Modeling Design and Control Problems Involving Neural Network Surrogates

Dominic Yang · **Prasanna Balaprakash** · **Sven Leyffer**

Abstract We consider nonlinear optimization problems that involve surrogate models represented by neural networks. We demonstrate first how to directly embed neural network evaluation into optimization models, and highlight a difficulty with this approach that can prevent convergence. We then present two alternative formulations of these problems as mixed-integer optimization problems and as optimization problems with complementarity constraints. Each of these formulations may be solved with state-of-the-art optimization methods, and we show how to obtain good initial feasible solutions for these methods. We compare our formulations on three practical applications arising in the design and control of combustion engines, in the generation of adversarial attacks on classifier networks, and in the determination of optimal flows in an oil well network.

1 Introduction and Background

We are interested in solving general optimization problems that include deep neural networks (DNNs) that are used as surrogate models of complex functions (e.g., physical processes [2], classification schemes [24, 28, 43]). In particular, we consider optimization problems of the form

$$\underset{x}{\text{minimize}} \ f(M(x), x) \quad \text{subject to } c(M(x), x) \leq 0, \ x \in X, \tag{1}$$

where f and c are smooth functions representing the objective function and constraints, respectively, $M(x)$ is the prediction of a DNN at x (which we assume to have been previously trained on suitable data), x are the optimization variables, and $X \subset \mathbb{R}^n$ is a compact set that may include integer restrictions. We consider feedforward neural networks in this paper which are composed of a sequence of multiple layers of neurons: the values of which are a nonlinear function (the *activation function*) applied to a linear transformation of the values in the prior layer:

$$x^\ell = \sigma(W^{\ell,T} x^{\ell-1} + b^\ell), \ \ell = 1, \dots, L \tag{2}$$

where x^ℓ are the values of neurons in layer ℓ ($x^0 = x$ and $x^L = M(x)$ above), L is the number of layers, W and b are weights determined by a training procedure, and σ is the activation function applied componentwise.

D. Yang
University of California at Los Angeles
E-mail: domyang@math.ucla.edu

P. Balaprakash
Argonne National Laboratory
E-mail: pbalabra@anl.gov

S. Leyffer
Argonne National Laboratory
E-mail: leyffer@anl.gov

An example of an optimization problem we may wish to solve is to minimize the output of a neural network regressor that predicts the quantity of emissions from automobile engine specifications. In this case, X would be the set of existing automobile specifications, the constraints c would be constraints to ensure the engine is realistic, and the objective would be a function of the emissions or engine performance. We will discuss this problem in detail in Section 2.1.

To arrive at a tractable form of Eq. (1), we make the following assumptions.

Assumption 1 *We assume that the following conditions are satisfied for problem Eq.* (1)*:*

1. *The objective function, $f : \mathbb{R}^n \rightarrow \mathbb{R}$, and the constraint functions, $c : \mathbb{R}^n \rightarrow \mathbb{R}^m$, are twice continuously differentiable* convex *functions.*
2. *The set $X \subset \mathbb{R}^n$ is convex and compact.*

The most restrictive assumption is the convexity assumption. We can relax this assumption by leveraging standard global optimization techniques (see, e.g., [4, 39, 45]), at the expense of making the reformulated problem harder to solve. The smoothness assumption on f and c can be relaxed to Lipschitz continuity by using subgradients, and the compactness assumption is typically satisfied as long as x are constrained by bounds or the constraint function has compact level sets. Below, we consider deep neural networks with ReLU activation functions that satisfy this assumption.

Outline and Contributions.

We start by discussing three applications that employ DNN surrogates within an optimization problem such as Eq. (1), namely (a) the minimization of emissions in an engine design problem (as in [2]), (b) the generation of optimal adversarial examples to "fool" a given classifier (as in [15]), and (c) the optimization of pump configuration for oil (as in [22]). Next, we demonstrate empirically that simply including a ReLU DNN within an optimization problem can lead to poor convergence results. In particular, we have developed a new nonlinear constraint for JuMP [12] that allows us to directly include DNNs within an optimization model specified in JuMP.

Next, we consider alternate formulations and show that DNNs that use purely ReLU activation functions can be formulated as mixed-integer sets, building on [3, 15]. We can then formulate Eq. (1) *equivalently* as a convex mixed-integer nonlinear program (MINLP). We also introduce a new formulation that results in a non-convex nonlinear program (NLP) with complementarity constraints. Our reformulations involve a lifting into a higher-dimensional space in which the MINLP problem is convex. We show empirically that for moderately sized machine learning (ML) models the resulting programs can be solved by using state-of-the-art MINLP and NLP solvers, making optimization problems with machine learning (ML) models tractable in practice. We demonstrate that using the MINLP formulation, we can find provably optimal solutions for small problems. Using the NLP formulation, we can address significantly larger networks at the cost of guaranteeing only locally optimal solutions. We develop a warmstart heuristic for both approaches that generates good initial guesses from the training data and helps us overcome the challenges of the nonconvex formulation. We also show how to add constraints that restrict the optimization problem to search only near where existing training points can be added.

Throughout this paper we assume that the ML model has been trained and is fixed for the optimization, and we do not consider the question of updating the ML model during the optimization loop. One limitation of our approach is that we use standard MINLP solvers to tackle the reformulated MINLP, which limits the size of the ML model, $M(x)$, that can be used in the optimization.

Related Work.

Prior approaches to optimization over neural networks using mixed integer programming (MIP) formulations include [8, 14, 15, 27, 42, 46]. These approaches primarily model the piecewise ReLU constraint using standard big-M modeling tricks. They generally use the same basic formulation, but each augments the solve by adding methods to tighten the big-M constraints [15, 22, 46], decomposing the problem into smaller problems [27], or adding local search routines [14]. [3] provides an in-depth overview of how to strengthen these models to an ideal formulation with exponentially many constraints, as well as a method to separate in linear time.

Some other approaches to these problems using methods from MINLP have been tried. [9] solves an inverse problem over ReLU constraints using an outer approximation inspired method, but without global optimality guarantees. [26] uses an approach from satisfiability modulo theory to address an optimization problem over neural networks. [41] uses an MINLP approach involving McCormick relaxations. These optimization problems focus primarily on testing the resilience of neural networks against adversarial attack [7]. This involves either maximizing a notion of resilience [8] or finding minimal perturbations needed to misclassify an image [15]. Some work has also been done in using optimization to visualize features corresponding to neurons [15] and for the purposes of surrogate optimization in the context of optimizing the production of a set of oil wells [22].

2 Modeling Optimization Applications involving Neural Network Surrogates

In this section, we describe three optimization models that make use of neural-network surrogates and discuss some of the challenges that arise.

2.1 Optimal Design of Combustion Engine

Automobile engine operation is typically modeled by using highly intensive physics-based simulation code, which even on powerful computers can take hours just to model even a short drive. Hence, modeling the evaluation of this simulation code by using a neural network surrogate model can produce significant time savings at the cost of producing slightly less accurate results. A study examining this approach is documented in [2].

Given a trained neural network, we could formulate several optimization problems answering questions related to the operation of an engine on a given commute. A natural problem one might have is that of minimizing emissions over the course of a given drive. The resulting solution would then be both the engine type and the drive style (e.g., RPM at all times in the drive) that produces the most environmentally efficient commute. The following section demonstrates how such a problem can be formulated.

2.1.1 A Simple Engine Design and Control Problem

Suppose we have a trained DNN that predicts engine behavior based on engine specifications and driving parameters, given in Table 1. The DNN is trained by using 64 trips each split into 1,500 observations (25 minutes observed at second intervals).

Input Parameter	Output Parameter
fuel injection (g) /s	nitrogen oxide, NO /s
engine RPM /s	carbon monoxide, CO /s
compression ratio	torque /s

Table 1: Input/output parameters of engine DNN model. Time-dependent parameters are shown with time per second (/s).

The problem we propose to solve is the optimal design and control of an engine for a given 25-minute trip. We will use a prescribed torque profile as a surrogate for the trip characteristics and minimize a weighted sum of NO and CO output. We have design variables: fuel injection, f; engine RPM, r; and compression ratio, c which are input to the DNN. Note that f and r are characteristics that change over time whereas c is an engine parameter which is fixed for the full drive. We also have variables NO and CO which represent nitrogen oxides and carbon monoxide emissions produced by the engine, as well as a variable Torque which indicates engine torque. Each of these quantities are predicted by the neural network for each time interval of the drive. Formally, we state the following optimal design and control problem over the time horizon $t = 1, \ldots, T$ (where T is the number of time intervals):

$$
\begin{array}{lll}
\underset{f,r,c,\text{CO},\text{NO},\text{Torque}}{\text{minimize}} & \sum_{t=1}^{T} (\text{NO}_t + \lambda \text{CO}_t)\Delta t & \text{min. NO and CO emissions} \\
\text{subject to} & \text{Torque}_t \geq \text{PrescribedTorque}_t, \ \forall t = 1, \ldots, T & \text{trip profile} \\
& (\text{NO}_t, \text{CO}_t, \text{Torque}_t) = \text{DNN}(f_t, r_t, c) & \text{DNN constraints} \\
& f_t \in [f_{\min}, f_{\max}], \ r_t \in [r_{\min}, r_{\max}], \ c \in [c_{\min}, c_{\max}] & \text{bounds on controls.}
\end{array} \tag{3}
$$

We are using the 1500 time intervals (corresponding to $T = 1500$ seconds in a 25-minute drive). As written, we will have $2T + 1$ continuous control variables in addition to the $3T$ output variables predicted by the DNN for a total of $5T + 1$ continuous variables in this formulation.

Our model has a separate DNN evaluation for each time interval t, meaning each evaluation engenders different neural network activations. Because of the presence of the design variable c, which is independent of t, this problem does not decompose into individual time steps. One may construct a bilevel optimization problem wherein on the upper level, we decide c and other engine-level variables of interest, and on the lower level we determine the drive-specific variables that change over time by solving T separate optimization problems of much smaller size. This approach will not be addressed in this paper.

2.1.2 Convex-Hull Constraints

In addition to the constraints that encode the evaluation of a neural network, constraints must be added to ensure that the solver does not extrapolate significantly from the training data. Without these constraints, the optimization routine may find that the optimal solution resides in an area for which the neural network has not learned the behavior of the modeled function leading to a solution that, while optimal, does not reflect true function behavior and may be nonsensical. In fact, in an earlier implementation without the convex-hull constraint, we observed that the optimal design was obtained for an engine that produced negative emissions. We rectify this error by introducing constraints that constrain the input data to our model to be within the convex hull of the input training data.

In our experiments, we examined how the optimization models operated with simple box constraints that bounded the input by the extremal values of the training data, as well as with the convex hull constraints. Figure 1 demonstrates the location of fuel mass and RPM for solutions computed for a sample neural network trained on our data set compared with the actual training data. The corresponding solutions are displayed in Figure 2.
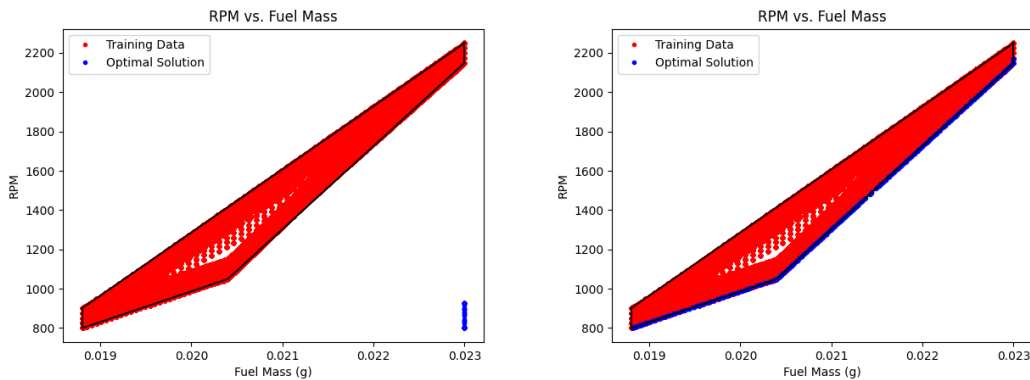

Fig. 1: Location of training data and optimal solution for an engine design problem on a neural network with 1 hidden layer of 16 nodes. On the left, the input is constrained by box constraints and on the right by convex-hull constraints.

Without the convex hull constraints, the optimal solution of neural network evaluation appears to wander off to an area for which there is no training data. This immediately creates problems in the computed solutions, which are obviously nonsensical because they involve negative emissions. This suggests that our neural network does not generalize well to data that has not yet been seen by the network. On the other hand, constraining the input data to be within the convex hull has the effect of producing solutions that look like the training data.
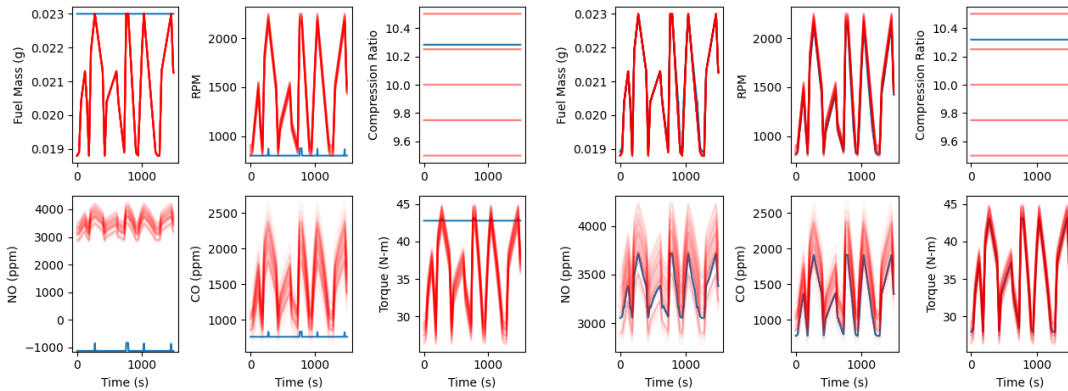

Fig. 2: Optimal configuration for each time step when using box constraints (left) and using convex-hull constraints (right). The red lines indicate the drive cycles used to train the neural network, and the blue lines indicate the optimal solution.

Unfortunately, these constraints have the effect of restricting possible solutions to those that are close to the training data, when the function being modeled may have a minimum that appears far from the training data. This situation speaks to the limitations of using solely a surrogate model for optimization. To find a global optimum of the original function would necessarily involve an alternating approach wherein the surrogate model was optimized and then this solution was queried against the modeled function for new data to be added to the surrogate. This type of approach is addressed in [36].

If we fix the compression ratio, we can plot the contours of the objective function as a function of fuel mass and RPM. Figure 3 shows the contours of the objective function for a 5-layer DNN. We observe that this objective function is highly nonconvex in the reduced space of the original variables but it is convex in the lifted MINLP space according to Proposition 1. We also observe that the convex-hull constraint fulfills a second function by acting as a mild convexifier of the problem by restricting the variables to a small sliver of the feasible set.
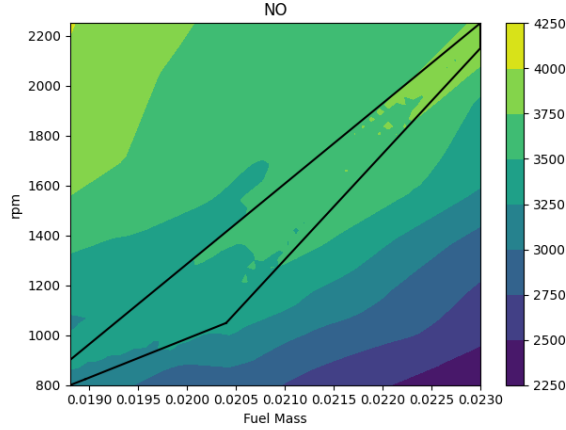

Fig. 3: Contour plot for the NO output of a 5-layer DNN.

2.1.3 Warmstarting the Problem

Seeding a MILP or NLP solver with a feasible solution of high quality can significantly reduce the solve time. For MILP solvers, a feasible solution with sufficiently low value can be used to prune subproblems with higher objective value in a branch-and-bound tree search. For NLP solvers, beginning with a feasible solution avoids the need to search for a feasible solution, and the choice of a good solution can ensure quicker convergence to better locally optimal solutions.

For the optimal engine design problem, we can use our collection of training data to find parameterizations of the engine that exceed the desired torque value but also have a low amount of emissions produced. We do not directly work with the output data from the training set but rather with the output from the neural network applied to the input data since this is what is constrained.

To produce a high-quality solution is then a matter of fixing the parameters that must remain constant for the entire drive (i.e., the compression ratio) and then, from the training data with these fixed parameters, choosing the remaining controls so as to minimize emissions while still exceeding the desired torque for each time step. The remaining variable values correspond to activation of each neuron in the neural network when applied to the input data and hence must be set to exactly those values. This procedure is summarized in Algorithm 1.

Data: Input DNN, compression ratio c_0, training data $\mathcal{X}$, torque profile TorqueProfile_t, T

1   Select subset of training data $\tilde{\mathcal{X}}$ with compression ratio c_0;

2   Let $\tilde{\mathcal{Y}} = \text{DNN}(\tilde{\mathcal{X}})$;

3   **for** $t = 1, \ldots, T$ **do**

4   Let $\text{NO}, \text{CO}, \text{Torque} = \tilde{\mathcal{Y}}$;

5   Let $\mathcal{T}$ be the set of times t where $\text{Torque}_t \geq \text{TorqueProfile}_t$;

6   Let $\tau = \text{argmin}_{t \in \mathcal{T}}(NO_t + \lambda CO_t)$;

7   Let $x_t = \tilde{\mathcal{X}}_\tau$;

8   **end**

9   Set remaining variable values from activation of DNN when applied to x_t for all t.;

Algorithm 1: Algorithm for finding a warmstart solution to Eq. (3)

2.2 Adversarial Attack Generation

Our next problem involves the resilience of neural networks to incorrect classifications. Deep neural networks have the immensely useful property of being able to uniformly approximate any function in a certain general

class, but at the cost of being fairly opaque in terms of how the underlying machinery works. This opacity may mask unpredictable behavior that makes these neural networks susceptible to attacks that disrupt the correct classifications observed on training and testing data. [44] first demonstrated that almost imperceptible perturbations of image data may lead to misclassifications, in essence demonstrating key instabilities in neural networks. This work has led to a number of papers [7, 8, 15] developing algorithms that produce estimates of the resilience of neural networks, estimating how close an incorrectly classified image can be to a correctly classified one.

The central problem of interest as introduced by [44] is described as follows. Given a classifier DNN, an image $x \in [0, 1]^m$, and a desired classification label l, we have the following problem.

$$
\begin{aligned}
& \underset{z}{\text{minimize}} \ \|x - z\| \\
& \text{subject to } \text{DNN}(z) = l \\
& \qquad\qquad z \in [0, 1]^m,
\end{aligned}
\tag{4}
$$

where $\| \cdot \|$ is a suitable norm. Essentially, we ask for an image of minimal distance to image x that is classified with a different label.

We add additional constraints (see, e.g., [15]) to ensure that this is a very confident classification. Specifically, we replace the constraint $\text{DNN}(z) = l$ with a constraint asking that the activation for label l is some factor larger than the activation for each other label; in other words, if y_L is the final layer, we have $y_{L,l} \geq \alpha y_{L,i}$ for $i \neq l$. Often the final layer is given by applying a softmax function, i.e.,

$$
y_{L,i} = \sigma(z_1, \ldots, z_n)_i = \exp(z_i) / \sum_{j=1}^{n} \exp(z_j),
\tag{5}
$$

where the produced values $y_{L,i}$ effectively represent probabilities that the image is a given label i. In this case, we can represent this constraint in terms of z_i as

$$
\sigma(z)_l / \sigma(z)_i \geq \alpha \Leftrightarrow \exp(z_l - z_i) \geq \alpha \Leftrightarrow z_l \geq z_i + \log(\alpha),
$$

which is a linear constraint. Then if we write $y_L = \text{DNN}(z)$ as the final layer prior to the softmax layer, we will have the following problem:

$$
\begin{aligned}
& \underset{z}{\text{minimize}} \ \|x - z\| \\
& \text{subject to } y_L = \text{DNN}(z) \\
& \qquad\qquad y_{L,l} \geq y_{L,i} + \log(\alpha), \qquad i \neq l \\
& \qquad\qquad z \in [0, 1]^m.
\end{aligned}
\tag{6}
$$

We observe that the formulation in Eq. (6) is easily extendable and that constraints on the allowable perturbations can augment the formulation. For example, we may restrict the magnitude of the perturbation to any given pixel by restricting $|x_i - z_i| \leq \epsilon$ for all i. Alternatively, we may be interested in continuous perturbations and therefore restrict $|(x_i - z_i) - (x_j - z_j)| < \epsilon$ for all pixels j adjacent to pixel i. Constraints of this sort have been used to extend adversarial attack optimization problems in [15].

2.3 Surrogate Modeling of Oil Well Networks

The next example of an optimization problem with an embedded neural network involves the operation of an offshore oil platform and is taken from [22]. The full optimization problem is reproduced in Eqns. (7) and the associated sets are given in Table 2.

In this problem, we have a network comprised of three sets of nodes: the wells N^w which are the sources of oil, water, and gas, the manifolds N^m which are connected to wells and mix incoming flows of oil, water, and gas, and the separators N^s which are the sinks of all the flow. Each well is connected to each manifold by a pipeline in E^d which may be turned on and off. Each manifold is then connected to a unique separator by a riser in E^r. A depiction of an example network with 8 wells, 2 manifolds, and 2 separators taken from [22] is presented in Figure 4.

Set	Description
N	Set of nodes in the network.
N^w	Subset of well nodes in network.
N^m	Subset of manifold nodes in the network.
N^s	Subset of separator nodes in the network.
E	Set of edges in the network.
E^d	Subset of discrete edges that can be turned on or off.
E^r	Subset of riser edges.
E_i^{in}	Subset of edges entering node i.
E_i^{out}	Subset of edges leaving node i.
C	Oil, gas, and water.

Table 2: Sets used in Oil Well Optimization Problem

$$\underset{y,q,p,\Delta p}{\text{maximize}} \sum_{e \in E^r} q_{e,\text{oil}} \tag{7a}$$

$$\text{subject to} \sum_{e \in E_i^{\text{in}}} q_{e,c} = \sum_{e \in E_i^{\text{out}}} q_{e,c}, \ \forall c \in C, i \in N^m \tag{7b}$$

$$p_j = \text{DNN}_e(q_{e,\text{oil}}, q_{e,\text{gas}}, q_{e,\text{wat}}, p_i), \qquad \forall e = (i,j) \in E^r \tag{7c}$$

$$\Delta p_e = p_i - p_j, \qquad \forall e = (i,j) \in E^r \tag{7d}$$

$$-M_e(1 - y_e) \le p_i - p_j - \Delta p_e, \qquad \forall e = (i,j) \in E^d \tag{7e}$$

$$p_i - p_j - \Delta p_e \le M_e(1 - y_e), \qquad \forall e = (i,j) \in E^d \tag{7f}$$

$$\sum_{e \in E_i^{out}} y_e \le 1, \qquad \forall i \in N^w \tag{7g}$$

$$y_e q_{e,c}^L \le q_{e,c} \le y_e q_{e,c}^U \qquad \forall c \in C, e \in E^d \tag{7h}$$

$$p_i^L \le p_i \le p_i^U, \qquad \forall i \in N \tag{7i}$$

$$\sum_{e \in E_i^{\text{out}}} q_{e,\text{oil}} = \text{DNN}_i(p_i), \qquad \forall i \in N^w \tag{7j}$$

$$\sum_{e \in E_i^{\text{out}}} q_{e,\text{gas}} = c_{e,\text{gor}} \sum_{e \in E_i^{\text{out}}} q_{e,\text{oil}}, \qquad \forall i \in N^w \tag{7k}$$

$$\sum_{e \in E_i^{\text{out}}} q_{e,\text{wat}} = c_{e,\text{wor}} \sum_{e \in E_i^{\text{out}}} q_{e,\text{oil}}, \qquad \forall i \in N^w \tag{7l}$$

$$p_i = p_i^s = \text{const.}, \qquad \forall i \in N^w \tag{7m}$$

$$y_e \in \{0,1\}, \forall e \in E^d \tag{7n}$$

Our goal in this problem is then to optimize the flow rate of oil to each of the sinks. There are a variety of physical and logical constraints to encode the operation of this flow network. Eq. (7b) ensures that the flow into each node matches the flow out of each node. Eq. (7e) and Eq. (7f) ensure that if a pipeline is open, then the difference in pressure between the two nodes is actually represented by Δp_e. Eq. (7g) ensures that each well only routes its flow to a single manifold. Eq. (7h) bounds the flow rate of each material when pipelines are active and forces the rate to zero when inactive whereas Eq. (7i) bounds the pressure at each node. Eq. (7k) and Eq. (7l) establish the expected ratio of flow rates of each material.

The neural networks DNN_e and DNN_i appear in the problem in Eq. (7c) and Eq. (7j) and represent nonlinear functions which predict the separator pressure and outgoing oil flow rate based on incoming flow rate and pressure, respectively. There is one neural network DNN_e for each riser edge e, and one neural network DNN_i for each well i. Each is trained separately and for our particular configuration of the network, we have 2 risers and 8 wells for a total of 10 separate neural networks which are encoded into our problem.

The authors in [22] considered two different configurations of neural networks for this problem: a "shallow" network with few layers of many nodes and a "deep" network with many layers of fewer nodes. The shallow configuration had two hidden layers of 20 nodes for the well networks f_i and two hidden layers of 50 nodes for
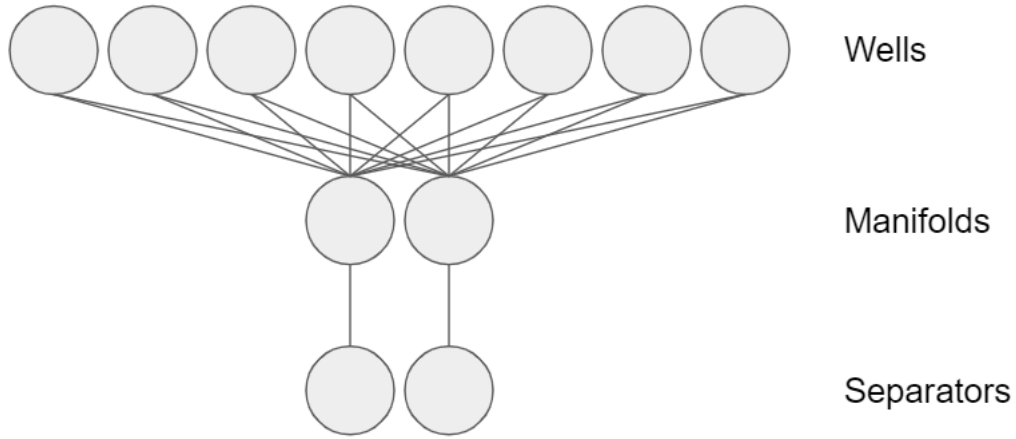
Fig. 4: Example oil well network with 8 wells, 2 manifolds and 2 separators from [22].

the riser networks g_e. The deep configurations have four hidden layers of 10 nodes for the well networks and 5 layers of 20 nodes for the riser networks. All in all, this corresponds to 520 ReLU nodes for each configuration.

3 Embedded Neural Network Formulation

Given that many deep learning libraries (for example TensorFlow [1] and PyTorch [34]) have well-developed builtin automatic differentiation capabilities, it is natural see if we can directly embed the evaluation of the neural network into an NLP:

$$\underset{x}{\text{minimize}}\ f(M(x), x) \quad \text{subject to } c(M(x), x) = 0,\ x \in X, \tag{8}$$

Unlike the formulations in Section 4, this formulation has the advantage of not requiring auxiliary variables for each of the internal nodes of the neural network meaning that this formulation should scale significantly better as the number of nodes in the neural network increases.

The modeling language JuMP [13] which supports formulations of optimization problems in the programming language Julia provides the ability to handle nonlinear functions with user-provided gradients. Using this framework, we can directly embed neural network evaluation which JuMP will then communicate to solvers.

As of the current release, JuMP only supports univariate nonlinear functions, but we can represent vector-valued functions by listing each output component separately. A single nonlinear constraint representing $y_i = M(x)_i$ can be written as

```
register(model, DNN_i, n, DNN_i, DNN_i_prime)))
@NLconstraint(model, y[i] == DNN_i(x...))))
```

where `model` is the JuMP optimization model, `n` is the dimension of the input, and `DNN_i` and `DNN_i_prime` represent function evaluation of the neural network's ith output neuron and its gradient which can be provided by TensorFlow.

For networks with many outputs, explicitly listing these commands this can quickly become unwieldy, but we can enumerate these constraints using macros. The following code demonstrates how one can encode $y = M(x)$ where $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$, and `f` and `f_p` are evaluation of the neural network and the Jacobian respectively.

```
macro DNNConstraints_grad(model, x, y, n, m, f, f_p)
  ex = Expr(:block)
  for i = 1:m
  dnn = gensym("DNN")
  push!(ex.args, :($dnn_s(z...) = $f(z...)[$i]))
    p = gensym("DNN_prime")
    push!(ex.args,:($prime(g, z...) = begin g .= $f_p(z...)[$i,:] end))
    push!(ex.args,:(register($model, $(QuoteNode(dnn_s)), $n, $dnn, $p)))
    push!(ex.args,:(@NLconstraint($model, $y[$i] == $dnn($x...)))))
  end
```
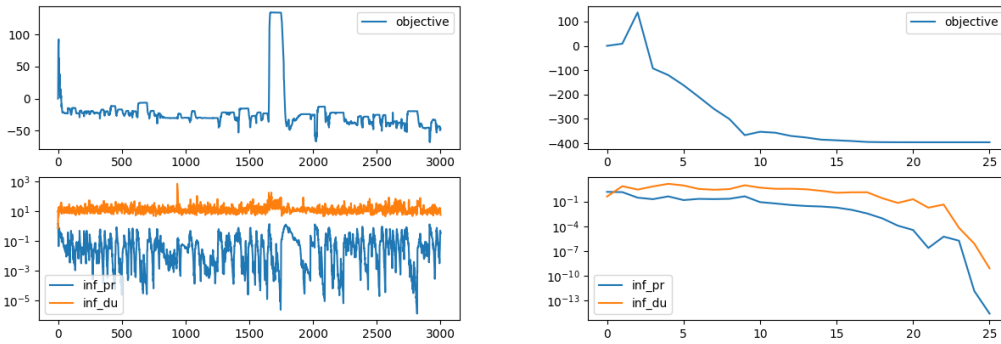
Fig. 5: Objective value, primal infeasibility and dual infeasibility plotted against iteration number for a sample ReLU network (left) and a sample swish network (right) over an Ipopt solve.

```
    ex
end
```

This macro can then be called to add constraints on a neural network which takes in values from $\mathbb{R}^{20}$ and outputs values in $\mathbb{R}^{10}$ using the following command,

```
@DNNConstraints_grad(model, x, y, 20, 10, DNN, DNN_prime)
```

where `DNN` and `DNN_prime` are now vector and matrix valued functions which return the output vector and Jacobian for neural network evaluation respectively.

With this kind of formulation, we are able to take any trained neural network and represent them naturally in JuMP which can then communicate neural network activity to solvers.

3.1 Convergence Behavior

As we are considering neural networks which make use of the widely popular ReLU activation function,

$$\text{ReLU}(a) = \max(a, 0). \tag{9}$$

This popular choice of activation function results in nonconvex constraints and objective in Eq. (1). In addition, the constraints are nonsmooth whenever $a = 0$.

The nonsmoothness in Eq. (9) has generally been considered to be of little concern as in practice neural networks involving ReLU neurons have found great success especially in problems of classification [24, 28, 43] so that they have become an industry standard in deep learning. Part of their success is attributed to their ease in implementation as well as their tendency to produce sparse activation patterns and avoid the vanishing gradient problem in training [20]. There have also been some theoretical results which affirm some convergence guarantees under mild conditions [11, 32] and [21] demonstrates qualitatively that the training problem often experiences few of the issues common in nonconvex and nonsmooth optimization.

These results pertain primarily to the optimization problems solved in the training process, and we have observed relatively little work in the literature regarding optimization problems involving trained networks. Our preliminary experiments suggest that in this context, the expectation of good behavior may be unfounded. Initial solves of Problem (3) using the state-of-the-art NLP solver Ipopt [47] experienced serious issues with convergence. Figure 5 illustrates the objective value and primal and dual infeasibility for one example solve of the problem. Qualitatively, we observe that instead of terminating, the solver bounces about among objective values of about the same magnitude. The reason for the lack of convergence can be explained by the fact that one measure of convergence, the dual infeasibility, remains high throughout the duration of the solve. This type of behavior makes this formulation difficult for general use as there are not clear conditions in general for checking if the current solution at hand is "good." We suspect that these convergence issues can arise in part due to the non-differentiability of the ReLU neurons. Classical optimization algorithms such as quasi-Newton algorithms [35] often have baked in the assumption of differentiability and convergence results typically depend on them.

An alternate type of activation function which has seen some success in classifier networks [38] is the swish function which is defined as follows:

$$\text{swish}(a) = \frac{a}{1 + e^{-\beta a}} \tag{10}$$

9

β is a hyperparameter which may be learned but is typically set to 1. Note that for $\beta = 0$, this is simply a linear function and for $\beta \to \infty$, this approaches the ReLU function. With the activation function in Eq. (10), all functions in Eq. (1) are twice continuously differentiable under Assumption 1.

To ensure that failure converge when using the ReLU network is not due to choice of NLP solver, We perform the same solve of Problem (3) using instead swish neurons and plot the objective and infeasibilities in Figure 5. Instead of failing to terminate after 3000 iterations, Ipopt converges successfully after only 25 iterations and both primal infeasibility and dual infeasibility converge rapidly to zero. As a differentiable network, the swish formulation does not express the convergence issues that the nondifferentiable ReLU networks appear to exhibit.

These initial experiments results confirm the observation in [5, Fig. 1] that showed that ReLU networks can be *nondifferentiable almost everywhere* for certain types of networks. This observation makes the use of ReLU networks questionable as embedded constraints within optimization solvers that rely on differentiable problem functions. Hence, in the next section, we develop alternative formulations that have better numerical properties.

4 Formulating DNNs as Optimization Models

We provide two alternative formulations of ReLU DNNs in terms of optimization models that avoid the pitfalls of the embedded formulation of the previous section. The first formulation uses binary variables to model the max-functions, resulting in a mixed-integer program (MIP), building on [3, 15, 22]. Our formulation differs from [9], which only considered inverse problems, and did not exploit the convex structure of the DNNs that arises when we lift the DNN constraint, resulting in a nonconvex mode. The second formulation uses complementarity constraints that can be solved as systems of nonlinear inequalities. We derive theoretical properties of both formulations.

4.1 Formulating DNNs with Mixed-Integer Sets

In this section we show how general optimization problems involving DNNs, such as Eq. (1), can be equivalently formulated as convex MIPs, extending [15].

We assume that the DNN is a deep neural network with ReLU activation functions, and we rewrite the non-convex problem Eq. (1) as a constrained problem:

$$\underset{x, y_L}{\text{minimize}} \ f(x, y_L) \quad \text{subject to } y_L = M(x), \ c(x, y_L) \le 0, \ x \in X, \tag{11}$$

where L is the number of layers of the DNN. [15] have shown that the nonconvex constraint, $y_L = M(x)$, can be formulated as a mixed-integer linear set in the case of DNNs with ReLU activation functions. We let $w_{l,i}$ denote the weights of neuron $i = 1, \ldots, N_l$ at level $l = 1, \ldots, L$, and $b_{l,i}$ its corresponding offset. Then, the levels $l = 1, \ldots, L$ are computed as

$$y_{l,i} = \text{ReLU}\left(w_{l,i}^T y_{l-1} + b_{l,i}\right) = \max\left(0, w_{l,i}^T y_{l-1} + b_{l,i}\right). \tag{12}$$

We lift this constraint by introducing slack variables, $s_{l,i}$, and binary variables, $z_{l,i} \in \{0, 1\}$, and observe that Eq. (12) is equivalent to the mixed-integer linear constraints

$$y_{l,i} - s_{k,l} = w_{l,i}^T y_{l-1} + b_{l,i}, \ 0 \le y_{l,i} \le M_{l,i}^y(1 - z_{l,i}), \ 0 \le s_{l,i} \le M_{l,i}^s z_{l,i}, \ z_{l,i} \in \{0, 1\}, \tag{13}$$

where $M_{l,i}^y, M_{l,i}^s > 0$ are sufficiently large constants (if $z_{l,i} = 1$, we are on the 0-branch of ReLU, and if $z_{l,i} = 0$, we are on the positive branch). By substituting Eq. (13) into Eq. (11) we obtain a convex MINLP that is equivalent to the problem Eq. (1):

$$
\begin{aligned}
&\underset{x, y, s, z}{\text{minimize}} \ f(x, y_L) \\
&\text{subject to } c(x, y_L) \le 0 \\
&\quad\quad y_{l,i} - s_{k,l} = w_{l,i}^T y_{l-1} + b_{l,i}, \quad\quad\quad\quad l = 1, \ldots, L \\
&\quad\quad 0 \le y_{l,i} \le M_{l,i}^y(1 - z_{l,i}), \ 0 \le s_{l,i} \le M_{l,i}^s z_{l,i}, i = 1, \ldots, N_l, l = 1, \ldots, L \\
&\quad\quad y_0 = x, \ z_{l,i} \in \{0, 1\}, x \in X.
\end{aligned}
\tag{14}
$$

We observe that we have as many binary variables in this problem as we have ReLU nodes. Next, we show that the resulting lifted formulation results in a tractable convex MINLP.

Proposition 1 *Let Assumption 1 hold. Then it follows that Eq.* (14) *is a convex MINLP in the sense that the continuous relaxation of Eq.* (14) *is a convex NLP.*

Proof The result follows from the convexity of f, c, and X and the fact that the remaining constraints are affine (with the exception of the integrality restriction on z).

The choice of the big-M constants $M_{l,i}$ in (13) can have a great impact on the solution time of our MINLP. If $M_{l,i}$ is too small, the problem excludes solutions that should be feasible; but if it is too large, the space that must be searched by the solver may become so large as to be too computationally intractable for most computers.

These upper bounds are common in mixed-integer programs, and we follow the approach in [15] where we consider each neuron in our neural network individually, removing all constraints on other neurons in the same or subsequent layers. Then we set as our objective function to maximize $y_{l,i}$ in one iteration and $s_{l,i}$ in another iteration. These computed optimal values will serve as upper bounds $M_{l,i}^y$ and $M_{l,i}^s$, respectively, and can be used in our model. This process of obtaining bounds is related to the optimality-based bound tightening technique used in global optimization, see, e.g. [19]. Since these constants depend solely on the weights for the neural network as well as inputs to the neural network, we need to compute these bounds only once and may reuse them in any optimization problem involving this neural network. Alternative methods for handling the big-M constraints are considered in [22], and an alternative formulation with exponentially many constraints alleviating the above concerns is presented in [3].

With this formulation established, we may then easily pass the problem as is to any standard MINLP solver such as Gurobi [23], CPLEX [25], Bonmin [6], MINOTAUR [33], or Baron [39] to compute the solution. Because of the NP-complete nature of MINLPs, however, these problems do not scale well, and only problems involving modestly-sized networks (on the order of 100 hidden nodes) may be tractably solved. Neural networks used in commercial settings generally involve at least thousands of hidden nodes, resulting in thousands of binary variables, a number that typically is well beyond the reach of commercial solvers.

4.2 Formulating DNNs with Complementarity Constraints

In this section we discuss an alternative formulation of Eq. (1) as a nonconvex nonlinear program using complementarity constraints. This approach has the advantage of scaling significantly better for problems with larger neural networks, but with the caveat that solutions produced can only be guaranteed to be locally optimal.

Our approaches are based on the following observations. We can rewrite the ReLU activation function in Eq. (12) equivalently as a complementarity constraint (using vector notation):

$$y_l = \max(W_l^T y_{l-1} + b_l, 0) \ \Leftrightarrow \ 0 \le y_l \ \perp \ y_l \ge W^T y_{l-1} + b_l, \tag{15}$$

where $\perp$ means that for each component, i, both inequalities $y_{l,i} \ge 0$ and $y_{l,i} \ge [W_l^T y_{l-1} + b_l]_i$ are satisfied and at least one is satisfied at equality. By replacing the ReLU function with these complementarity constraints, we obtain a mathematical program with complementarity constraints (MPCC), which we can solve using standard NLP solvers; see, for example, [17, 30, 31, 37]. The mathematical program modeling language AMPL [18] allows the modeling of complementarity constraints: for example, the above constraint can be written as follows.

```
ReLUCompl{l in Level, i in Neuron[l]}: 0 <= y[l,i] complements
y[l,i] >= sum{j in Neuron[l-1]} W[l,i,j] y[l-1,j] + b[l,i] ;
```

The most successful NLP solvers handle MPCCs by reformulating the complementarity constraints in Eq. (15) by first introducing the same slack variables as for the MINLP and lifting the formulation

$$s_l = y_l - w_{l,i}^T y_{l-1} + b_l,$$

and then rewriting Eq. (15) equivalently as

$$s_l = y_l - w_{l,i}^T y_{l-1} + b_l \text{ and } 0 \le y_l \ \perp \ s_l \ge 0.$$

A nonlinear optimization formulation is then given as

$$s_l = y_l - w_{l,i}^T y_{l-1} + b_l, \quad y_l \ge 0, \quad s_l \ge 0, \text{ and } y_l^T s_l \le 0,$$

where the lower bound, $y_l^T s_l \ge 0$, is implied by the nonnegative bounds on $y_l, s_l \ge 0$, and again omitted for numerical reasons; see [17].

Because the last constraint involves the nonconvex term $y_l^T s_l$, the problem as a whole is nonconvex, and standard NLP solvers will produce only locally optimal solutions instead of globally optimal ones. Altogether, we then have the following NLP:

$$
\begin{aligned}
\underset{x,y,s,z}{\text{minimize}}\quad & f(x, y_L) \\
\text{subject to}\quad & c(x, y_L) \leq 0 \\
& y_{l,i} - s_{k,l} = w_{l,i}^T y_{l-1} + b_{l,i},\ l = 1, \ldots, L \\
& 0 \leq y_{l,i},\ 0 \leq s_{l,i}, \qquad i = 1, \ldots, N_l, l = 1, \ldots, L \\
& \sum_{l=1}^{L} \sum_{i=1}^{N_l} y_{l,i} s_{l,i} \leq 0, \\
& y_0 = x,\ x \in X.
\end{aligned}
\tag{16}
$$

Note that we could have used the constraint $y_l^T s_l \leq 0$ for each l separately. We prefer the formulation in Eq. (16) because it has better convergence behavior in practice, see [16]. Again we have a continuous variable for every node in our neural network as well as a slack variable for each ReLU node. The absence of integer variables, however, produces a much more scalable problem.

Next, we show that the stationarity conditions of the MPCC formulation, Eq. (16), and of the embedded formulation, Eq. (1), are equivalent. In particular, we consider the following pair of problems:

$$
\begin{aligned}
\underset{x}{\text{minimize}}\quad & f\left(x, \max(0, W^T x + b)\right) \\
\text{subject to}\quad & c\left(x, \max(0, W^T x + b)\right) \leq 0,
\end{aligned}
\tag{17}
$$

and

$$
\begin{aligned}
\underset{x}{\text{minimize}}\quad & f(x, y)) \\
\text{subject to}\quad & c(x, y)) \leq 0 \\
& 0 \leq y \perp y \geq W^T x + b.
\end{aligned}
\tag{18}
$$

In particular, we will show that generalized stationary points of Eq. (17) correspond to strongly stationary points of Eq. (18). We start by stating the following result about generalized gradients of composite functions.

Lemma 1 *Consider the function $h\left(x, \max(0, W^T x + b)\right)$, where $h(x, y)$ is smooth in both arguments. Its generalized gradient is*

$$
\partial_x h = \left\{ v \mid v = \nabla_x h + \hat{W} \nabla_y h,\ \hat{W} \in \partial_x \max(0, W^T x + b) \right\},
\tag{19}
$$

where all gradients of h are evaluated at the point $(x, \max(0, Wx + b))$, and where $\hat{W} \in \partial_x \max(0, W^T x + b)$ is a particular column scaling of W.

Proof We denote the columns of W by $W := [w_1 : \cdots : w_N]$, and we distinguish three cases for each $j = 1, \ldots, N$ that define a multiplier κ_j:

1. $w_j^T x + b_j < 0$: Then it follows that $\max(0, w_j^T x + b_j) = \{0\}$, and $\partial_x \max(0, w_j^T x + b_j) = 0$, and we set $\kappa_j = 0$.
2. $w_j^T x + b_j > 0$: Then it follows that $\max(0, w_j^T x + b_j) = w_j^T x + b_j$, and $\partial_x \max(0, w_j^T x + b_j) = \{w_j\}$, and we set $\kappa_j = 1$.
3. $w_j^T x + b_j = 0$: Then it follows that $\partial_x \max(0, w_j^T x + b_j) = \text{conv}(0, w_j)$ is the convex hull of the vectors 0 and w_j, and we let $\kappa_j \in [0, 1]$.

Overall, it follows that there exists $\kappa \in [0, 1]^N$, satisfying the three conditions above such that

$$
\partial_x \max(0, W^T x + b) \ni \hat{W} = W \text{diag}(\kappa).
\tag{20}
$$

The remainder of the proof follows by applying the chain rule to $h\left(x, \max(0, W^T x + b)\right)$. $\square$

Following [10, Theorem 6.1.1], the first-order conditions of the nonsmooth problem Eq. (17) can be stated in terms of condition on the generalized gradients of the Lagrangian function of Eq. (17):

$$
\mathcal{L}(x, \lambda) = f\left(x, \max(0, W^T x + b)\right) + \lambda^T c\left(x, \max(0, W^T x + b)\right).
\tag{21}
$$

If x^* is a regular local minimum of Eq. (17), then it follows that there exist multipliers, $\lambda^* \geq 0$ such that $\lambda^{*^T} c\left(x^*, \max(0, W^T x^* + b)\right) = 0$ and that $0 \in \partial_x \mathcal{L}(x^*, \lambda^*)$, i.e. there exists $\lambda^* \geq 0$ and κ^* as in Lemma 1 such that

$$0 \geq c\left(x^*, \max(0, W^T x^* + b)\right) \ \perp \ \lambda^* \geq 0 \tag{22a}$$

$$0 = \nabla_x f^* + \nabla_x c^* \lambda^* + W \operatorname{diag}(\kappa^*)\left(\nabla_y f^* + \nabla_y c^* \lambda^*\right), \tag{22b}$$

where f and c are evaluated at x^*.

Next, we state the definition of strong stationarity for the MPCCs Eq. (18), see, e.g. [40] for its general form, and then show that the two conditions are equivalent in our case.

Definition 1 (Scheel and Scholtes, [40]) We say (x^*, y^*) is a strongly stationary point of Eq. (18), if there exist multipliers $\mu^* \geq 0$, and ν_1^*, ν_2^* such that the following conditions are satisfied:

$$c^* \leq 0 \text{ and } 0 \leq y^* \ \perp \ y^* \geq W^T x^* + b \tag{23a}$$

$$\mu^{*^T} c^* = 0 \tag{23b}$$

$$0 = \nabla_x f^* + \nabla_x c^* \mu^* + W \nu_2^* \tag{23c}$$

$$0 = \nabla_y f^* + \nabla_y c^* \mu^* - \nu_1^* - \nu_2^* \tag{23d}$$

$$y_j^* > 0 \Rightarrow \nu_{1j}^* = 0 \text{ and } y_j^* > w_j^T x^* + b_j \Rightarrow \nu_{2j}^* = 0 \tag{23e}$$

$$0 = y_j^* = w_j^T x^* + b_j \Rightarrow \nu_{1j}^* \geq 0, \nu_{2j}^* \geq 0, \tag{23f}$$

where all functions and gradients are evaluated at (x^*, y^*), i.e. $f^* := (x^*, y^*)$ etc.

We note, that the last two conditions represent the fact that the multipliers of the complementarity ν_{1j}^*, ν_{2j}^* need only be nonnegative for biactive or degenerate complementarity constraints. We can now show that the two sets of first-order conditions, Eq. (22) and Eq. (23), are equivalent.

Theorem 1 x^* *is a stationary point of Eq.* (17)*, if and only if, (x^*, y^*) is a strongly stationary point of Eq.* (18) *with $y^* = \max(0, W^T x^* + b)$.*

Proof Assume that (x^*, y^*) is a strongly stationary point that satisfies Eq. (23). Then it follows that $y^* = \max(0, W^T x^* + b)$, and hence that $c\left(x^*, \max(0, W^T x^* + b)\right) = c(x^*, y^*) \leq 0$, which implies that x^* is a feasible point in Eq. (17). Moreover, complementarity holds, i.e., $\mu^{*^T} c^* = 0$, and hence, it remains to show that $0 \in \partial_x \mathcal{L}(x^*, \mu^*)$. Solving Eq. (23d) for ν_2^*, and substituting into Eq. (23c), we obtain

$$0 = \nabla_x f^* + \nabla_x c^* \mu^* + W\left(\nabla_y f^* - \nabla_y c^* \mu^* - \nu_1^*\right),$$

and it remains to be shown that there exists $\kappa \in [0,1]^N$ as in Lemma 1 such that $W\left(\nabla_y f^* - \nabla_y c^* \mu^* - \nu_1^*\right) = W \operatorname{diag}(\kappa)\left(\nabla_y f^* - \nabla_y c^* \mu^*\right)$. We now examine each component, $j = 1, \ldots, N$, of the complementarity constraint in turn:

1. $w_j^T x^* + b_j < 0$: Then it follows that $y_j^* = 0$, $\nu_{2j}^* = 0$, and ν_{1j}^* is not sign-constrained. It follows from Eq. (23d) that $\nu_{1j}^* = [\nabla_y f^* - \nabla_y c^* \mu^*]_j$, and hence that $\kappa_j = 0$.
2. $w_j^T x^* + b_j > 0$: Then it follows that $y_j^* = w_j^T x^* + b_j > 0$, and hence that $\nu_{1j}^* = 0$. Thus, we can set $\kappa_j = 1$.
3. $w_j^T x^* + b_j = 0 = y_j^*$: Then it follows that $\nu_{1j}^*, \nu_{2j}^* \geq 0$ from Eq. (23f). From Eq. (23d), we obtain that $[\nabla_y f^* + \nabla_y c^* \mu^*]_j \geq 0$, and $\nu_{1j}^* \in [0, [\nabla_y f^* + \nabla_y c^* \mu^*]_j]$ corresponds to choosing $\kappa_j \in [0, 1]$.

Putting the three cases together, we see that there exists $\kappa \in [0,1]^N$ satisfying the conditions from Lemma 1 such that Eq. (22) holds.

The reverse direction follows by a similar construction. □

We show that the stationarity conditions for the two problems are equivalent. To handle the points of nondifferentiability in the embedded formulation, we consider generalized gradients, which can be defined for all x in the domain of f as follows:

$$\partial f(x) = \operatorname{conv}\{\lim_k \nabla f(x_k) : x_k \to x, x_k \notin S\} \tag{24}$$

where S is the set of points of nondifferentiability of f and $\mathrm{conv}(A)$ is the convex hull of the set A. Since ReLU networks are piecewise linear functions by construction, the generalized gradient will be given by a convex hull of the gradients of linear functions on finitely many regions in the neighborhood of a given point. Stationarity of the function at a point is then equivalent to having $0 \in \partial f(x)$.

In particular, we consider the following problem:

$$\begin{aligned}
&\underset{x}{\text{minimize}} \ f\left(x, \mathrm{DNN}(x)\right) \\
&\text{subject to } c\left(x, \mathrm{DNN}(x)\right) \leq 0,
\end{aligned} \tag{25}$$

where each intermediate layer is given as in Eq. (2) with ReLU activation. Assume that we have L layers and that there are n_ℓ neurons in layer ℓ.

Before we state the theorem, we will introduce some notation. We write $x^\ell(x^0)$ to be the values of each neuron in layer ℓ given input x^0. We write the affine transformation of the in layer ℓ as $A^\ell x = W^{\ell,T} x + b^\ell$. Note $A_i^\ell x = w_i^{\ell,T} x + b^i$ where w_i is the ith column of W. Then we can write the evaluation of the DNN as

$$\mathrm{DNN}(x) = \sigma(A^L(\sigma(A^{L-1}(\cdots(\sigma(A^1 x))\cdots)))).$$

We can partition the neurons of layer ℓ into three sets given an x^0: $I^{\ell,-}(x^0)$, $I^{\ell,0}(x^0)$, and $I^{\ell,+}(x^0)$. We define these to be the indices i of the neurons in layer ℓ for which $A_i^\ell x^{\ell-1}(x^0)$ is negative, zero, and positive, respectively. The sets $I^+(x^0), I^0(x^0), I^-(x^0)$ are then defined to be all layer-neuron pairs (ℓ, i) where $i \in I^{\ell,+}(x^0), I^{\ell,0}(x^0)$, or $I^{\ell,-}(x^0)$ respectively.

Given a set of neurons in the neural network $B = \{(\ell_1, i_1), \ldots, (\ell_N, i_N)\}$ (we will refer to these as *activation patterns*), we define F_B to be the affine form given when the set of neurons in B are active and all other neurons are inactive. Put precisely, for each layer ℓ, given affine form $A^\ell x = W^{\ell,T} x + b^\ell$, if neuron (ℓ, i) is inactive, we replace column w_i and component b_i with zeros. Our new affine form can then be written:

$$\hat{A}^\ell(x) = \hat{W}^{\ell,T} x + \hat{b}^\ell, \ \hat{W}^\ell = W \mathrm{diag}(\kappa^\ell), \ \hat{b} = \mathrm{diag}(\kappa^\ell) b.$$

where κ^ℓ is vector with κ_i^ℓ equal to 1 if neuron $(\ell, i) \in B$. and zero otherwise. Then $F_B(x) = \hat{A}^L \hat{A}^{L-1} \cdots \hat{A}^1 x$. We observe that $\mathrm{DNN}(x) = F_{I^+(x)}(x)$. Note that F_B is now smooth as a composition of affine functions and we can compute the Jacobian J_{F_B} simply by the chain rule:

$$J_{F_B} = \hat{W}^1 \hat{W}^2 \cdots \hat{W}^L. \tag{26}$$

In light of Eq. (24), the generalized gradient at x can then be written as the convex hull of these Jacobians associated with the activation pattern for each region in the neighborhood of x. Any such activation pattern must necessarily include $I^+(x)$ and exclude $I^-(x)$, but may only include a subset of $I^0(x)$. In the event there is a region corresponding each subset of $I^0(x)$, we can write the generalized gradient in a nice form given by the following proposition:

Proposition 2 *Suppose at a point x, there is a region corresponding to each activation pattern of the form $I^+(x) \cup A$ where $A \subset I^0(x)$. Let S be the set of all matrices of the form*

$$W^1 diag(\kappa^1) \cdots W^L diag(\kappa^L)$$

where

$$\begin{cases}
\kappa_i^\ell = 1 & (\ell, i) \in I^+(x), \\
\kappa_i^\ell \in [0, 1] & (\ell, i) \in I^0(x), \\
\kappa_i^\ell = 0 & (\ell, i) \in I^-(x).
\end{cases}$$

Then the generalized gradient at x, $\partial \mathrm{DNN}(x)$, equals S.

Proof First, we remark that any Jacobian J_{F_B} where $B = I^+(x) \cup A$ is clearly in S as we can take $\kappa_i^\ell = 1$ for all $(\ell, i) \in A$ and take it equal to 0 for $(\ell, i) \in I^0(x) \setminus A$ and the two expressions should agree. Furthermore any convex combination of the J_{F_B} should be contained in S as S is a convex set. Hence $\partial \mathrm{DNN}(x) \subset S$.

Now given a matrix $V = W^1 \mathrm{diag}(\kappa^1) \cdots W^L \mathrm{diag}(\kappa^L)$ in S, we show that is in $\partial \mathrm{DNN}(x)$, the convex hull of Jacobians J_{F_B} with $B = I^+(x) \cup A, A \subset I^0(x)$. we enumerate $I^0(x) = \{(\ell_1, i_1), \ldots, (\ell_N, i_N)\}$. Consider first any alteration $\hat{V}_1$ of V by fixing $\kappa_{i_1}^{\ell_1}$ and replacing $\kappa_{i_j}^{\ell_j}$ by $\hat{\kappa}^{\ell_j} \in \{0, 1\}$ for $j > 1$. Note that $\hat{V}_1$ is a convex combination of the two matrices given by replacing $\kappa_{i_1}^{\ell_1}$ with 1 and by replacing it with 0. Both these matrices
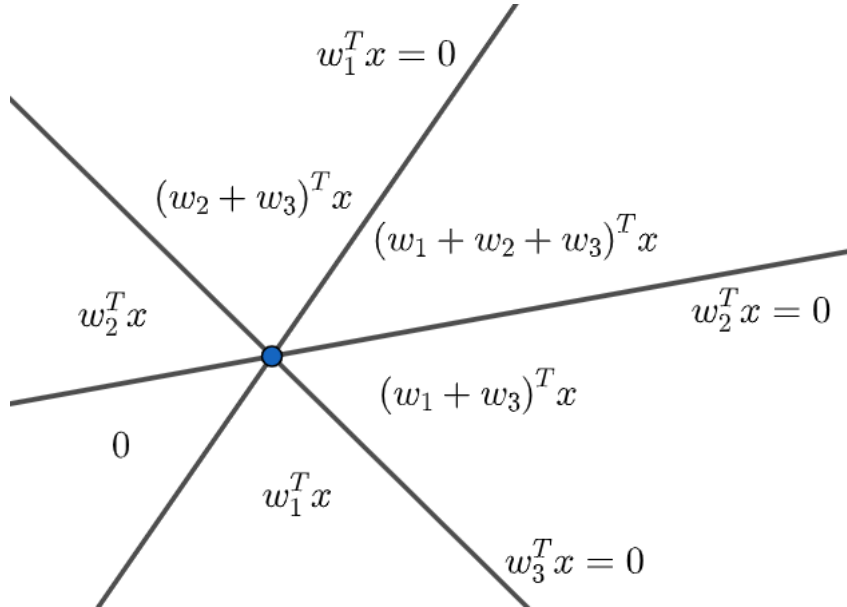
Fig. 6: Regions of a zero-bias, single layer network with two input variables and three output variables.

are in $\partial \mathrm{DNN}(x)$ as this set includes all Jacobians associated with any activation pattern on $I^0(x)$. Hence we must have that $\hat{V}_1 \in \partial \mathrm{DNN}(x)$. If we then consider fixing the first two components, and replacing the remaining such components to produce $\hat{V}_2$, in the same fashion, we can find two matrices of the form $\hat{V}_1$ of which $\hat{V}_2$ is a convex combination and hence $\hat{V}_2 \in \partial \mathrm{DNN}(x)$. Repeating until we have fixed all components, we will have that $V \in \partial \mathrm{DNN}(x)$.

Without all the necessary activation patterns, the generalized gradient will not necessarily take this form. As a counterexample, consider a two-layer network with one input variable and $W^1 = (1,1)$, $W^2 = (1,-1)^T$ and the biases are zero. This corresponds to having $\mathrm{DNN}(x) = \mathrm{ReLU}(x) - \mathrm{ReLU}(x) = 0$, and so the generalized gradient is $\{0\}$ everywhere whereas the formulation given above would introduce many more possible gradients.

For a slightly more interesting example, we consider a one-layer network with two input variables, no bias, and three output variables, i.e., $\mathrm{DNN}(x) = \mathrm{ReLU}(W^T x) = \mathrm{ReLU}(w_1^T x) + \mathrm{ReLU}(w_2^T x) + \mathrm{ReLU}(w_3^T x)$. The input domain $\mathbb{R}^2$ is then divided into regions by the lines $w_i^T x = 0$ for $i = 1,2,3$ each intersecting at the origin as can be seen in Figure 6. Observe that in the neighborhood of the origin, there are only 6 neighboring regions and is missing two regions where the output would be $(w_1 + w_2)^T x$ and another where the output is w_3^T corresponding to activation patterns where only the first two neurons are active and only the third neuron is active, respectively.

Before we state our main theorem, we need to introduce a geometric definition about hyperplanes. Suppose we have a collection of hyperplanes $H_1, \ldots, H_n \subset \mathbb{R}^d$ where H_i is given by $\{x \in \mathbb{R}^d : a_i^T x = b_i\}$ for some vectors $a_i \in \mathbb{R}^d$ and constants $b_i \in \mathbb{R}$. We say the collection is in general position if the intersection of m hyperplanes has dimension $d - m$ if $m \leq d$ and is empty if $m > d$. In particular, if $d = 2$, the hyperplanes are lines in $\mathbb{R}^2$ and they are in general position if no three lines intersect at a point and no two lines are parallel. From FIND CITATION HERE, it follows that if we have m intersecting hyperplanes where $m \leq d$, they divide $\mathbb{R}^d$ into 2^m regions, and we can identify each region by choosing for each i whether $a_i^T x > b_i$ or $a_i^T x < b_i$.

We now state our general theorem:

Theorem 2 *Given $x^{0,*}$, suppose the set of hyperplanes $\{H_{\ell,i} : \ell = 1, \ldots, L, i \in I^{\ell,+}(x^{0,*}), H_{\ell,i} = \{x : L_i^{\ell,+} x = 0\}\}$ is in general position. Then x^* is a stationary point of Eq. (25), if and only if, $(x^{0,*}, x^{1,*}, \ldots, x^{L,*})$ is a strongly stationary point of Eq. (27) with $x^{\ell,*} = x^\ell(x^{0,*}$.*

Lemma 2 *Suppose a finite set of hyperplanes $H_1, \ldots, H_n$ where $H_i = \{x : a_i^T x = b_i\}$ has nonempty intersection. Then the set $H_1, \ldots, H_{n-1}, H_n^A$ where $H_n^A = \{x : (\sum_{i \in A} a_i)^T x + a_n = \sum_{i \in A} b_i + b_n$ is also in general position for any $A \subset \{1, \ldots, n-1\}$.*

Proof Because the hyperplanes intersect, we can change coordinates so that each hyperplane intersects the origin, so without loss of generality, we can write $b_i = 0$ for all i. In this setting, hyperplanes being in general position is equivalent to the normal vectors being linearly independent. Obviously if $a_1, \ldots, a_n$ are linearly independent, $a_1, \ldots, a_{n-1}, a_n + \sum_{i \in A} a_i$ are linearly independent.

The corresponding MPCC formulation of the same problem is then given by

$$\begin{aligned}
&\underset{x}{\text{minimize}} \ f\left(x^0, x^L\right)) \\
&\text{subject to } c\left(x^0, x^L\right)) \leq 0 \\
&\qquad 0 \leq x^\ell \perp x^\ell \geq W^{\ell,T} x^{\ell-1} + b^\ell, \ \ell = 1, \ldots, L.
\end{aligned} \tag{27}$$

Next, we state the definition of strong stationarity for the MPCCs Eq. (18), see, e.g. [40] for its general form, and then show that the two conditions are equivalent in our case.

Definition 2 (Scheel and Scholtes, [40]) We say $(x^{0,*}, x^{1,*}, \ldots, x^{L,*})$ is a strongly stationary point of Eq. (27), if there exist multipliers $\mu^* \geq 0$, and ν_1^*, ν_2^* such that the following conditions are satisfied:

$$c^* \leq 0 \text{ and } 0 \leq x^{\ell,*} \perp x_{\ell,*} \geq W^{\ell,T} x^{\ell-1,*} + b^\ell, \ \ell = 1, \ldots, L \tag{28a}$$

$$\mu^{*,T} c^* = 0 \tag{28b}$$

$$0 = \nabla_{x^0} f^* + \nabla_{x^0} c^* \mu^* + W^1 \nu_2^{1,*} \tag{28c}$$

$$-\nu_1^{\ell,*} - \nu_2^{\ell,*} + W^{\ell+1} \nu_2^{\ell+1,*} = 0, \ \ell = 1, \ldots, L-1 \tag{28d}$$

$$\nabla_{x^L} f^* + \nabla_{x^L} c^* \mu^* - \nu_1^{L,*} - \nu_2^{L,*} = 0 \tag{28e}$$

$$x_j^{\ell,*} > 0 \Rightarrow \nu_{1j}^{\ell,*} = 0 \text{ and } x_j^{\ell,*} > w_j^{\ell,T} x^{\ell-1,*} + b_j^\ell \Rightarrow \nu_{2j}^{\ell,*} = 0, \ \ell = 1, \ldots, L \tag{28f}$$

$$0 = x_j^{\ell,*} = w_j^{\ell,T} x^{\ell-1,*} + b_j \Rightarrow \nu_{1j}^{\ell,*} \geq 0, \nu_{2j}^* \geq 0, \ \ell = 1, \ldots, L \tag{28g}$$

where all functions and gradients are evaluated at (x^*, y^*), i.e. $f^* := f(x^*, y^*)$ etc.

Proposition 3 *Given a strongly stationary point satisfying Eq. (28), there exist $\hat{W}^1, \ldots, \hat{W}^L$, column scalings of $W^1, \ldots, W^L$ such that $\nabla_{x^0} f^* + \nabla_{x^0} c^* \mu^* + \hat{W}^1 \cdots \hat{W}^L (\nabla_{x^L} f^* + \nabla_{x^L} c^* \mu^*) = 0$.*

Proof To construct such matrices, we work backwards starting with layer L and examine each component $j = 1, \ldots, N_L$. Backsubstituting $\nu_2^{L,*}$ using Eq. (28d) and Eq. (28e), we have

$$-\nu_1^{L-1,*} - \nu_2^{L-1,*} + W^L (\nabla_{x^L} f^* + \nabla_{x^L} c^* \mu^* - \nu^{L,*}) = 0$$

For a given component j we can determine the appropriate scaling factor κ_j^L in three cases:

1. $w_j^{L,T} x^{L-1,*} + b_j^L < 0$: Then it follows that $x_j^{L,*} = 0$, $\nu_{2j}^{L,*} = 0$, and ν_{1j}^* is not sign-constrained. It follows from Eq. (28e) that $\nu_{1j}^{L,*} = [\nabla_{x^L} f^* - \nabla_{x^L} c^* \mu^*]_j$, and hence that $\kappa_j^L = 0$.

2. $w_j^{L,T} x^{L-1,*} + b_j^L > 0$: Then it follows that $x_j^{L,*} = w_j^{L,T} x^* + b_j > 0$, and hence that $\nu_{1j}^{L,*} = 0$. Thus, we can set $\kappa_j^L = 1$.

3. $w_j^{L,T} x^{L,*} + b_j^L = 0 = x_j^{L,*}$: Then it follows that $\nu_{1j}^{L,*}, \nu_{2j}^{L,*} \geq 0$ from Eq. (28g). From Eq. (23d), we obtain that $[\nabla_{x^L} f^* + \nabla_{x^L} c^* \mu^*]_j \geq \nu_{1j}^{L,*}, \nu_{2j}^{L,*} \geq 0$, and given $\nu_{1j}^* \in [0, [\nabla_y f^* + \nabla_y c^* \mu^*]_j]$ we choose $\kappa_j^L = \frac{[\nabla_{x^L} f^* + \nabla_{x^L} c^* \mu^*]_j - \nu_j^{L,*}}{[\nabla_{x^L} f^* + \nabla_{x^L} c^* \mu^*]_j}$ which is in $[0, 1]$.

Hence we can write $W^L (\nabla_{x^L} f^* + \nabla_{x^L} c^* \mu^* - \nu^{L,*}) = \hat{W}^L (\nabla_{x^L} f^* + \nabla_{x^L} c^* \mu^*)$ where $\hat{W}^L = W \text{diag}(\kappa^L)$. Repeating this process for each ℓ, we arrive at $\nabla_{x^0} f^* + \nabla_{x^0} c^* \mu^* + \hat{W}^1 \cdots \hat{W}^L (\nabla_{x^L} f^* + \nabla_{x^L} c^* \mu^*) = 0$.

5 Numerical Experiments

Here, we present our numerical experiments with neural-network surrogates for the three sample applications introduced in Section 2.

We assess the performance of solvers for the MIP, MPCC, and embedded formulations of Eq. (3). We use the commercial solver CPLEX [25] to solve the MIP formulation of the problem and the solver Ipopt [47] to solve the MPCC and embedded formulations of the problem. Both solvers are run with the default options. All experiments are performed on a single thread on an Intel Xeon Gold 6130 CPU and 188 GB of memory.

5.1 Numerical Experiments with Engine Design Optimization

We have applied our models and algorithms to a collection of neural networks with varying numbers of layers to study how well each formulation scales. The architectures we tested all have the simple structure of an input layer with the three input variables, followed by n hidden layers of 16 nodes, and then the output layer with the three output variables. We consider networks with 1, 3, and 5 hidden layers and train each of these networks for 20 epochs on the simulation data produced by the simulator of [2]. We use the `adam` solver in TensorFlow [1] to train the neural network.

The number of auxiliary variables for the MIP and MPCC formulations scales with the number of ReLU neurons as well as with the number of time steps. The last layer has no ReLU neurons since we want the final layer to be able to take all real values, but each of the hidden layers use ReLU neurons. We will then have $T \times (\#\text{Hidden Layers}) \times (\#\text{Nodes Per Layer})$ additional sets of auxiliary variables.

Solving the full integer program with 1500 time steps to optimality is generally intractable (even for a single-layer network, this amounts to $1500 \times 16 \times 1 = 24000$ binary variables, beyond the scope of any general-purpose state-of-the-art solver), so we consider instead a coarser discretization using larger time steps.

Instead of using data at each second as is presented in the original data, we consider time intervals of 750, 500, 250, 150, 30, 15, 10, 5, and 1 seconds. Since we have 1,500 seconds of data, this corresponds to 3, 6, 10, 50, 100, 150, 300, and 1,500 time steps for which we are evaluating our neural network. At each of the larger time steps, our prescribed torque profile will be the average of the prescribed torques in that interval. This approach gives us separate problems for each choice of coarseness of the discretization and for each choice of neural network architecture. As the number of time steps increases, we have observed that the computed solutions converge to the solution of the full problem.

For each instance, we run until convergence to a solution, 3 hours have passed or Ipopt has performed 3000 iterations. If the solver has not converged at the end of the experiment, we record the best solution discovered which is feasible which we determined to be having a constraint violation under 10^{-6}.

The amount of time needed until a solver found its best solution with a time limit of 3 hours on each architecture with each time step size is displayed in Figure 7. The percentage gap between the computed objective and the best known objective for each solver is shown in Figure 8.

Except for the very smallest problems CPLEX fails to converge to the optimal solution of the MIP within the allotted time limit of 3 hours. Even worse, it fails to even find a feasible solution for most problems with architectures with 3 or 5 hidden layers. This is to be expected because, except for the smallest cases, these problems can involve thousands of binary variables. With the warmstart, CPLEX is actually able to find solutions with significantly better objective values for each of the configurations. It still fails to prove optimality for any of these solutions, however, and times out for the same set of problems as without the warmstart.

For the MPCC, on the other hand, Ipopt always finds locally optimal solutions within the time limit, often in seconds for problems with a small number of time steps. These solutions may be suboptimal, worse by up to 10% in some instance. With the warmstart solution, Ipopt solves the MPCC in terms of speed generally by a factor of 2. Furthermore, the objective solution tends to have a significantly smaller objective although there are a few exceptions. Solving the MIP without warmstart finds a better solution than the MPCC formulation in only three problem instances and in all three cases the MPCC solution is within 0.1% of the MIP's globally optimal solution.

In almost all cases, the embedded network formulation is able to find its best solution more quickly than either the MIP or MPCC formulations. However, without the warmstart, the embedded ReLU network formulation performs worse in all but a few cases with an objective value which is up to 10% larger than the MPCC formulation objective values. With the warmstart, the objective values are comparable except for the largest network where in most cases, Ipopt failed to find any solutions better than the warmstart solution in the allotted time and so the computed objectives are worse by a small margin. The time until Ipopt finds the best solution to the embedded formulation is faster than the MIP formulation but often slower than the MPCC formulation. Each iteration is faster since the formulations involve significantly fewer variables, but due to issues with convergence, the solver often takes significantly more steps. This is noticeable for the 5-layer network when warmstarting, as Ipopt often fails to converge to any solution better than the warmstart.

Overall, these experiments confirm our prior suspicions that solving using the MILP formulation without warmstart quickly becomes computationally intractable as the size of the neural network increases past modest architectures. Switching to the MPCC formulation of the problem offers significant speedup at the cost of losing global optimality, although when the MPCC formulation obtains a worse objective value, it tends to be by only a marginal amount, and when it does better, it can be by a significant amount. The embedded formulation on the other hand can provide speed and scalability, but often encounters difficulty with convergence and finds slightly
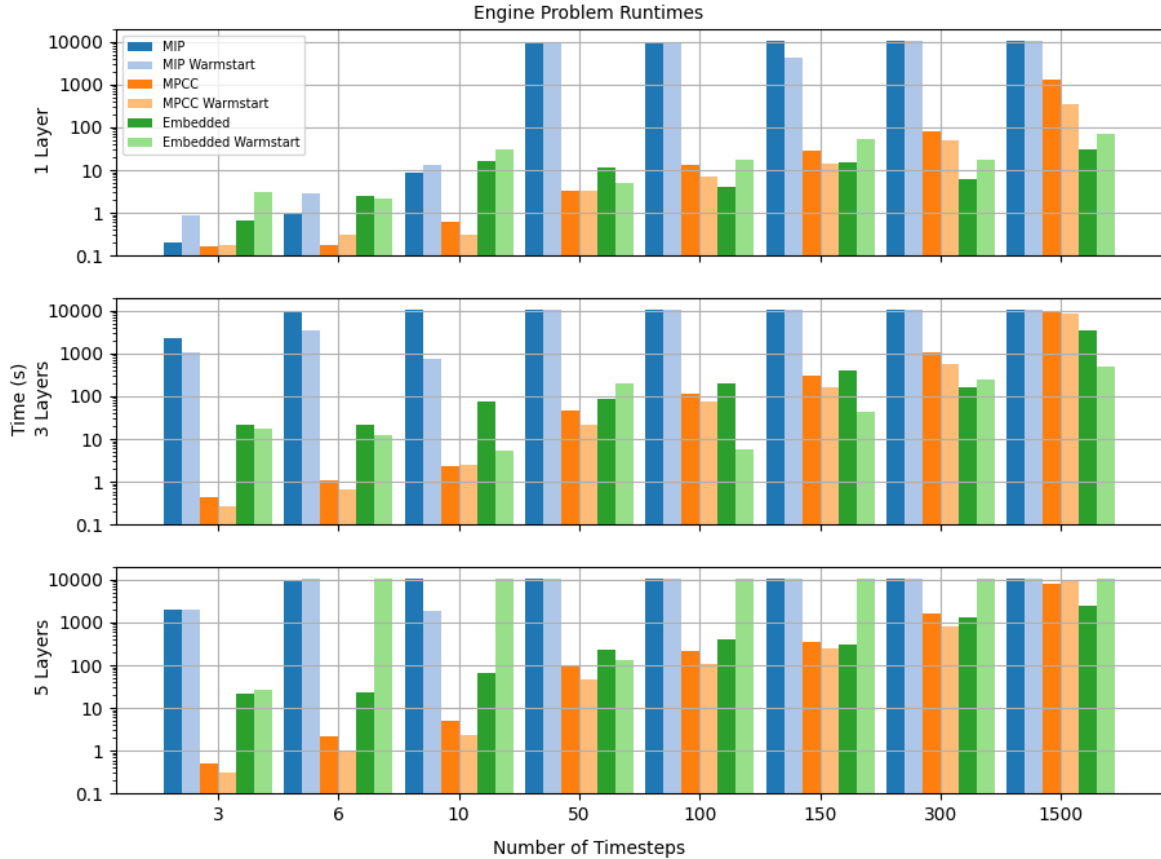
Fig. 7: Time until a solver found its best solution within 3 hours (10800 seconds) on each architecture for varying values of T. If a runtime of 10800 seconds is recorded, this indicates the solver failed to find a feasible solution in the full 3 hours (if a warmstart is used, this means it failed to find a feasible solution better than the warmstart).

worse objective values. For all formulations, significant gains can be realized both in terms of solving time and solution quality by providing the solver with a high-quality warmstart solution.

5.2 Numerical Experiments with Adversarial Attack Generation

Next we consider each of the formulations of the adversarial attack generation problem in Eq. (6) by considering neural networks trained on the MNIST handwritten digit recognition data set [29]. We consider 10 different architectures corresponding to having an input layer with 28×28 nodes followed by either 1 or 2 fully connected layers with 20, 40, 60, 80, or 100 hidden ReLU nodes each and then the output layer with 10 nodes and a softmax activation function.

If we use the mixed integer formulation from Eq. (14) for the DNN constraints, then depending on our choice of norm, it becomes an mixed-integer linear program (for L^1 or L^∞ norms) or a mixed-integer quadratic program (for L^2 norm) that we can solve using CPLEX. We will use the L^2 norm in experiments.

For each neural network, we train the model by minimizing the categorical cross entropy loss function for 10 epochs using the 60,000 digits of training data. Each neural network attains a test accuracy of 96–98% when tested on the 10,000 digits of testing data.

We set $\alpha = 1.2$, meaning the probability for the given classification l is 1.2 times higher than for any other. On each architecture, we solve the problem for 100 digits from the training data, with the goal of finding the closest image to the given digit that will be classified as a zero. As a warmstart, we initialize the solution in each solve to be a digit which is classified as the desired digit, so that the solver starts at a feasible point. We run each iteration until convergence to the globally optimal solution (for the MIP formulation) or to a locally optimal solution (for the MPCC and embedded network formulations) or until one hour has passed or 3000 iterations have occurred in Ipopt, at which point we terminate with the best feasible solution seen so far.

In Table 3, we tabulate the results of the experiments for all three formulations. For each neural network architecture, we record the average solve times and objectives for CPLEX and Ipopt over the 100 solves. We
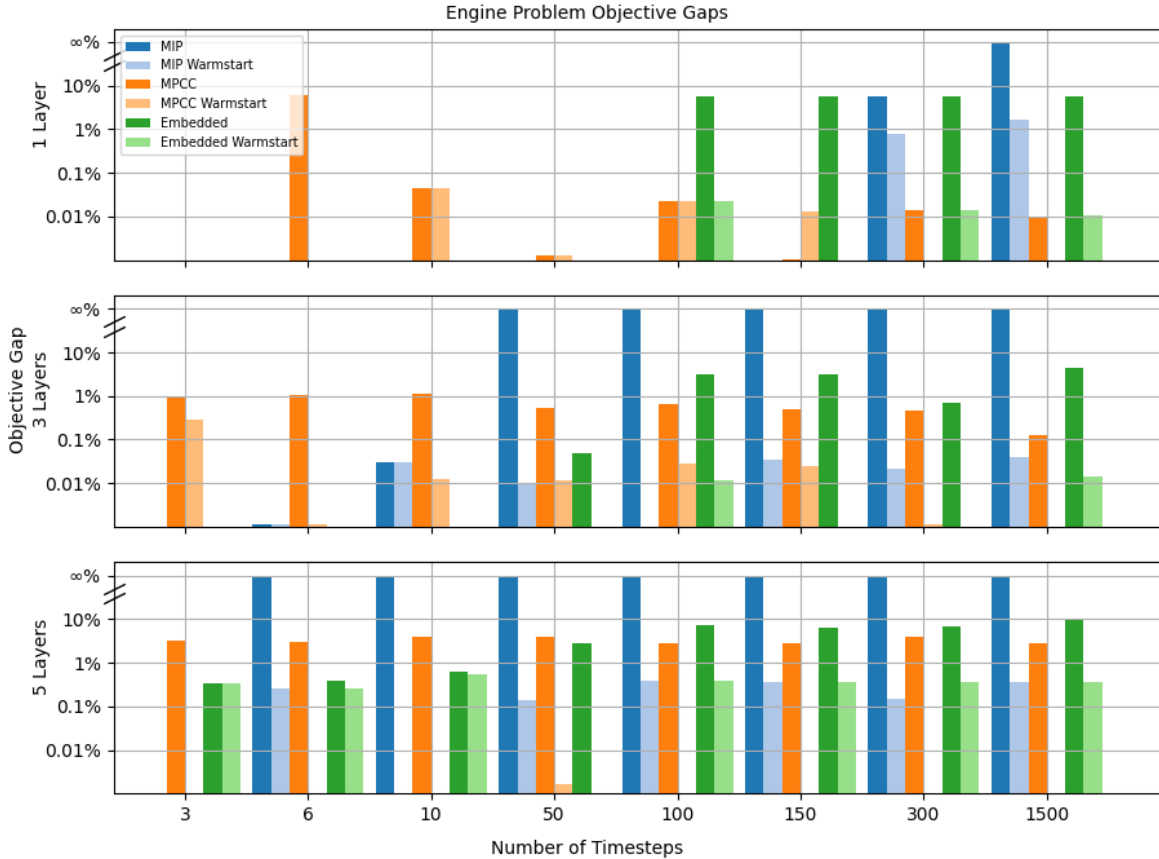
Fig. 8: Percentage gap in objective between final objective value and best known objective value for each solver on each architecture for varying values of T. A value of 0 indicates the solver found the best known objective value and a value of ∞ indicates that no solution was found.

Architecture	MIP				MPCC			Embedded ReLU			
	Avg. Time	Avg. Obj.	Num. Opt.	Num. Feas.	Avg. Time	Avg. Obj.	Num. Feas.	Avg. Time	Avg. Obj.	Avg. Feas. Obj.	Num. Feas.
784,20,10	1.8	**1.79**	100	100	**0.3**	1.80	100	35.5	14.67	1.80	88
784,40,10	76.9	**1.75**	100	100	**1.5**	1.77	100	28.4	8.64	1.77	94
784,60,10	2302	**1.85**	52	100	**4.8**	1.87	100	31.8	1.91	1.91	100
784,80,10	2571	**2.04**	38	100	**14.2**	2.07	100	48.1	17.35	2.08	86
784,100,10	3078	**2.16**	16	100	**22.9**	2.18	100	30.1	10.66	2.09	92
784,20,20,10	106.1	**1.20**	100	100	**0.8**	1.22	100	26.6	4.04	4.04	100
784,40,40,10	3100	1.90	16	100	**4.4**	**1.77**	100	38.8	19.97	1.68	84
784,60,60,10	3513	6.85	5	100	**14.3**	**1.88**	100	29.8	12.61	12.61	100
784,80,80,10	3587	27.59	1	76	36.1	**1.46**	100	**27.8**	3.83	3.83	100
784,100,100,10	3600	54.83	0	42	57.0	**1.78**	100	**35.1**	1.83	1.83	100

Table 3: Comprehensive results for solves of the MIP formulation using CPLEX, the MPCC formulation using Ipopt, and the embedded ReLU formulation also using Ipopt. Times are recorded in seconds.

also record how many times CPLEX finds the optimal solution to the MIP as well as how many times in each formulation, a feasible solution that is better than the initial solution is found. For the embedded formulation, we also include the average objective excluding the infeasible problems as we observed a significant difference between these cases.

The average time to convergence for the CPLEX solves of the MIP formulation is significantly higher than that of the Ipopt solves of the MPCC formulation. Most CPLEX solves for the larger networks time out at 1 hour before finding an optimal solution; for the smaller networks, it can take 10 to 100 times longer to converge to a solution. Ipopt, on the other hand, takes less than a minute on average to converge to a locally optimal solution of the MPCC formulation. The embedded formulations are fairly uniform in how long they take to find their best
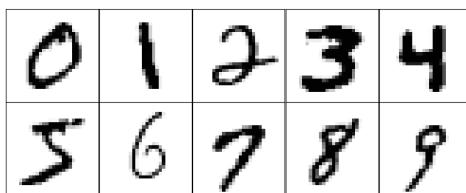
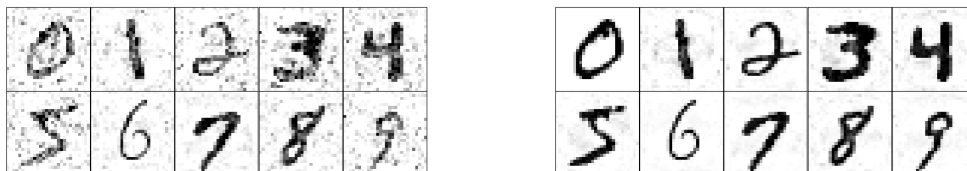*Fig. 9:* Suite of 10 digits from the MNIST training data for which adversarial attacks are generated.

Fig. 10: Perturbed images produced after 1 hour of computation when using CPLEX to solve the MIP formulation (left) and Ipopt to solve the NLP formulation (right) with given digits in Figure 9 for the neural network with 2 hidden layers of 100 nodes

solution which makes sense given that the the problem does not change in size with the number of neurons in the network.

The longer solve times of the MIP formulation have the advantage of eventually uncovering better solutions the majority of the time even if they are not provably optimal. For all the single hidden layer architectures and the two smallest double hidden layer architecture, the MIP formulation almost always produces a better solution than the MPCC formulation does. These solutions generally offer only a marginal improvement on the order of 1–2% difference in the objective. For perturbations of this magnitude, the differences are essentially imperceptible.

For the largest networks, the MIP formulation fares poorly and cannot find a better feasible solution than the initial solution in all cases for the two largest networks. The feasible solutions it does find are of noticeably lower quality: out of all instances from the three largest networks, CPLEX found the best solution in only 13 cases out of 300. The MIP solutions for the larger networks are visually worse, as can be seen in a comparison between the perturbed images generated from CPLEX and Ipopt in Figure 10 with respect to the original images in Figure 9.

When compared to the MIP and MPCC formulations, the embedded formulation while often faster and more scalable, can be more unstable in the sense that while using it, Ipopt can fail to find feasible solutions even though for this problem, they certainly exist. The ReLU networks are more prone to failure and exhibit the poor convergence behavior as in Figure 5 leading to failure to find solutions in about 5% of instances. In a few cases, the objective value of the found solutions are also noticeably worse.

5.3 Numerical Experiments with Oil Well Networks

Next, we consider solving the oil well problem using each of our formulations. We again used CPLEX for the MIP formulation and solved the problem with a time limit of 1 hour recording the best time for each neural network configuration. This problem involves binary variables even in the complementary constraint formulation, so we needed to use an MINLP solver, and elected to use the solver Bonmin [6]. We used the default branch-and-bound scheme with Ipopt as the NLP solver. Because no MINLP solver is currently supported on JuMP for handling both integer variables and user-defined nonlinear functions, we could not solve the problem as stated using the embedded neural network formulations.

The results for the solves on the full problem are presented in Table 4. We observe that in this small set of problems, the MPCC formulation outperforms the MIP formulation in both network configurations both in terms of time taken as well as objective value produced. CPLEX fails to even produce a solution for the deep neural network configuration. These are hard problems to be sure as in 3 of the 4 solves performed, the solutions found are not proven optimal.

In order to obtain a problem which did not have binary variables, we considered fixing each of the binary variables randomly to either 0 or 1 while still ensuring feasibility. If we represent the neural networks using complementary constraints or using the embedded formulation, then the problems become standard NLPs which we can directly solve using Ipopt as before. In our setup, we considered 10 different configurations of the binary variables, and solved them using CPLEX for the MIP formulation and Ipopt for the rest. We ran each instance until convergence or 1 hour had passed and recorded the best solution found.

20

	Shallow		Deep	
Solver	Time (s)	Objective	Time (s)	Objective
CPLEX	3388*	1.271	3600*	NaN
bonmin	1865*	**1.283**	1275.3	**1.304**

Table 4: The time to find the best solution for each instance of the full oil well problem with a time limit of 1 hour. A * indicates that the solver could not prove the solution optimal.

	Shallow			Deep		
Instance	MIP	MPCC	ReLU NN	MIP	MPCC	Embedded
1	17.5	**1.8**	10.3	3600*	**1.6**	16.7
2	22.0	**1.2**	8.2	3600*	**1.4**	19.4
3	23.9	**1.5**	7.5	3600*	**1.7**	14.2
4	40.3	**1.5**	6.2	3600*	**1.4**	7.2
5	19.0	**1.4**	4.0	3600*	**1.6**	13.0
6	21.0	**2.1**	3.0	3600*	**1.7**	7.1
7	169.4	**1.7**	5.4	3600*	**1.2**	14.1
8	14.9	**1.5**	5.7	3600*	**1.4**	7.3
9	86.3	**1.9**	4.3	3600*	**1.6**	12.1
10	15.5	**1.7**	5.8	3600*	**1.5**	5.8

Table 5: Time until a solver finds the best feasible solution for each formulation and for both shallow and deep neural networks on each particular fixing of the oil well problem. A * indicates that the solver could not prove the found solution optimal.

	Shallow			Deep		
Instance	MIP	MPCC	ReLU NN	MIP	MPCC	Embedded
1	**1.231**	**1.231**	**1.231**	NaN	**1.248**	1.183
2	**1.228**	**1.228**	**1.228**	NaN	**1.228**	1.155
3	**1.219**	**1.219**	**1.219**	NaN	1.229	**1.274**
4	**1.264**	**1.264**	**1.264**	NaN	**1.274**	1.255
5	**1.182**	**1.182**	**1.182**	NaN	1.192	**1.253**
6	**1.252**	**1.252**	**1.252**	NaN	1.256	**1.266**
7	**1.229**	**1.229**	**1.229**	NaN	**1.244**	1.215
8	**1.207**	**1.207**	**1.207**	NaN	1.201	**1.225**
9	**1.239**	**1.239**	**1.239**	NaN	1.249	**1.282**
10	**1.263**	**1.263**	**1.263**	NaN	**1.276**	1.243

Table 6: The objective value of the best feasible solution for each formulation for both shallow and deep neural networks on each particular fixing of the oil well problem. A NaN indicates that no feasible solution was found.

The results for the experiments performed with the binary variables fixed to arbitrary configurations are presented in Table 5 which details the solve times and Table 6 which details the objective values produced. We observe that overall, the MPCC formulation clearly outperforms all the other formulations in terms of speed in that it is able to find solutions in about 2 seconds where the embedded network formulations take 10 to 20 seconds and the MIP formulation can take much longer. In fact, the MIP formulation fails to find a single solution for any of the instances using the deep neural network configuration which suggests that optimization problems with deeper networks are more difficult than networks with shallow networks even if the number of nodes is the same.

In terms of objective value, for the shallow network, all formulations were able to find the global optimum in every instance relatively quickly. For the deep networks, the different formulations outperformed each other on different instances leaving no clear winner. The MPCC formulation solves each instance the quickest which is due to the relatively small size of each of the neural networks.

Unlike the other two problems, we observed for these instances that the embedded ReLU network formulation did not have trouble with convergence and in spite of nondifferentiability, the dual infeasibility was brought down to zero signifying convergence in all instances. We postulate that the reason for the success here as compared to the other instances may be due to the simplicity of the networks: 8 of the networks are single input and single output, and so the difficulties that might emerge in multiple dimensions do not appear.

6 Conclusions and Future Research

We have presented three alternative formulations of ReLU deep-neural network constraints as a mixed-integer problem, an optimization problem with complementarity constraints, and as a problem with the neural network directly embedded respectively. The MIP and MPCC formulations can be viewed as lifted formulations, and we have shown that the lifting convexifies optimization problems with deep neural network constraints in the case of the mixed-integer formulation. We have also presented a warmstart technique that uses training data of the neural network to construct good initial solutions. We have compared the three formulations on three examples arising in the design of engines, the design of images that "fool" a given classifier, and the assignment of flow in an oil well network. Each formulation has its advantages. We have shown the MIP formulation to be useful in finding an optimal solution but this comes at the cost of a particularly long solve time. We observed that the new complementarity constraint formulation generally outperforms the mixed-integer formulation in terms of solution time, but may not find the optimal solution although it often comes close. We also observed that the embedded neural network formulation has the advantage of being scalable and quick to solve, but has real difficulties with convergence related to the nondifferentiability of the ReLU activation function (which could be rectified using a smooth activation function like the Swish function). The experiments performed in this report portray the versatility of each of the different formulations in handling the breadth of optimization problems which may involve neural networks.

References

1. M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

2. S. M. Aithal and P. Balaprakash. MaLTESE: Large-scale simulation-driven machine learning for transient driving cycles. In *High Performance Computing*, pages 186–205, Cham, 2019. Springer International Publishing.

3. R. Anderson, J. Huchette, C. Tjandraatmadja, and J. P. Vielma. Strong mixed-integer programming formulations for trained neural networks. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 27–42, 2019.

4. P. Belotti. Couenne: A user's manual. Technical report, FICO, 2020.

5. J. Bolte and E. Pauwels. Conservative set valued fields, automatic differentiation, stochastic gradient methods and deep learning. *Mathematical Programming*, pages 1–33, 2020.

6. P. Bonami and J. Lee. BONMIN user's manual. *Numer Math*, 4:1–32, 2007.

7. N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. *arXiv preprint arXiv:1608.04644*, 2016.

8. C.-H. Cheng, G. Nührenberg, and H. Ruess. Maximum resilience of artificial neural networks. *arXiv preprint arXiv:1705.01040*, 2017.

9. M.-S. Cheon. An outer-approximation guided optimization approach for constrained neural network inverse problems. *arXiv preprint arXiv:2002.10404*, 2020.

10. F. H. Clarke. *Optimization and Nonsmooth Analysis*. John Wiley & Sons, Inc., New York, 1983. Reprint by SIAM, Philadelphia, 1990.

11. S. S. Du, X. Zhai, B. Poczos, and A. Singh. Gradient descent provably optimizes over-parameterized neural networks. In *International Conference on Learning Representations*, 2018.

12. I. Dunning, J. Huchette, and M. Lubin. Jump: A modeling language for mathematical optimization. *SIAM review*, 59(2):295–320, 2017.

13. I. Dunning, J. Huchette, and M. Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017.

14. S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari. Output range analysis for deep feedforward neural networks. In *NASA Formal Methods Symposium*, pages 121–138. Springer, 2018.

15. M. Fischetti and J. Jo. Deep neural networks and mixed integer linear optimization. *Constraints*, 23(3):296–309, 2018.

16. R. Fletcher and S. Leyffer. Solving mathematical program with complementarity constraints as nonlinear programs. *Optimization Methods and Software*, 19(1):15–40, 2004.

17. R. Fletcher, S. Leyffer, D. Ralph, and S. Scholtes. Local convergence of SQP methods for mathematical programs with equilibrium constraints. *SIAM Journal on Optimization*, 17(1):259—-286, 2006.

18. R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, 1993.

19. A. M. Gleixner, T. Berthold, B. Müller, and S. Weltge. Three enhancements for optimization-based bound tightening. *Journal of Global Optimization*, 67(4):731–757, 2017.

20. X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

21. I. J. Goodfellow, O. Vinyals, and A. M. Saxe. Qualitatively characterizing neural network optimization problems. *arXiv preprint arXiv:1412.6544*, 2014.

22. B. Grimstad and H. Andersson. Relu networks as surrogate models in mixed-integer linear programs. *Computers & Chemical Engineering*, 131:106580, 2019.

23. Gurobi Optimization, Inc. *Gurobi Optimizer Reference Manual, Version 5.0*, 2012.

24. K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

25. IBM Corp. *IBM Ilog CPLEX V12.1: User's Manual for CPLEX*, 2009.

26. G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

27. E. B. Khalil, A. Gupta, and B. Dilkina. Combinatorial attacks on binarized neural networks. *arXiv preprint arXiv:1810.03538*, 2018.

28. A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

29. Y. LeCun. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

30. S. Leyffer. Mathematical programs with complementarity constraints. *SIAG/OPT Views-and-News*, 14(1):15–18, 2003.

31. S. Leyffer, G. Lopez-Calva, and J. Nocedal. Interior methods for mathematical programs with complementarity constraints. *SIAM Journal on Optimization*, 17(1):52–77, 2006.

32. Y. Li and Y. Yuan. Convergence analysis of two-layer neural networks with relu activation. *Advances in Neural Information Processing Systems*, 30:597–607, 2017.

33. A. Mahajan, S. Leyffer, J. Linderoth, J. Luedtke, and T. Munson. MINOTAUR: a toolkit for solving mixed-integer nonlinear optimization. wiki-page, 2011. http://wiki.mcs.anl.gov/minotaur.

34. A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

35. M. Powell. A method for nonlinear constraints in minimization problems in optimization. In R. Fletcher, editor, *Optimization*. Academic Press, 1969.

36. N. V. Queipo, R. T. Haftka, W. Shyy, T. Goel, R. Vaidyanathan, and P. K. Tucker. Surrogate-based analysis and optimization. *Progress in aerospace sciences*, 41(1):1–28, 2005.

37. A. Raghunathan and L. T. Biegler. An interior point method for mathematical programs with complementarity constraints (MPCCs). *SIAM Journal on Optimization*, 15(3):720–750, 2005.

38. P. Ramachandran, B. Zoph, and Q. V. Le. Swish: a self-gated activation function. *arXiv preprint arXiv:1710.05941*, 7:1, 2017.

39. N. V. Sahinidis. BARON: A general purpose global optimization software package. *Journal of Global Optimization*, 8(2):201–205, 1996.

40. H. Scheel and S. Scholtes. Mathematical program with complementarity constraints: Stationarity, optimality and sensitivity. *Mathematics of Operations Research*, 25:1–22, 2000.

41. A. M. Schweidtmann and A. Mitsos. Deterministic global optimization with artificial neural networks embedded. *Journal of Optimization Theory and Applications*, 180(3):925–948, 2019.

42. T. Serra and S. Ramalingam. Empirical bounds on linear regions of deep rectifier networks. In *AAAI*, pages 5628–5635, 2020.

43. K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

44. C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

45. M. Tawarmalani and N. V. Sahinidis. *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming: Theory, Algorithms, Software, and Applications*. Kluwer Academic Publishers, Boston MA, 2002.

46. V. Tjeng, K. Xiao, and R. Tedrake. Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356*, 2017.

47. A. Wächter and L. T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.