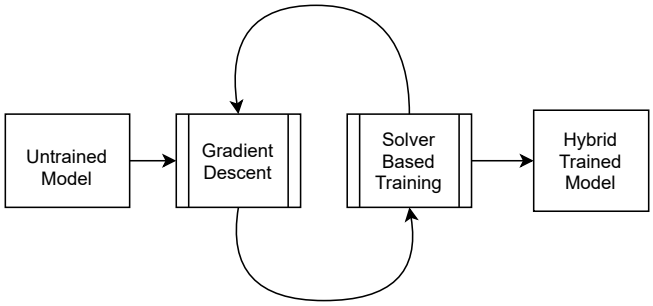

Figure 1: Hybrid **GDSolver** Architecture

state-of-the-art Gurobi MILP solver [Pedroso, 2011]). Given a DNN D and a training data set S as input, **GDSolver** initially invokes GD to train D using S until it gets stuck in a local minima (this can be detected using a variety of methods), at which point **GDSolver** then invokes an MILP optimization solver to exhaustively search a region of the loss landscape around the **current** weight assignments to *tunnel through and escape the local minima*. The GD and solver methods are invoked alternatively until an appropriate level of accuracy is achieved. When comparing **GDSolver** against multiple GD baselines on a suite of regression and classification tasks, we find that **GDSolver** not only scales well to additional data and model sizes, but also outperforms all other competing methods in terms of rates of convergence and data efficiency.

1.1 Key Contributions.

1. The **GDSolver** Algorithm - a novel hybrid training algorithm that iteratively calls GD and MILP solver in a way that it is able to escape local minima by "tunneling" through them. In order to accomplish this, we had to come up with a novel formulation of DNN Training as an MILP instance that solves the severe overfitting problem of previous MILP formulations and enables its usage with real-valued DNNs. The **GDSolver** algorithm is highly scalable in terms of being able to train very large DNN models, very general in terms of the DNN architectures it can handle, and data/resource efficient relative to competing methods [2].
2. **Extensive Experimental Evaluations:** We perform an comprehensive experimental evaluation of our algorithm against four state-of-the-art baselines, namely, Stochastic Gradient Descent (SGD), SGD with Learning Rate Scheduling (LRS), Adam Optimzation, and Adam Optimization with LRS, on a set of regression and classification tasks.
    - On a suite of regression equations, we show that **GDSolver** can produce models with, on average, 31.5% lower MSE in 48% less time compared to state-of-the-art competing methods.
    - On a set of standard classification datasets - MNIST and CIFAR10, we show that **GDSolver** is able to achieve the highest accuracy against all competing methods, using only 50% the training data than competing GD baselines required for the same.

[2]Code available at: https://dhananjayashok.github.io/Hybrid-Solver-NN-Training/

Algorithm 1 The GDSolver Algorithm

Input: Untrained DNN, Training Data, Validation Data
Parameter: Desired Loss, MaxIter
Output: Trained DNN

```
1: i := 0
2: while Validation Loss is Decreasing do
3:     Train DNN using Gradient Descent
4:     Measure Validation Loss
5: end while
6: Convert final layer to an MILP Instance
7: Solve MILP Instance
8: Map Solution of MILP Instance back to NN parameters
9: i++
10: if Validation Loss > Desired Loss and i < MaxIter then
11:     Go to Line 2
12: else
13:     return trained DNN
14: end if
```

2 GDSolver: The Architecture of Gradient Descent + Solver DNN Training Method

In this section, we detail the steps outlined in the architecture diagram in Figure 1 and Algorithm 1 detailed above. The first step of **GDSolver** is to train the network with GD alone, as one would for any DNN [lines 1-3], until the plateauing of validation loss, indicative of a local minimum, is observed [line 1]. At which point, **GDSolver** halts the GD training and proceeds to the second step, namely, solver-based training. The value of using GD to train networks is well known, namely, scalability to very large networks and the ability of GD-based methods in obtaining low loss in many settings.

In the solver phase [lines 5-7], the **GDSolver** algorithm takes the partially trained network and focuses on "Fine Tuning" the final layer using an MILP solver. In this step, **GDSolver** first converts the problem of training the final layer of the neural network to an MILP instance using a specialized formulation [line 5] (discussed in greater detail in Subsection 4 below). The idea here is to search in a region around the values assigned by GD to the network's final layer weights and biases, such that the resultant assignment found by the solver has even lower loss than the one found by GD alone in step 2 (assuming such a lower loss point exists). If no lower loss point is found, **GDSolver** stops training and returns the trained DNN [line 11].

The MILP instance thus formulated is solved by an MILP solver [line 6] (specifically, the Gurobi solver), and the solution is then mapped back to the network weights and biases [line 7]. We refer to this process as *final layer fine-tuning*. The termination condition for the training loop is a check that ascertains whether the desired accuracy has been achieved or further improvements to the weights and biases are possible. If yes, then the loop continues, else it terminates [lines 8-12].

Regression MILP Formulation

$$w_{j,i} \in \mathbb{R} \bigcap \mathbf{w}[j,i] \pm \mathbf{r}, \forall i,j \in [0, N-1] \times [0, M-1] \quad (1)$$

$$b_j \in \mathbb{R}, \forall j \in [0, M-1] \quad (2)$$

$$a_{t,j} = \sum_i w_{i,j} h_{t,i} + b_j, \forall t,j \in [0,T] \times [0, M-1] \quad (3)$$

$$\text{relu} \Rightarrow o_{t,j} = \max(a_{t,j}, 0), \forall t,j \in [0, T-1] \times [0, M-1] \quad (4)$$

$$\neg\text{relu} \Rightarrow o_{t,j} = a_{t,j}, \forall n, j \in [0, T-1] \times [0, M-1] \quad (5)$$

$$|o_{t,j} - y_{t,j}| < \text{max\_loss}_{t,j} \forall t,j \in [0, T-1] \times [0, M-1] \quad (6)$$

Figure 2: Regression Formulation

Classification MILP Formulation

$$\max(\sum_{t=1}^{T} c_t) \quad (7)$$

$$w_{j,i} \in \mathbb{R} \bigcap \mathbf{w}[j,i] \pm \mathbf{r}, \forall i,j \in [0, N-1] \times [0, M-1] \quad (8)$$

$$b_j \in \mathbb{R}, \forall j \in [0, M-1] \quad (9)$$

$$c_t \in \{0,1\} \forall t \in [0, T-1] \quad (10)$$

$$s_{t,j} \in \{0,1\} \forall t \in [0, T-1] \forall j \in [0, M-1] \quad (11)$$

$$a_{t,j} = \sum_i w_{i,j} h_{t,i} + b_j, \forall t,j \in [0,T] \times [0, M-1] \quad (12)$$

$$\text{relu} \Rightarrow o_{t,j} = \max(a_{t,j}, 0), \forall t,j \in [0, T-1] \times [0, M-1] \quad (13)$$

$$\neg\text{relu} \Rightarrow o_{t,j} = a_{t,j}, \forall n, j \in [0, T-1] \times [0, M-1] \quad (14)$$

$$(s_{t,j} = 1) \Rightarrow o_{t,\text{argmax}(y_t)} > o_{t,j}, \forall t, \forall j \neq \text{argmax}(y_t) \quad (15)$$

$$(c_t = 1) \Rightarrow \sum_{j \neq \text{argmax}(y_t)} s_{t,j} = M - 1 \quad (16)$$

Figure 3: Classification Formulation

2.1 Motivation and Advantages of a Hybrid Solver+GD Training

Since the MILP solver is only involved in fine tuning, and does not train the entire network end-to-end, it is completely invariant to the architecture and size of the Neural Network until the final layer. This hybrid approach makes **GDSolver** much more scalable than prior methods for training neural networks using solvers alone [Icarte *et al.*, 2019]. Further, the specific design choices we have made enables our method **GDSolver** to be very general, i.e., handle a variety of architectures, since a large variety of DNN architectures can be symbolically modelled as MILP problems. At the same time, our method **GDSolver** retains the ability to be highly influential on the final prediction strength of the DNN, as shown in our experiments.

The idea of fine tuning the final layer(s) alone is not new, and other methods have been proposed with dramatic and highly consequential impact on network performance [Howard and Ruder, 2018; Pan and Yang, 2009]. To the extent we know, our work is unique in that we use a MILP solver for final layer fine tuning.

3 Escaping Local Minima via GDSolver vs. Competing Methods

Escaping Local Minima via Solvers: As has been noted, GD performs well until it gets stuck in a local minima, and limited options exist to escape local minima. By contrast, the solver performs exhaustive search in a large space around the weights and biases assigned by GD in the previous iteration, and thus may discover a new point with lower loss. By assigning this new point (i.e., new weights and biases) to the DNN, it may be able to escape the local minima by tunnelling through it and with the help of an additional iteration of GD more effectively than only using GD.

Adam Optimizer and LRS: Current alternatives for handling local minima include using momentum based optimization methods (Adam [Kingma and Ba, 2014], RMSProp [Kurbiel and Khaleghian, 2017] etc.) and LRS [Li and Arora, 2019]. Momentum based optimization methods struggle in several instances due to the fact that they too use gradient information to decide the size of steps taken in training. While they are very good at finding the right step size to take, they are not so useful if the direction towards the better solution is currently unknown or not discoverable given local gradient information. Learning Rate Scheduling methods, while simple and efficient , are often highly dependent on hyper-parameter tuning and thus can be unreliable.

The strength of the **GDSolver** Algorithm is that it can be used alongside all of these above methods, and offers another route to escaping local minima if they perform poorly in particular settings. In the rest of this work, we focus on showing that our hybrid method is uniquely useful in efficient training on widely useful metrics of efficiency and generalization.

4 The MILP Formulation

Key to the success of our **GDSolver** method is a symbolic formulation of the final layer of a Neural Network as an instance of the MILP problem, and then mapping the solution obtained by invoking an MILP solver back to the parameters of the final layer. Put differently, we convert the final layer of the DNN into a mathematical formula as described below. For our base formulation we use a variation of that used in [Icarte *et al.*, 2019], with significant improvements as discussed here. The full formulation is presented in Figures [2] and [3].

Similar to previous work in symbolic formulations of DNNs [Bunel *et al.*, 2017; Cheng *et al.*, 2017], we restrict our system to using only linear piece-wise or the soft-max activation functions for the final layer. In the formulation shown in the Figures [2] and [3], we restrict ourselves to the constraint for the ReLU activation function as this is most commonly used one. Having said that, our formulation can easily handle any linear piece-wise activation.

4.1 Setup and Definitions

Let f denote a partially trained Neural Network and L its final layer, and $\{X, Y\}$ denote a dataset with T datapoints. Let N denote the input dimension of the final layer L and M be the dimension of the output. Then, $L : \mathbb{R}^{N \times 1} \rightarrow \mathbb{R}^{M \times 1}$: is a map that can be written out using a weight matrix $\mathbf{w} \in$

Equation ID:	GD(10)		GD(20)		**GDSolver**	
Equation	MSE	Time	MSE	Time	**MSE**	**Time**
Identity	0.579	0.0353	0.2412	0.084	**0.109**	**0.043**
Affine	16.467	0.0278	8.075	0.071	**7.2095**	**0.0321**
Polynomial	93.86	0.0324	20.805	0.076	**12.07024**	**0.0387**
Formula	10.44	0.0361	4.208	0.0864	**3.32117**	**0.0452**

Table 1: **Regression Experiment Results:** Values for best GD baseline (SGD with LRS) GD benchmarked at halfway point (10) and final epoch (20). Results show GDSolver after 10 epochs of GD outperforms 20 epochs of GD for both MSE and Time

$\mathbb{R}^{M\times N}$, a bias vector $b \in \mathbb{R}^{M\times 1}$, input $h \in \mathbb{R}^{N\times 1}$ and activation function σ as follows: $L(h) = \sigma(\mathbf{w}h + b)$. We can express f as: $f = L \circ f'$ where f' is all the previous layers of the DNN other than the final layer L. Then, the goal of DNN training is to learn the mapping $L \circ f'(X) = y \Leftrightarrow \sigma(\mathbf{w}h + b) = y$, where $h = f'(X)$.

4.2 Base Formulation for Regression and Classification

When converting this final layer L to an MILP instance, all the weights and biases of the final layer are represented as variables $w_{i,j}$ and b_j. Constraints (1,2) in Figure 2 (respectively, constraints (8,9) in Figure 3) are box constraints that bound the region around the value assigned to the variables $w_{i,j}$ and b_j that the solver is required to search.

Constraints (3, 4, 5, 12, 13, 14) as given in Figures[2, 3], essentially encode the architecture of the neural network. More precisely, for every data point (x_t, y_t) we compute $h_t = f'(x_t)$, where $h_t \in H$ is the input to the final layer. Then, constraint (3) (respectively, constraint (12)) encodes the input to the activation function as a linear combination of h_t and the parameters of the final layer, while the constraints (4, 5, 13, 14) encode the activation with ReLU. Finally, each training data point also has constraints to relate the output of the neural network to the intended target label/ value y_t. The encoding of the target label depends on whether the problem is one of regression or classification.

4.3 Formulation of DNN Output

Regression Output: In the regression formulation, constraint (6) bounds the L1 distance of the output and target by a constant value $\text{max\_loss}_{t,j}$. In practice we set $\text{max\_loss}_{t,j} = L1(o, y)_{t,j}$, i.e., the L1 loss of that data point using the current weight assignment of the network. This ensures that if a solution is found by the solver, then it has a better L1 loss on the training dataset than the current assignment given by the previous GD step of the **GDSolver** algorithm.

Classification Output: The output dimension of classification models is typically equal to the number of classes that could be predicted, and the prediction of the neural network is the class which corresponds to the highest output neuron value in the final layer for a given data point. With this in mind, constraints (15, 16) encode whether a given data point is correctly classified by the DNN, i.e., the variable c_t is 1 iff datapoint t is correctly classified, $\sum_t c_t$ is hence a measure of the total accuracy of the network. We set this accuracy as the maximization objective of the solver constraint (7), setting a lower bound on the accuracy as the current accuracy of the model given by the prior GD step of **GDSolver** tool ensures that any solution found will have a better accuracy on the training set than the current assignment.

4.4 Mapping Solutions to the DNN

Given this formulation, it is fairly straightforward to convert the final layer of the given network to a valid MILP problem and query a solver for satisfying assignments for the weights and biases. If a feasible solution is found we simply assign $W[i, j] = w_{i,j} \forall i, j$ and $b[j] = b_j \forall j$.

4.5 Discussion on Formulation

There are several key differences between our formulation and the original used in [Icarte *et al.*, 2019], that enable us to train networks faster and with less over-fitting.

Local Neighbourhood Restrictions: The first is in how we define the weight and bias variables in constraints (1, 2, 8, 9) - we ensure that these variables can only be set to an interval around the current weight and bias assignments given by the prior GD step of the **GDSolver** algorithm. This restricts the space of assignments that the solver has to search over, vastly improving its scalability. It has the additional advantage to preventing over-fitting, as as new solutions cannot be too "different" from the current assignment. Finally, in practice, this restriction does not seem to impede the solver from tunnelling through the local minima.

Regression Flexibility: Instead of regression constraint (6) and classification (7, 15, 16), previous formulations relate the output of the neural network to the intended target label with the constraint $o_{j,t} = y_{j,t} \forall j, \forall t$ [Icarte *et al.*, 2019; Thorbjarnarson and Yorke-Smith, 2020]. While this is more straightforward, it has significant flaws. This forces the network to search for assignments that perfectly regress every single training point which is highly likely to cause overfitting. Our alternative of constraint (6) acknowledges that perfect accuracy on the training set is undesirable, striking a better balance by simply requiring a lower loss than the current assignment. This observation was absolutely key to the success of our formulation and DNN training tool.

Classification Flexibility: The problem with previous formulations of the constraints that relate the network output to labels is even more pronounced in the classification domain, where the constraint not only requires the solver to achieve perfect accuracy on the training set, but also requires the output vectors to match [Icarte *et al.*, 2019; Thorbjarnarson and Yorke-Smith, 2020]. The output vector for classification problems are often vectors of binary indicators of class membership for each class. For example, given

a 3 class classification problem target vector $y_{3\text{ class}} \in \mathbb{R}^{T\times 3}$ where $y_{i,3\text{ class}} = [1,0,0]$ would mean that the ith data point is of the first class. In the vast majority of cases Neural Networks output vectors do not attempt to predict the exact 0-1 value, but rather predict un-normalized probabilities, such that the final prediction is the class with the highest output value in the predicted vector. Thus if $\text{pred}_i = [0.75, 0.2, 0.05]$ or $\text{pred}_i = [5,1,3]$ the DNN has correctly predicted the output label, but the previous MILP formulations would consider all of these to be incorrect as they do not match the exact vector $[1,0,0]$. Our formulation has constraints (7, 15, 16) which allow the model to fall short of perfect training accuracy and permits the DNN to predict un-normalized probabilities in its final layer.

Generalization of previous formulations: The strength of our novel formulation is that it is a generalization of the ones that came before, i.e., the previous formulations are a special case of ours that is obtained by setting the max_loss and minimum accuracy parameters to 0 and 1 respectively.

5 Experimental Evaluation

Experimental Setup: For all our experiments, we compare our hybrid method against the four GD baselines of SGD, SGD with LRS, Adam Optimization and Adam with LRS. The experiments were run on a system with the following specs: 18.04.2-Ubuntu with Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz. Models were created and trained with standard PyTorch implementations of GD baselines, the datasets for MNIST and CIFAR10 were the standard datasets provided by PyTorch [Paszke *et al.*, 2019]. The MILP solver used was the python interface of the Gurobi solver - Gurobipy [Pedroso, 2011].

5.1 Experiment 1: Regression

In this experiment, our goal was to ascertain whether the **GDSolver** algorithm and tool achieves faster convergence with greater data and resource efficiency than GD baselines. In order to make the comparison as fair as possible, we use regression datasets (namely, identity, affine, polynomial of degree 4, and trignometric and exponential formula. See Appendix for more details) that we know the baseline models can accurately predict with a low loss.

The experiments were performed as follows: we vary the number of epochs e (from 1 to 20) and for each GD baseline note the testing loss and time taken after e epochs. We then compare this to the testing loss and time taken to complete e epochs of SGD and a single solver sweep of the final layer. We expect to see a strict increase in time taken, as the hybrid method does all the iterations that the baselines do and an additional step, however if the improvement in loss is sufficiently large, then it would justify the additional time cost. This also allows us to quantify how many additional epochs of GD would have been required to achieve equivalent loss.

Analysis of Results: The results for the first experiment can be seen in Table 1. For brevity the results shown are for the best GD baseline (SGD with LR Scheduling) at the median epoch and maximum epochs used - 10 and 20. (Figures showing complete results for all epochs can be found in the Appendix). The table compares the GD baseline after 10 and 20 epochs to **GDSolver** after 10 epochs. The results suggest that the hybrid solver method is very useful in quickening the rate of convergence of loss - For each of these datasets, after 10 epochs, the hybrid method outperforms the other baselines with respect to generalization, and in most cases only after more than 20 epochs would the baselines catch up to the hybrid solvers generalization loss. The time taken by **GDSolver** is greater than the baselines at 10 epochs, but significantly less than the time taken for 20 iterations, which is how long the baselines take to achieve a comparable loss - **GDSolver** produced models with, on average, 31.5% lower MSE in 48% less time. These two observations put together motivated the conclusion that the hybrid solver method is an efficient and valuable method to quicken the rate of convergence, outperforming classic GD approaches.

5.2 Experiment 2: Classification

In this experiment we ascertain whether our Solver+GD hybrid approach consistently performs better than GD baselines in terms of generalization to a test set as the amount of training data is varied.

We perform the experiment by varying the number of training datapoints n and epochs e. For each pair of these variables (n, e) we train the baseline GD methods with n points for e epochs. For comparison, using the **GDSolver** algorithm we train for a maximum of $\frac{e}{2}$ epochs of SGD (n datapoints), stopping early and calling the solver if we detect a loss plateau - we do this process 2 times. This 2 Loop **GDSolver** method uses a **maximum** of e epochs in its GD steps, hence making sure that any improved performance is not a consequence of simply more computation. When calling the solver, we do not give it the entire training dataset, but rather a single batch (32 datapoints) of datapoints which are incorrectly classified by the current assignment given by GD. We compare the GD baselines with the 2 Loop **GDSolver** on test accuracy that measures generalization. We perform the above experiment on two well-known datasets - MNIST, CIFAR10.

Analysis of Results: The results for the second experiment are shown in Figures [4] and [5]. Results show that on average the hybrid method outperforms all GD baselines in terms of testing accuracy. The trend lines give deeper insight into the advantage that the hybrid method provides. It shows that consistently and significantly the hybrid method performs much better when fewer data points are used, i.e., the hybrid method is more data efficient. For both the datasets we can observe a trend where as training data increases the baseline methods eventually achieve similar performance to our method - **GDSolver** on MNIST and CIFAR10 achieves better accuracy than GD baselines with only 50% of the training data. This phenomenon of solver-based training methods performing better when data is scarce has been noted before. For example, in [Icarte *et al.*, 2019] they showed that for binary neural networks, models trained by solvers with limited data were vastly superior to GD methods. The results are also consistent with the idea that the solver sweep in **GDSolver** tool helps with tunnelling through the local minimum that the GD baselines struggle with, and thus reaching higher accuracy with fewer data points than the GD baselines.

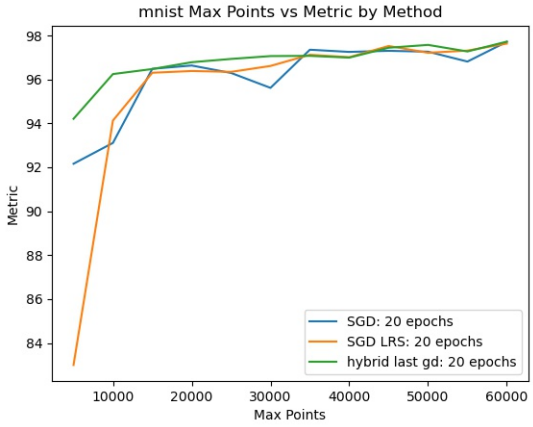

Figure 4: **GDSolver** (green, hybrid last gd) achieves 96% accuracy with half the datapoints that it takes the best GD baseline to

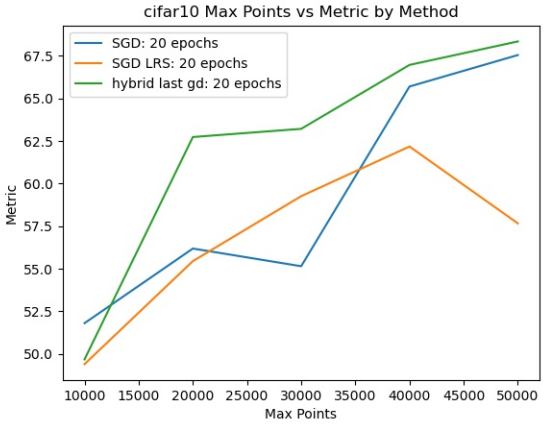

Figure 5: **GDSolver** (green, hybrid last gd) is consistently better than both baselines, especially with less data

Limitations: The **GDSolver** makes a fundamental tradeoff, namely, in order to work with real valued DNNs it can only perform solver sweeps on the final layer of the network, for otherwise the resultant optimization would be a non-linear optimization problem and hence outside the scope of MILP solvers. **GDSolver** also currently assumes that the DNNs it takes as input use a feed-forward densely connected final layer. This assumption is mostly true for regression and classification problems, however does not hold for most specialized networks like Image generating GANs etc.

6 Related Work

Symbolic Formulation of DNNs: There is growing literature in the interpretation of Neural networks symbolically as MILP, SAT, or SMT problems [Tjeng *et al.*, 2017; Zhang *et al.*, 2018; Bunel *et al.*, 2017]. Almost all of this work is aimed towards verification of pretrained neural networks via DNN verification solvers (see VNN-LIB Initiative for more details). This has important implications as it is a fundamentally different formulation to the one we use in our tool, wherein the variables in the symbolic formulation for verification are input data points, while the variables in the context of training are network parameters.

DNN Training via Solvers: Training neural networks using solvers has mainly been studied in the binary and integer settings. Narodytska et al. [Narodytska *et al.*, 2019] studies converting Binary Neural Networks to SAT problems and studied which architectures were more "SAT" friendly so they may be solved efficiently. Icarte et al. [Icarte *et al.*, 2019] established the first MILP formulation for Binary Neural Networks for training purposes. They attempt to deal with problems of over-fitting using regularizing objective functions and show that MILP solvers outperform GD as the training algorithm of choice when there is a sparsity of data points. Thorbjarnarson et al. [Thorbjarnarson and Yorke-Smith, 2020] uses this same formulation and attempts to extend the analysis to integer valued neural networks. However both these methods suffer from scalability, over-fitting and cannot be used on real-valued networks [Icarte *et al.*, 2019].

7 Conclusions and Future Work

We present **GDSolver**, a hybrid solver and GD training algorithm for DNNs, that consistently outperforms 4 state-of-the-art GD methods on several regression and classification tasks in terms of higher accuracy with greater data and resource efficiency. Further, to the best of our knowledge, ours is the first solver-based method that can scale to real-world sized DNNs. MILP Solvers and GD excel in different settings and ways. Our method **GDSolver** leverages the advantages of each method, giving rise to an algorithm that combines the best of both worlds and is better than its individual parts on measures such as accuracy and data efficiency. By using solvers to *tunnel through and escape the local minima*, we tackle one of the most important and difficult problems that GD methods often encounter. This gives **GDSolver** the ability to scale well to additional data and model sizes, but also outperforms pure GD methods in terms of rates of convergence and data efficiency. In the future, we plan to extend our work to handling highly non-linear constraints, since other classes of solvers, such as SMT solvers, are capable of handling such non-linearity. Further, one of our goals is to train DNNs in a manner that ensures (probabilistic) adherence to security or reliability constraints. It is unclear how a purely GD-based method can be used to provide guaranteed adherence to such specifications. By contrast, we believe that solver-based hybrid training methods could enable us to train DNNs in a manner that ensures (probabilistic) adherence to logical specifications.

A Regression Experiment Details

A.1 Equation Definitions

The regression experiments described in section 5.1 used multiple different mathematical equations that are defined in the following section. The names of the below functions matches the equation ID column of table 1:

For all these equations we define $K = X[:, :2]$

Identity:

$$y \in \mathbb{R}^{T\times 2} X \in R^{T\times 6} y = K \tag{1}$$

Affine:

$$y \in \mathbb{R}^{T\times 2} X \in R^{T\times 6} y = 3K + 4 \tag{2}$$

Polynomial:

$$y \in \mathbb{R}^{T\times 2} X \in R^{T\times 6} y = 3K^4 + 6K^3 + 2K^2 + K + 9 \tag{3}$$

Formula:

$$y \in \mathbb{R}^{T\times 2} X \in R^{T\times 6} y = 2e^K + 3sin(K)K^5 \tag{4}$$

A.2 Model Definitions

All the equations used the same model architecture, every layer is a dense, feed forward, linear layer

- Layer 0 (Input, 6 nodes)
- Layer 1 (Hidden Layer, 50 nodes)
- ReLU activation
- Layer 2 (Prediction Layer, 2 nodes)

The experiments were conducted with multiple different model architectures, including DNNs with over 5 hidden layers, however for these simple regression experiments a single hidden layer proved to perform the best and easiest to train with all methods.

A.3 Parameters

The parameters and ranges for all experiments are stated below. The results in table 1 are an average of the parameters marked in **bold**. This is because parameters that are outside usual ranges were also tested and they were excluded as they would bias the results

1. Epochs: [1 - 50]. We observe that performance rarely improves beyond epoch 20 and hence in our experiments set that as the final epoch of meaningful comparison.
2. Learning Rate: [0.00001, 0.0001, 0.0001, **0.001**, **0.01**, **0.1**]
3. weight range for solver formulation constraints (1, 8) 2, 3: [0.001, **0.01, 0.1, 1,** 10]
4. Training data points: [100, **1000, 10000**]

B Classification Experiment Details

B.1 Model Definitions:

The following model was used for the classification experiments that produce figures - 4, 5:

1. Conv2d(3, 16, 1, padding=1)
2. ReLU
3. Conv2d(16, 32, 3, 1, padding=1)
4. Conv2d(32, 64, 3, 1, padding=1)
5. ReLU
6. Linear(4*4*64, 50)
7. Linear(50, 10)

Dropout and Batch Normalization was tested, however these either made the model worse or offered little improvement and hence were not included to keep the model simpler.

B.2 Parameters

The parameters and ranges for all experiments are stated below. The figures 4, 5 show the amount of training data vs accuracy at a fixed number of epochs (20)

1. Epochs: [10 - 50]. We use 20 in our figures as 20, 30, 40 and 50 epochs all had very similar training accuracies for all methods.
2. Learning Rate: [**0.001**, **0.01**]
3. weight range for solver formulation constraints (1, 8) 2, 3: 0.01
4. Training data points: [**10,000, 20,000, 30,000,** . . ., **Maximum**]

References

[Bunel *et al.*, 2017] Rudy Bunel, Ilker Turkaslan, Philip HS Torr, Pushmeet Kohli, and M Pawan Kumar. A unified view of piecewise linear neural network verification. *arXiv preprint arXiv:1711.00455*, 2017.

[Cheng *et al.*, 2017] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. Maximum resilience of artificial neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 251–268. Springer, 2017.

[Gupte *et al.*, 2013] Akshay Gupte, Shabbir Ahmed, Myun Seok Cheon, and Santanu Dey. Solving mixed integer bilinear problems using milp formulations. *SIAM Journal on Optimization*, 23(2):721–744, 2013.

[Howard and Ruder, 2018] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[Icarte *et al.*, 2019] Rodrigo Toro Icarte, León Illanes, Margarita P Castro, Andre A Cire, Sheila A McIlraith, and J Christopher Beck. Training binarized neural networks using mip and cp. In *International Conference on Principles and Practice of Constraint Programming*, pages 401–417. Springer, 2019.

[Kingma and Ba, 2014] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[Kurbiel and Khaleghian, 2017] Thomas Kurbiel and Shahrzad Khaleghian. Training of deep neural networks based on distance measures using rmsprop. *arXiv preprint arXiv:1708.01911*, 2017.

[Li and Arora, 2019] Zhiyuan Li and Sanjeev Arora. An exponential learning rate schedule for deep learning. *arXiv preprint arXiv:1910.07454*, 2019.

[Narodytska *et al.*, 2019] Nina Narodytska, Hongce Zhang, Aarti Gupta, and Toby Walsh. In search for a sat-friendly binarized neural network architecture. In *International Conference on Learning Representations*, 2019.

[Pan and Yang, 2009] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

[Papernot *et al.*, 2016] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European symposium on security and privacy (EuroS&P)*, pages 372–387. IEEE, 2016.

[Paszke *et al.*, 2019] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[Pedroso, 2011] Joo Pedro Pedroso. Optimization with gurobi and python. *INESC Porto and Universidade do Porto,, Porto, Portugal*, 1, 2011.

[Ruder, 2016] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[Thorbjarnarson and Yorke-Smith, 2020] Tómas Thorbjarnarson and Neil Yorke-Smith. On training neural networks with mixed integer programming. *arXiv preprint arXiv:2009.03825*, 2020.

[Tjeng *et al.*, 2017] Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356*, 2017.

[Verma *et al.*, 2019] Sunny Verma, Chen Wang, Liming Zhu, and Wei Liu. A compliance checking framework for dnn models. In *Twenty-Eighth International Joint Conference on Artificial Intelligence {IJCAI-19}*. International Joint Conferences on Artificial Intelligence Organization, 2019.

[Zhang *et al.*, 2018] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. *arXiv preprint arXiv:1811.00866*, 2018.