An Outer-approximation Guided Optimization Approach for Constrained Neural Network Inverse Problems

Myun-Seok Cheon

Corporate Strategic Research,
ExxonMobil Research and Engineering

February 17, 2020

Abstract

This paper discusses an outer-approximation guided optimization method for constrained neural network inverse problems with rectified linear units. The constrained neural network inverse problems refer to an optimization problem to find the best set of input values of a given trained neural network in order to produce a predefined desired output in presence of constraints on input values. This paper analyzes the characteristics of optimal solutions of neural network inverse problems with rectified activation units and proposes an outer-approximation algorithm by exploiting their characteristics. The proposed outer-approximation guided optimization comprises primal and dual phases. The primal phase incorporates neighbor curvatures with neighbor outer-approximations to expedite the process. The dual phase identifies and utilizes the structure of local convex regions to improve the convergence to a local optimal solution. At last, computation experiments demonstrate the superiority of the proposed algorithm compared to a projected gradient method.

1 Introduction

Neural networks are the most essential and widely used ingredient of modern machine learning applications [Schmidhuber, 2015]. The primary reason of the prevalent usages is the flexibility and capability of approximating any continuous function, known as the *universal approximation theorem* [Hornik et al., 1990]. Deep neural networks have been demonstrating the strength of universal approximation capabilities in many machine learning applications including image processing, natural language processing and reinforcement learning problems. In the image classification problems, convolutional neural networks are commonly adopted to describe the spatial and temporal dependencies in images without any tailored feature engineering [Krizhevsky et al., 2012]. Recurrent neural networks are another example of the wide usage of analyzing temporal data [Hochreiter and Schmidhuber, 1997]. In reinforcement learning, various types of neural networks are employed to approximate intricate value and policy functions [Silver et al., 2017, Peters and Schaal, 2008].

Neural network inverse problems refer to a class of optimization problems that find a right set of input parameters to achieve a desired output with a trained neural network. The trained neural networks can be viewed as a surrogate function that predicts outputs of an input. For example, in the material design context, neural networks can be trained to predict material properties or performances with material fingerprints such as molecular structures. The inverse problem is to find the right molecular structure to produce a certain material property. The input parameters are the main optimization variables for the inverse problems whereas the forward problems optimize weights and biases of a neural network with a large amount of input and output data to improve the prediction accuracy.

1.1 Constrained neural network inverse problems

Let $f(\boldsymbol{x}) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ be a neural network function whose input and output dimensions are n and m, respectively. Let $\hat{\boldsymbol{f}} \in \mathbb{R}^m$ be a desired output. Let $\mathcal{L}(\cdot, \cdot) : \mathbb{R}^m \times \mathbb{R}^m \rightarrow \mathbb{R}$ be a function to measure the difference or loss between two vectors. Let $\mathcal{X}$ be the feasible space for input values. The constrained neural network inverse problem can be described as follows;

$$\text{(P)} \qquad \min_{x \in \mathcal{X}} \qquad \mathcal{L}(f(x), \hat{\boldsymbol{f}}) \tag{1.1}$$

The optimization problem is to find the input values $(\boldsymbol{x})$ that minimize the difference between the desired output $(\hat{\boldsymbol{f}})$ and the corresponding neural network output $(f(\boldsymbol{x}))$. For the sake of simplicity, let $g(\boldsymbol{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$ denote $\mathcal{L}(f(\boldsymbol{x}), \hat{f}) \in \mathbb{R}$ and $\nabla g(\boldsymbol{x}) \in \mathbb{R}^n$ be the gradient of $\mathcal{L}(f(\boldsymbol{x}), \hat{\boldsymbol{f}})$ at $\boldsymbol{x}$.

1.2 Potential applications

Adversarial examples for a trained neural network are crafted to increase robustness of neural networks. The main idea is from the observation that trained neural networks often make mistakes on classification or prediction tasks for images that have a small perturbation of the original image, which are indifferent to human eyes [Szegedy et al., 2014, Linden and Kindermann, 1989]. Such adversarial examples in the training set improve the robustness. Generating adversarial examples can be formulated as a neural network inverse problem. For a given image and the corresponding classification, the main decision variable of the problem is perturbations of the image and the main objective is to minimize the prediction accuracy to the corresponding classification. In this problem, the magnitude of perturbation can be controlled by constraints [Fischetti and Jo, 2018, Anderson et al., 2019].

The second example is any engineering and scientific analysis that consists of forward and inverse modeling. For example, material design and discovery activities can be described with two major parts; one is the forward (prediction) problem that predicts the property of a given material and the other is the inverse problem that finds a set of materials for a given desired property. With a large amount of data or simulation, deep neural networks can be trained to capture the core characteristics of the forward problem. The inverse problem with the trained neural network can be of the form problem (P). The problem optimizes the input, which is a choice of materials, to achieve a desired output such as a target property [Chen and Gu, 2020]. Similar structures of two phase approaches for engineering problems have been discussed in several literatures [Rezaee and Dadkhah, 2019, Corts et al., 2009]

Another motivating example is physics-based inversion problems with a lower-dimensional representation. The physics-based inverse problems are to find the right set of parameters for a physics model that explain observations. For example, X-ray tomography is a nondestructive method to analyze the internal property and structure of an object with X-ray measurements. Another example is the subsurface analysis in the oil and gas industries, which utilize seismic and gravity data to describe the subsurface characterization. One of main challenges on such analysis is the ill-posedness of the inverse problem such that the number of model parameters to fit are several orders of magnitude larger than the number of independent observations. This phenomenon causes overfitting such that the resulting model explains the observation data extremely well but it is not a physically plausible solution. One idea to circumvent the issue is incorporating additional information through machine learning techniques into the inversion process. The main role of the machine learning techniques such as variants of variational autoencoders is to capture the main characteristics of reasonable physical realizations with many prior examples. Variational autoencoders consist of an encoder that maps or abstracts physical realizations to latent space variables and a decoder that projects latent variables to a physical realization. The decoder of a successfully trained variational autoencoder can produce a plausible physical realization with any value of the latent space along with a probabilistic measure. In the new approach, the resulting problem is of the form a neural network inverse problem that combines the physics based inverse problem along with the trained decoder. The problem optimizes the latent space variables of the decoder to minimize the difference between the given observation and the physics response of the corresponding physics model from the decoder [O'Malley et al., 2019].

1.3 Prior solution methods

Solution techniques for neural network inverse problems can be categorized into three classes; one group is methods that utilize the gradient information such as steepest descent algorithms, projected gradient method, etc. The second group is methods based on the forward function calls such as derivative-free methods and meta-heuristics. The last group is based on explicit mathematical formulations such as mixed integer nonlinear programming.

The gradient respect to input parameters can be computed with back-propagation of neural networks [Linden and Kindermann, 1989]. Unconstrained problems, i.e., no additional constraints for the input parameters, can be tackled with the gradient based methods such as steepest descent methods, quasi-Newton methods, etc. For constrained problems, the projected gradient method can be considered.

Meta-heuristics such as particle swarm optimization or genetic algorithms can be considered. The meta-heuristics are attractive because of inexpensive forward evaluation of trained neural networks. Compared to gradient based approaches, these methods avoid trapping into a local optimum [Rezaee and Dadkhah, 2019].

When a neural network has only rectified linear activation units, the inverse problem can be described as a mixed integer nonlinear programming problem [Fischetti and Jo, 2018, Anderson et al., 2019]. Section 2.1 will discuss a mathematical model.

1.4 Contributions

This paper proposes an optimization algorithm in the context of a multi-start optimization framework for neural network inverse problems. The multi-start concept is employed to mitigate the local optimal issue of neural network inverse problems. In the framework, the initial optimization starting points are collected from the training set, e.g., n-closest data for a given target or n-clustering center points. There are two benefits of using the training set to select starting points. One is that it is easy to collect meaningful starting points with polynomial time examination. The other is that the solutions stay within the training region. Since neural networks are a surrogate function with a limited training data, if a solution is far from the training data region, the corresponding prediction might not be accurate. By using the trained data for starting solutions, the resulting solution might be retained within the comfort zone of neural networks. The multi-start optimization framework requires solving many optimization problems with various starting points. Therefore, it is important to have an efficient solution method.

The contributions of this paper are twofold; One is that the paper analyzes the characteristics of local optimal solutions of neural network inverse problems with rectified activation units; The second contribution is the development of an efficient algorithm exploiting the characteristics and a demonstration of the superiority compared to a gradient based method through computational experiments.

The characteristics of neural network inverse problems and the types of local optimal solutions are discussed in Section 2. In Section 3, an outer approximation based algorithm is proposed. Section 4 contains the computational results for the proposed method and a projected gradient method.

2 Characteristics of neural network inverse problems

2.1 Mathematical models

A neural network consists of multiple layers of interconnected neurons. Each neuron is a computing unit defined by weights, bias and an activation function. The weights and bias describe a linear relationship with outputs from connected neurons and the activation function, typically a nonlinear operator and applied after the linear computation, generates the final output. Equation (2.1) describes the computation in a neuron with a rectified linear activation function.

$$t_j = \max\left(0, \sum_i w_{ij} t_i + b_j\right), \tag{2.1}$$

where t_i is the output of neuron i and $w_{i,j}$ and b_j are the weights from neuron i to neuron j and the bias term for neuron j, respectively. In Equation (2.1), the rectified linear activation is described with the max operator. Equation (2.1) can be described as a mixed integer linear system as follows;

$$
\begin{aligned}
& t_j - s_j = \sum_i w_{ij} t_i + b_j, && \forall j \in N^r, && (2.2)\\
& t_j \le \mathbb{M} z_j, && \forall j \in N^r, && (2.3)\\
& s_j \le \mathbb{M}(1 - z_j), && \forall j \in N^r, && (2.4)\\
& t_j, s_j \ge 0, && \forall j \in N^r, && (2.5)\\
& z_j \in \{0, 1\}, && \forall j \in N^r, && (2.6)
\end{aligned}
$$

where parameter $\mathbb{M}$ denotes a big-M, which is a large number and set N^r denotes the neurons with a rectified linear activation function. The binary variable z_j indicates whether neuron j is active or not. When neuron j is active such as $z_j = 1$, only variable t_j can take a nonzero value. Otherwise, variable s_j can take a nonzero value. Only a positive output value (t_j) is passed to connected neurons.

A neural network can be described as a mixed integer linear system as follows. Let N be the all neurons and let N^I and N^O be the neurons in the input and output layers, respectively.

$$
\begin{aligned}
& (2.2) - (2.6) && && (2.7)\\
& t_j = x_j, && \forall j \in N^I, && (2.8)\\
& y_j = \sum_i w_{ij} t_i + b_j, && \forall j \in N^O, && (2.9)\\
& y_j \in \mathbb{R}, && \forall j \in N^O && (2.10)
\end{aligned}
$$

In the the mixed integer linear system, x_j's are the parameter for the input layer in Constraint (2.8) and the result is the solution of variable y_j's of the linear system for the output layer in Constraint (2.9).

The formulation can be tightened with an extended formulation or projected cuts from an extended formulation [Anderson et al., 2019].

2.2 Characteristics of local optimal solutions

In this section, we discuss the properties of the neural network inverse problem with rectified linear activation units. Figure 1 shows a simple example of a neural network and the squared error loss with a given target. The neural network has 5 neurons and its output is a piece-wise linear, which is depicted in the right graph ((a)) of Figure 1. Given a target output, the orange dotted line in (a) of Figure 1, the left graph of Figure 1 shows the squared error loss respect to input x.

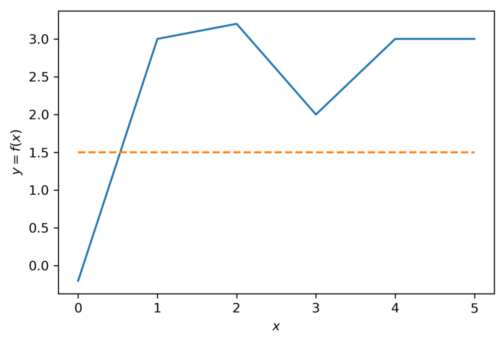
The first observation is that with a given activation such that a set of active neurons is predetermined, the resulting problem is convex. Let $\mathcal{N}(\boldsymbol{x})$ be a set of active neurons, whose output is positive for input $\boldsymbol{x}$. Note that for a given solution $\boldsymbol{x}$, there can be more than one active set definition such that $\mathcal{N}_1(\boldsymbol{x}) \neq \mathcal{N}_2(\boldsymbol{x})$.

Proposition 2.1. *If a neural network has only rectified linear activation units and a convex loss function such that $\mathcal{L}(\cdot, \hat{\boldsymbol{y}})$ is convex for a given target $\hat{\boldsymbol{y}}$, then, the neural network inverse problem $g(\boldsymbol{x})$ over $\boldsymbol{x} \in \{\boldsymbol{x}' | \mathcal{N}(\boldsymbol{x}') = \mathcal{N}(\hat{\boldsymbol{x}})\}$ for a given set of activation $\mathcal{N}(\hat{\boldsymbol{x}})$ is convex.*

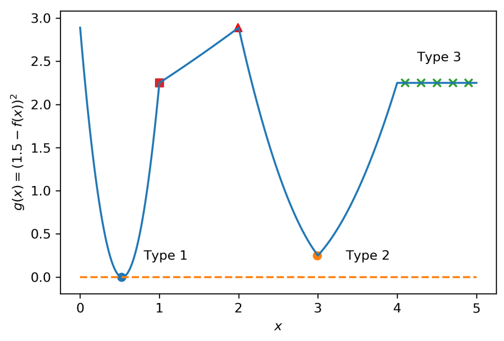
Proof. With a set of fixed activation $(\mathcal{N}(\hat{\boldsymbol{x}}))$, the feasible region of $\{\boldsymbol{x}' | \mathcal{N}(\boldsymbol{x}') = \mathcal{N}(\hat{\boldsymbol{x}})\}$ can be described with the following linear system by setting the binary variables of active neurons to one and the rest to zero.

$$
\mathcal{Y}(\mathcal{N}(\hat{\boldsymbol{x}})) \quad = \quad \left\{ \boldsymbol{x} \in \mathbb{R}^n \;\middle|\; \begin{array}{ll} t_j = x_j, & \forall j \in N^I \\ t_j = \sum_i w_{ij} t_i + b_j, & \forall j \in \mathcal{N}(\hat{\boldsymbol{x}}), \\ 0 \ge \sum_i w_{ij} t_i + b_j & \forall j \in N^r \setminus \mathcal{N}(\hat{\boldsymbol{x}}), \\ t_j \ge 0, & \forall j \in \mathcal{N}(\hat{\boldsymbol{x}}), \\ t_j = 0, & \forall j \in N^r \setminus \mathcal{N}(\hat{\boldsymbol{x}}) \end{array} \right\} \tag{2.11}
$$

Since the loss function is convex and the feasible region is convex, the resulting optimization problem is convex. □



(a) $f(\boldsymbol{x})$



(b) $g(\boldsymbol{x})$

Figure 1: An illustrative example of several types of local optimal solutions. (a) a neural network describes a piecewise linear. The blue line represents the output of the neural network over input x. Each integer interval has a unique activation pattern. The orange dotted line represents the target for the loss function; (b) the squared error loss function for $f(x)$ with a given target. The figure illustrates various local optimal solutions. The orange dotted line represents the zero loss.

Any feasible solution can be categorized into two groups based on the following set definition. Let $\mathcal{B}(\boldsymbol{x})$ be a set of neurons whose output after the linear computation, i.e., before the activation operation, is exactly zero for input $\boldsymbol{x}$ such as

$$\mathcal{B}(\boldsymbol{x}) = \left\{ j \in N^r \,\middle|\, \sum_i w_{ij}\tilde{t}_i + b_j = 0 \right\}, \tag{2.12}$$

where $\tilde{t}_i$ is the corresponding output of neuron i for input $\boldsymbol{x}$. One group is the solutions without any zero output neurons before the activation such that $\mathcal{B}(\boldsymbol{x}^*) = \emptyset$. The other group is at least one neuron with exactly zero output before the activation such that $\mathcal{B}(\boldsymbol{x}^*) \neq \emptyset$. Note that any solution is categorized into one of two categories such that $\{x|\mathcal{B}(x) = \emptyset \wedge \mathcal{B}(x) \neq \emptyset\} = \emptyset$ and $\{x \in \mathcal{X}|\mathcal{B}(x) = \emptyset \vee \mathcal{B}(x) \neq \emptyset\} = \mathcal{X}$. In Figure 1, any integer solution $x \in \{1, 2, 3, 4\}$ has at least one zero output neuron. Set $\mathcal{B}(x)$ for all other real numbers $x \in \mathbb{R} \setminus \{1, 2, 3, 4\}$ is empty.

The neural network inverse problem with rectified linear units can be viewed as a union of many convex optimization problems with all possible activation permutations.

Definition 2.2. (Type 1) An optimal solution $\boldsymbol{x}^*$ is called a *self-contained* local optimal solution if the solution $\boldsymbol{x}^*$ is optimal to the neural network inverse problem $g(\boldsymbol{x})$ over $\boldsymbol{x} \in \mathcal{Y}(\mathcal{N}(\boldsymbol{x}^*))$ and no neuron has an output of exact zero before the corresponding activation such as $\mathcal{B}(\boldsymbol{x}^*) = \emptyset$.

'Type 1' solution in Figure 1 is an example of a *self-contained* local optimal solution. In this case, it is sufficient to prove the local optimality without considering its neighbor activation regions. If there exist zero output neurons such as $\sum_i w_{ij}\tilde{t}_i + b_j = 0$, there exists more than one activation pattern that contains solution $\boldsymbol{x}^*$. In this case, it is required to check if the solution is optimal respect to all possible permutations of $\mathcal{N}(\boldsymbol{x}^*)$. Let $\mathcal{M}(\boldsymbol{x}^*)$ be a set of all feasible activation permutations for solution $\boldsymbol{x}^*$. Note that some permutations of activations can be infeasible.

Definition 2.3. (Type 2) An optimal solution $\boldsymbol{x}^*$ is called a *boundary* local optimal solution if the solution $\boldsymbol{x}^*$ is optimal to the neural network inverse problem $g(\boldsymbol{x})$ over $\boldsymbol{x} \in \{\boldsymbol{x}'|\mathcal{N}(\boldsymbol{x}') \in \mathcal{M}(\boldsymbol{x}^*)\}$ and the solution is associated with more than one activation pattern.

In this case, there exists at least one neuron with exactly zero output such as $\mathcal{B}(\boldsymbol{x}^*) \neq \emptyset$. 'Type 2' solution in Figure 1 falls into this category. When $x = 3$, there are two possible activations - one is for the interval from 2 to 3 and the other is for the interval from 3 to 4. In both cases, the solution ($x = 3$) is locally optimal.

Conversely, consider the solution of $x = 1$, which is a red square in Figure 1. The solution $x = 1$ is a local optimal respect to the convex region defined by the activation for $1 \leq x \leq 2$ while it is not local optimal for the convex region for the activation $0 \leq x \leq 1$.

Definition 2.4. (Type 3) An optimal solution $\boldsymbol{x}^*$ is called a *redundant* local optimal solution if an element x_j of the solution $\boldsymbol{x}^*$ is insensitive to the local optimality such that solutions with any value x_j within the feasible region $\boldsymbol{x} \in \{\boldsymbol{x}' | \mathcal{N}(\boldsymbol{x}') = \mathcal{N}(\boldsymbol{x}^*)\}$ are still locally optimal.

Note that the *redundant* local optimal solutions can be categorized as a *self-contained* or *boundary* local optimal solution. 'Type 3' solution in Figure 1 is an example of *redundant* local optimal solutions. The solution is similar to a saddle point in nonlinear optimization problems. Note that the solution ($x = 4$) at the boundary of 'Type 3' solution in Figure 1 is not a local optimal solution. Its left neighbor has better objective values.

Theorem 2.5. *A solution ($\boldsymbol{x}^*$) is locally optimal to Problem (P) if and only if the solution ($\boldsymbol{x}^*$) is locally optimal for all neighbor activation subproblems such that*

$$g(\boldsymbol{x}^*) = \min_{\boldsymbol{x} \in \mathcal{X} \cap \mathcal{Y}(\mathcal{N}(x^*))} g(\boldsymbol{x}) \qquad \forall \mathcal{N}(\boldsymbol{x}^*) \in \mathcal{M}(\boldsymbol{x}^*). \tag{2.13}$$

Proof. Suppose that a solution ($\boldsymbol{x}^*$) is locally optimal respect to all subproblems defined by its neighbor activation patterns and it is not locally optimal to Problem (P). Then, for any $\epsilon > 0$, there exists a solution $\hat{\boldsymbol{x}}$ such that $\|\boldsymbol{x}^* - \hat{\boldsymbol{x}}\| < \epsilon$, $g(\boldsymbol{x}^*) > g(\hat{\boldsymbol{x}})$ and $\boldsymbol{x}^* \notin \mathcal{Y}(\mathcal{N}(\hat{\boldsymbol{x}}))$. Since $\mathcal{Y}(\mathcal{N}(\hat{\boldsymbol{x}}))$ is a closed set (a linear system), there exists $0 < \lambda < 1$ such that $(1 - \lambda) \times \boldsymbol{x}^* - \lambda \times \hat{\boldsymbol{x}} \notin \mathcal{Y}(\mathcal{N}(\hat{\boldsymbol{x}}))$. That is, $\hat{\boldsymbol{x}}$ is not a neighbor. It contradicts.

Suppose that there is an activation pattern $\mathcal{N}(x^*)$ whose corresponding neighbor convex region contains a better solution ($\bar{\boldsymbol{x}}$) such as

$$g(\boldsymbol{x}^*) > g(\bar{\boldsymbol{x}}) = \min_{\boldsymbol{x} \in \mathcal{X} \cap \mathcal{Y}(\mathcal{N}(x^*))} g(\boldsymbol{x}). \tag{2.14}$$

Since $g(\boldsymbol{x})$ is convex, the following inequality holds for $0 < \lambda \leq 1$;

$$g((1 - \lambda) \times \boldsymbol{x}^* + \lambda \times \bar{\boldsymbol{x}}) \leq (1 - \lambda) \times g(\boldsymbol{x}^*) + \lambda \times g(\bar{\boldsymbol{x}}) < g(\boldsymbol{x}^*). \tag{2.15}$$

It implies that there exists a neighbor solution $\bar{\boldsymbol{x}}$ that has a better objective value. Thus, $\boldsymbol{x}^*$ is not a local optimal solution. $\square$

3 Outer approximation guided algorithm

In this section, an outer-approximation guided algorithm is proposed for constrained neural network inverse problems. The proposed algorithm adapts core concepts from gradient based algorithms and outer approximation approaches [Geoffrion, 1970] while it exploits the characteristics of local optimal solutions. The proposed algorithm iteratively identifies a descent direction with outer approximation subproblems and determines the next solution with a step size.

The proposed algorithm comprises two phases - primal and dual phases; The primal phase is to improve the solution by incorporating local (neighbor) gradients and the dual phase is focusing on proving local optimality. The algorithm uses two outer approximation subproblems to find a descent direction for primal and dual phases and one outer approximation subproblem to prove local optimality.

3.1 Outer approximation algorithms

The outer approximation algorithms are widely used to solve large scale convex optimization and convex mixed integer programming problems[Geoffrion, 1970, Benders, 1962, Duran and Grossmann, 1986]. The main idea of the algorithms is to approximate nonlinear convex or complex linear systems with a set of hyperplanes. Consider the following optimization problem;

$$\min_{x \in \mathcal{X}} \quad g(x), \tag{3.1}$$

where $g(x)$ is convex over $x \in \mathcal{X}$. Then, we can approximate the problem with hyperplanes as follows;

$$\min_{x \in \mathcal{X}, v \in \mathbb{R}} \quad v \tag{3.2}$$
$$\text{s.t.} \quad v \geq g(x^k) + \nabla g(x^k)^T (x - x^k) \qquad k \in \mathcal{K}, \tag{3.3}$$

where x^k is feasible such that $x^k \in \mathcal{X}$ and $\nabla g(x^k)$ is the gradient at x^k. When $g(\cdot)$ is a certain linear system, the dual values can be used instead of gradients [Benders, 1962]. Similar approaches have been proposed and applied for large-scale nonsmooth convex optimization problems [Ben-Tal and Nemirovski, 2005].

The outer-approximation is valid only when function $g(x)$ is convex over the feasible set $\mathcal{X}$. One can easily show that the neural network functions with rectified linear activation functions are not convex. Even though the approximation is not valid, it can be used to determine a descent direction. The approximation will provide additional information from previous observations compared to simple first order methods. In addition, if the final solution is within a localized convex region, the corresponding approximation on the region is valid.

3.2 Outer approximation for neural network inverse problems

In this section, we discuss an outer approximation model for the neural network inverse problems. Let $\mathcal{K}$ be an index set for solutions. Let set $\mathcal{K}(\boldsymbol{x}')$ be an index subset for solutions $(\boldsymbol{x}^k)$ in the union of convex feasible regions defined by activation patterns of solution $\boldsymbol{x}'$ such that

$$\mathcal{K}(\boldsymbol{x}') \quad = \quad \{k \in \mathcal{K} | \exists \mathcal{N}(\boldsymbol{x}^k) \in \mathcal{M}(\boldsymbol{x}')\}. \tag{3.4}$$

Let $\boldsymbol{x}^*$ be the best solution among solutions $\boldsymbol{x}^k$'s such that $\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}^k, k \in \mathcal{K}} g(\boldsymbol{x}^k)$. Consider an outer approximation model for neural network inverse problems.

$$(\text{NOA}) \qquad \min_{\boldsymbol{x} \in \mathcal{X}, v \in \mathbb{R}} \quad v \tag{3.5}$$
$$\text{s.t.} \quad v \geq g(\boldsymbol{x}^k) + \nabla g(\boldsymbol{x}^k)^T (\boldsymbol{x} - \boldsymbol{x}^k), \qquad k \in \mathcal{K}(\boldsymbol{x}^*). \tag{3.6}$$

Note that $g(\boldsymbol{x}^*)$ is not differentiable when $\mathcal{B}(\boldsymbol{x}^*) \neq \emptyset$. Let $\nabla_{\mathcal{N}(\cdot)} g(\boldsymbol{x}^*)$ be the gradient of function $g(\cdot)$ at solution $\boldsymbol{x}^*$ in a specific activation pattern $\mathcal{N}(\cdot)$. With an activation pattern $\mathcal{N}(\cdot)$, the resulting problem such as minimizing $g(\boldsymbol{x})$ over $\boldsymbol{x} \in \mathcal{Y}(\mathcal{N}(\cdot))$ is convex and differentiable for any feasible solution.

Proposition 3.1. *Let $\boldsymbol{x}^*$ be a solution in the convex region defined by an activation pattern $\mathcal{N}(\boldsymbol{x}^*)$. The solution $\boldsymbol{x}^*$ is locally optimal for the convex region $\mathcal{Y}(\mathcal{N}(\boldsymbol{x}^*))$ if and only if the following inequality holds;*

$$\nabla_{\mathcal{N}(\cdot)} g(\boldsymbol{x}^*)^T (\boldsymbol{x} - \boldsymbol{x}^*) \geq 0, \qquad \forall \boldsymbol{x} \in \mathcal{Y}(\mathcal{N}(\boldsymbol{x}^*)). \tag{3.7}$$

Proof. Let $\boldsymbol{x}^*$ be a local optimal solution within convex region $\mathcal{Y}(\mathcal{N}(\boldsymbol{x}^*))$. It implies that there is no feasible descent direction $\boldsymbol{d} = \alpha(\boldsymbol{x} - \boldsymbol{x}^*)$ for any feasible $\boldsymbol{x} \in \mathcal{Y}(\mathcal{N}(\boldsymbol{x}^*))$ and a scalar $\alpha > 0$. That is, the inequality (3.7) is valid for all $\boldsymbol{x} \in \mathcal{Y}(\mathcal{N}(\boldsymbol{x}^*))$.

Conversely, suppose that the inequality (3.7) is valid for all $\boldsymbol{x} \in \mathcal{Y}(\mathcal{N}(\boldsymbol{x}^*))$ and $\boldsymbol{x}^*$ is not local optimal. Then, there exists at least a solution $\hat{\boldsymbol{x}}$ such that $g(\boldsymbol{x}^*) > g(\hat{\boldsymbol{x}})$. Since the problem is convex, the following outer-approximation is always valid.

$$\begin{aligned} & g(\hat{\boldsymbol{x}}) \geq g(\boldsymbol{x}^*) + \nabla_{\mathcal{N}(\cdot)} g(\boldsymbol{x}^*)^T (\hat{\boldsymbol{x}} - \boldsymbol{x}^*), \\ \Rightarrow \quad & 0 > g(\hat{\boldsymbol{x}}) - g(\boldsymbol{x}^*) \geq \nabla_{\mathcal{N}(\cdot)} g(\boldsymbol{x}^*)^T (\hat{\boldsymbol{x}} - \boldsymbol{x}^*). \end{aligned}$$

It contradicts that the inequality (3.7) is valid. Therefore, $\boldsymbol{x}^*$ is local optimal. □

Theorem 3.2. *Let v^* be the optimal objective value of the outer approximation (NOA) with solutions $\boldsymbol{x}^k \in \mathcal{X}, k \in \mathcal{K}$ and $\boldsymbol{x}^*$ be the best known solution respect to function $g(\boldsymbol{x})$ such that $\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}^k, k \in \mathcal{K}} g(\boldsymbol{x}^k)$. The solution $\boldsymbol{x}^*$ is local optimal to Problem (P) if the following conditions hold;*

1. $g(\boldsymbol{x}^*) = v^*$*,*

2. $\nabla_{\mathcal{N}(\cdot)} g(\boldsymbol{x}^*)^T(\boldsymbol{x} - \boldsymbol{x}^*) \geq 0, \forall \boldsymbol{x} \in \mathcal{Y}(\mathcal{N}(\cdot)), \mathcal{N}(\cdot) \in \mathcal{M}(\boldsymbol{x}^*)$*.*

Proof. Consider two cases; one is $\mathcal{B}(\boldsymbol{x}^*) = \emptyset$ and the other is $\mathcal{B}(\boldsymbol{x}^*) \neq \emptyset$.

i. $\mathcal{B}(\boldsymbol{x}^*) = \emptyset$
Consider the feasible set (2.11). Since $\mathcal{B}(\boldsymbol{x}^*) = \emptyset$, the following two constraints of (2.11) are non-binding;

$$0 \geq \sum_i w_{ij} t_i + b_j, \qquad \forall j \in N^r \setminus \mathcal{N}(\hat{x}), \tag{3.8}$$

$$0 \leq t_j = \sum_i w_{ij} t_i + b_j, \qquad \forall j \in \mathcal{N}(\hat{x}). \tag{3.9}$$

The rest of (2.11) is redundant to function $g(\boldsymbol{x})$. Therefore, the outer approximation (3.6) is a binding outer approximation for $g(\boldsymbol{x})$ around solution $\boldsymbol{x}^*$. It means that $\boldsymbol{x}^*$ is local optimal and *self-contained* local optimal.

ii. $\mathcal{B}(\boldsymbol{x}^*) \neq \emptyset$
In this case, it is required to show that the current solution ($\boldsymbol{x}^*$) is also local optimal to its neighbor activations ($\mathcal{M}(\boldsymbol{x}^*)$). By Proposition 3.1, the second condition ensures that the solution is local optimal for feasible regions defined by the corresponding activation patterns.

□

For a given activation pattern $\mathcal{N}(\cdot)$, the following linear programming problem can check if the corresponding feasible region has a descent direction.

$$(\mathrm{OC}(\mathcal{N}(\cdot))) \qquad \min_{\boldsymbol{x} \in \mathcal{X}} \quad \nabla_{\mathcal{N}(\cdot)} g(\boldsymbol{x}^*)^T(\boldsymbol{x} - \boldsymbol{x}^*) \tag{3.10}$$

$$\text{s.t.} \quad \boldsymbol{x} \in \mathcal{Y}(\mathcal{N}(\cdot)) \tag{3.11}$$

If the optimal objective is equal to zero, then it proves that there is no descent direction.

3.3 Localized outer approximation subproblems

The proposed algorithm consists of primal and dual phases. The primal phase is designed to find a better solution quickly while the focus of the dual phase is to prove the local optimality of the best known solution. The dual phase employs the outer approximation model (NOA) discussed in Section 3.2. In order to identify the proper hyperplanes, the (NOA) outer approximation models require bookkeeping efforts of activation patterns for each solution. In order to alleviate the bookkeeping efforts in the early stage, the primal phase uses an outer approximation constructed based on the simple distance to the best known solution.

3.3.1 Distance-localized outer approximation subproblem

Subproblem (DLOA^k) at iteration k is defined as follows;

$$(\mathrm{DLOA}^k) \qquad \min_{\boldsymbol{x} \in \mathcal{X}, v \in \mathbb{R}} \quad v \tag{3.12}$$

$$\text{s.t.} \quad v \geq g(\boldsymbol{x}^{k'}) + \nabla g(\boldsymbol{x}^{k'})^T(\boldsymbol{x} - \boldsymbol{x}^{k'}) \qquad k' \in \mathcal{K}^* \cup \{k\}, \tag{3.13}$$

where set $\mathcal{K}^*$ is an index subset of hyperplanes that satisfy the following two conditions. One is the base solutions for the selected hyperplanes should be close to the best known solution such that the Euclidean distance from each base solution for hyperplanes to the best known solution should be less than γ^c. The second condition is that the resulting hyperplane should not exclude the best feasible solution. The set $\mathcal{K}^*$ is defined as follows;

$$\mathcal{K}^* \quad = \quad \left\{ k \in \mathcal{K} \;\middle|\; \begin{array}{l} g(\boldsymbol{x}^*) < g(\boldsymbol{x}^k) + \nabla g(\boldsymbol{x}^k)^T(\boldsymbol{x}^* - \boldsymbol{x}^k), \\ \|\boldsymbol{x}^* - \boldsymbol{x}^k\| \leq \gamma^c \end{array} \right\}. \tag{3.14}$$

Note that since the algorithm drops previously generated outer approximation constraints, it can experience a *stalling* behavior such that the algorithm could revisit the solution that is previously examined. Such behaviors are prevented with two mechanism. The first mechanism is to always include the latest hyperplane even if it cuts off the best known solution. The other mechanism is adjusting the step size. If the algorithm fails to find a better solution, it decreases the step size, which guarantees a different solution. Suppose that the hyperplane defined by the latest solution cuts off the best feasible solution. There are two possible outcomes. One is that the new approximation identifies a better solution. In this case, there is no stalling problem. The other case is that the new approximation fails to identify a new solution. In this case, the newly added hyperplanes will be removed with the validity check against the best known solution. It can lead to the same approximation model as before. However, since the algorithm fails to find a better solution, it will reduce the step size, which prevents the *stalling* behavior.

3.3.2 Region-localized outer approximation subproblem

The following subproblems are employed in the dual phase to expedite proving the local optimality. Let $\mathcal{N}(\cdot) \in \mathcal{M}(\boldsymbol{x}^*)$ be a selected activation pattern.

$$\begin{aligned} (\text{RLOA}^k(\mathcal{N}(\cdot))) \qquad & \min_{\boldsymbol{x}\in\mathcal{X}, v\in\mathbb{R}} \quad v & & \qquad (3.15) \\ & \text{s.t.} \quad v \geq g(\boldsymbol{x}^{k'}) + \nabla g(\boldsymbol{x}^{k'})^T(\boldsymbol{x} - \boldsymbol{x}^{k'}) & \forall \boldsymbol{x}^{k'} \in \mathcal{Y}(\mathcal{N}(\cdot)), & \qquad (3.16) \\ & \qquad\;\; 0 \geq \boldsymbol{r}^{k'^T}\boldsymbol{x} + d^{k'} & \forall \boldsymbol{x}^{k'} \notin \mathcal{Y}(\mathcal{N}(\cdot)), & \qquad (3.17) \end{aligned}$$

where Constraint (3.17) are feasibility cuts under consideration of the convex region $\mathcal{Y}(\mathcal{N}(\cdot))$. Since the set $\mathcal{Y}(\mathcal{N}(\cdot))$ is a linear system, a feasibility cut for solution $\boldsymbol{x}^{k'} \notin \mathcal{Y}(\mathcal{N}(\cdot))$ can be constructed based on the extreme ray of the following dual problem.

$$\begin{aligned} \max_{r,\pi} \quad & \sum_{j\in N^I} x_j^{k'} r_j + \sum_{j\in N\setminus N^I} b_j \pi_j & & (3.18) \\ & \sum_{j\in N^I} x_j^{k'} r_j + \sum_{j\in N\setminus N^I} b_j \pi_j \leq 1, & & (3.19) \\ & r_j - \sum_{k\in N} w_{jk}\pi_k = 0, & \forall j \in N^I, \quad & (3.20) \\ & \pi_j - \sum_{k\in N} w_{jk}\pi_k \leq 0, & \forall j \in \mathcal{N}(\cdot), \quad & (3.21) \\ & r_j \in \mathbb{R}, & \forall j \in N^I, \quad & (3.22) \\ & \pi_j \in \mathbb{R}, & \forall j \in \mathcal{N}(\cdot), \quad & (3.23) \\ & \pi_j \geq 0, & \forall j \in N^r \setminus \mathcal{N}(\cdot), \quad & (3.24) \\ & \pi_j = 0, & \forall j \in N^O \quad & (3.25) \end{aligned}$$

When the neural network has many nodes, solving the aforementioned dual problem can be challenging and often numerically unstable. In order to overcome numerical and computational challenges, the proposed algorithm uses forward-propagation to populate Constraint (3.17).

Assume that nodes closer to the input layer are assigned with lower index numbers than ones further away. Let j^d be the lowest indexed node with activation discrepancy between the two solutions such that $j^d = \min(j \in \mathcal{N}^r | (j \in \mathcal{N}(\boldsymbol{x}^{k'}) \wedge j \notin \mathcal{N}(\cdot)) \vee (j \notin \mathcal{N}(\boldsymbol{x}^{k'}) \wedge j \in \mathcal{N}(\cdot)))$. Let L be the set of layers. Let n^l be the number of nodes in layer $l \in L$. Let $W^l \in \mathbb{R}^{n^l \times n^{l-1}}$ and $b^l \in \mathbb{R}^{n^l}$ be the weight matrix and the bias column vector for layer $l \in L$, respectively. Let $\boldsymbol{I}^l \in \mathbb{R}^{n^l \times n^l}$ be a matrix whose diagonal entries corresponding to active nodes in the activation pattern $\mathcal{N}(\cdot)$ are one, otherwise zero;

$$I^l_{i,j} = \begin{cases} 1, & \forall i = j, \nu_i \in \mathcal{N}(\cdot), \\ 0, & \text{otherwise,} \end{cases} \tag{3.26}$$

where $I^l_{i,j}$ denotes j^{th} element in i^{th} row of $\boldsymbol{I}^l$ and ν_i denotes the node index corresponding to i^{th} row.

Suppose $\boldsymbol{x}$ is not in the same activation region $\mathcal{N}(\cdot)$ such that $\mathcal{N}(\boldsymbol{x}) \neq \mathcal{N}(\cdot)$. With a known activation pattern, the max operation can be described with matrix $\boldsymbol{I}^l$. The first layer of forward propagation can be computed as follows;

$$\boldsymbol{I}^1[\boldsymbol{W}^1\boldsymbol{x} + \boldsymbol{b}^1] \tag{3.27}$$

Let l^{j_d} be the layer for node j_d. Then, the forward propagation up to node j_d can be described as follows;

$$\widehat{\boldsymbol{W}} = \prod_{l=l^{j_d}}^{1} \boldsymbol{I}^l\boldsymbol{W}^l, \tag{3.28}$$

$$\hat{\boldsymbol{b}} = \boldsymbol{I}^{l^{j_d}}\boldsymbol{b}^{l^{j_d}} + \sum_{l=l^{j_d}-1}^{1} \left(\prod_{l'=l+1}^{l^{j_d}} \boldsymbol{I}^{l'}\boldsymbol{W}^{l'} \right) \boldsymbol{I}^l\boldsymbol{b}^l \tag{3.29}$$

Now, the feasibility constraint can be constructed as follows; suppose i^d be i^{th} row of the corresponding output to discrepancy node j^d such that $\nu_{i^d} = j^d$. Let $\widehat{\boldsymbol{W}}_i$ and $\hat{\boldsymbol{b}}_i$ denote i^{th} row of matrix $\widehat{\boldsymbol{W}}$ and $\hat{\boldsymbol{b}}$, respectively.

$$\widehat{\boldsymbol{W}}_{i^d}\boldsymbol{x} + \hat{\boldsymbol{b}}_{i^d} \leq \boldsymbol{0} \quad \text{if } j^d \notin \mathcal{N}(\cdot), \tag{3.30}$$

$$\widehat{\boldsymbol{W}}_{i^d}\boldsymbol{x} + \hat{\boldsymbol{b}}_{i^d} \geq \boldsymbol{0} \quad \text{if } j^d \in \mathcal{N}(\cdot), \tag{3.31}$$

By the construction of j^d, solution $\boldsymbol{x}^{k'}$ violates one of the constraints above.

3.4 Outer approximation guided algorithm

Algorithm 3.1 describes the outer approximation guided algorithm. The algorithm starts with an initial point $\boldsymbol{x}^0$ and hyperparameters in line 2 - 4. Parameter γ^s, $\gamma^s_{\min}$, and $\gamma^s_{\max}$ denote the step size and its lower and upper bounds, respectively. Parameter γ^c defines the neighborhood size of distance-localized outer approximation subproblems. Parameter ρ^c and ρ^e are decrease and increase ratios for the step size, respectively. Parameter ϵ is the local optimality gap for termination and parameter N defines the maximum number of iterations. From line 5 to line 7, it initializes the bookkeeping parameters; $\bar{x}^*$ and $\bar{g}^*$ denote the best known solution and the corresponding objective value, respectively. Parameter k tracks the number of iteration and parameter dual-phase takes a boolean value to indicate whether the algorithm is in the dual phase or not.

In each iteration, the algorithm evaluates the function value and the gradient for the current solution $\boldsymbol{x}^k$ and updates the corresponding constraint for subproblems (line 9 - 10). In line 11 to 17, if the algorithm finds a better solution than the current best known solution, it updates the best known solution and increases the step size with parameter ρ^e. When there is no improvement, it reduces the step size with parameter ρ^c. The step size is truncated by predetermined parameters $\gamma^s_{\min}$ and $\gamma^s_{\max}$. Line 18 updates the iteration count. If the step size is too small (Line 19), the algorithm sets to the dual phase. When the algorithm is not in the dual phase, it solves the distance-localized outer approximation subproblem to find the next direction. If the objective of the approximation is close to the objective value of the best known solution, it reevaluates the approximation with solutions within the neighbor activation regions. If the resulting approximation is close to the best known objective within ϵ, the algorithm activates the dual phase. In the dual phase, the algorithm terminates if the best known solution is $\epsilon-$optimal for all possible neighbor feasible region. If the algorithm finds a neighbor region with potential improvement, it generates a next direction with the solution

Algorithm 3.1 Outer approximation guided algorithm

```
1: $\boldsymbol{x}^0 \in \mathcal{X}$
2: Define $\gamma^s > 0, \gamma^s_{\min} > 0, \gamma^s_{\max} > 0$
3: Define $\gamma^c > 0, 0 < \rho^c < 1, \rho^e > 1$
4: Define $\epsilon > 0, N > 0$
5: $\bar{\boldsymbol{x}}^* \leftarrow \boldsymbol{x}^0$, $\bar{g}^* \leftarrow g(\boldsymbol{x}^0)$
6: $k \leftarrow 0$
7: dual-phase $\leftarrow false$
8: for $k \in \{0, \dots, N\}$ do
9:     compute $g(\boldsymbol{x}^k)$ and $\nabla g(\boldsymbol{x}^k)$
10:    update constraints
11:    if $g(\boldsymbol{x}^k) < \bar{g}^*$ then
12:        $\gamma^s \leftarrow \min(\rho^e \times \gamma^s, \gamma^s_{\max})$
13:        dual-phase $\leftarrow false$
14:        $\bar{\boldsymbol{x}}^* \leftarrow \boldsymbol{x}^k$, $\bar{g}^* \leftarrow g(\boldsymbol{x}^k)$
15:    else
16:        $\gamma^s \leftarrow \max(\gamma^s_{\min}, \rho^c \times \gamma^s)$
17:    end if
18:    $k \leftarrow k + 1$
19:    if $\gamma^s = \gamma^s_{\min}$ then
20:        dual-phase $\leftarrow true$
21:    end if
22:    if $\neg$ dual-phase then
23:        $\hat{\boldsymbol{x}}^k$, $v^k \leftarrow$ solve (DLOA$^k$)
24:        if $\bar{g}^* - v^k < \epsilon$ then
25:            $\hat{\boldsymbol{x}}^k$, $v^k \leftarrow$ solve (NOA)
26:            if $\bar{g}^* - v^k < \epsilon$ then
27:                dual-phase $\leftarrow true$
28:            end if
29:        end if
30:    end if
31:    if dual-phase then
32:        $\hat{\boldsymbol{x}}^k \leftarrow \emptyset$
33:        for $\mathcal{N}(\cdot) \in \mathcal{M}(\boldsymbol{x}^*)$ do
34:            $\hat{\boldsymbol{x}}^k$, $v^k \leftarrow$ solve (RLOA$^k(\mathcal{N}(\cdot))$)
35:            if $\bar{g}^* - v^k \geq \epsilon$ then
36:                break
37:            end if
38:        end for
39:        if $\hat{\boldsymbol{x}}^k = \emptyset$ then
40:            break
41:        end if
42:    end if
43:    $\boldsymbol{x}^k = \bar{\boldsymbol{x}}^* + \gamma^s \times (\hat{\boldsymbol{x}}^k - \bar{\boldsymbol{x}}^*)$
44: end for
```

of (RLOA$^k(\mathcal{N}(\cdot))$). In Line 43, the algorithm sets the next solution under consideration of the approximation solution $\hat{\boldsymbol{x}}^k$ and the step size parameter γ^s.

Set $\mathcal{M}(\boldsymbol{x}^*)$ in Line 33 can have exponentially many activation patterns. In the implementation, the algorithm records the previously examined activation patterns to avoid reexamining same regions, repeatably. The algorithm refreshes the record when it finds a new best solution. In the examination of neighbor patterns, the algorithm filters the activation patterns with the following inequality in a preliminary way.

$$\nabla_{\mathcal{N}(\cdot)} g(\boldsymbol{x}^*)^T (\boldsymbol{x}^k - \boldsymbol{x}^*) < 0, \boldsymbol{x}^k \in \mathcal{Y}(\mathcal{N}(\cdot)), \mathcal{N}(\cdot) \in \mathcal{M}(\boldsymbol{x}^*) \tag{3.32}$$

For each activation pattern $\mathcal{N}(\cdot) \in \mathcal{M}(\boldsymbol{x}^*)$, the algorithm checks whether the direction to all previous observations within the corresponding feasible region is a descent direction or not. If it is a descent direction, the algorithm solves RLOA$^k(\mathcal{N}(\cdot))$ in Line 34. Even if the algorithm fails to find an activation pattern that satisfies the condition (3.32), it does not mean that the algorithm identifies an $\epsilon-$optimal solution. The algorithm still needs to solve RLOA$^k(\mathcal{N}(\cdot))$ in Line 34 for all possible activation patterns. The main purpose of the preliminary check is to quickly find a region with higher potential of improvement.

It is very challenging to find an exact solution at the boundary. Therefore, in the implementation, the boundary solutions are determined with a numerical tolerance $\tau > 0$ as follows;

$$\hat{\mathcal{B}}(\boldsymbol{x}) = \left\{ j \in N^r \middle| \left| \sum_i w_{ij} \tilde{t}_i + b_j \right| \leq \tau \right\}. \tag{3.33}$$

3.5 Convergence

Theorem 3.3. *If $\mathcal{X}$ is compact, Algorithm 3.1 will converge to an $\epsilon-$local optimal solution as $N \to \infty$.*

Proof. By the design of the algorithm, it only checks the optimality condition with hyperplanes within the corresponding convex region respect to the best known solution. Since $\mathcal{L}(\cdot, \hat{f})$ is convex and $\mathcal{X}$ is compact, the algorithm converges to an $\epsilon-$local optimal[Kelley, 1960]. ☐

Note that the feasible set $\mathcal{X}$ does not need to be convex in order to prove the local optimality [Geoffrion, 1972, Eaves and Zangwill, 1971]. The current representation of the algorithm retains all prior hyperplanes. The algorithm can be improved by deleting prior hyperplanes [Hogan, 1973].

Checking all possible activation patterns in $\mathcal{M}(\boldsymbol{x}^*)$ guarantees an $\epsilon-$local optimal. However, Set $\mathcal{M}(\boldsymbol{x}^*)$ can have exponentially many possible patterns, which leads to computational challenges. Alternatively, the algorithm can examine the previously visited patterns and add a termination check at Line 40 by solving the outer-approximation (NOA). If the bound from the approximation is close to the best known objective, then the algorithm terminates. Otherwise, the algorithm continues with the solution of approximation (NOA). While this approach would reduce the computational challenges of examining on all possible patterns, the approach does not guarantee a local optimality in some cases. For example, the boundary solution $x = 4$ in Figure 1 is the case if all previous observations are strictly greater than 4.

3.6 Gradient Projection Methods

The gradient projection methods can solve a class of constrained neural network inverse problems. The gradient projection methods are an iterative method that identifies a descent direction with gradients and ensures the feasibility through projection operations [Bertsekas, 1999]. An iteration of gradient projection methods can be described as follows;

$$x^{k+1} = x^k + \alpha^k(\bar{x}^k - x^k), \tag{3.34}$$
$$\bar{x}^k = [x^k - s^k \nabla g(x^k)]^+_{\mathcal{X}}, \tag{3.35}$$

where x^k is the solution at iteration k, $\nabla g(x^k)$ is the gradient of x^k, $[\cdot]^+_{\mathcal{X}}$ denotes projection on the set $\mathcal{X}$, and parameters α^k and s^k are a step size and a positive scalar, respectively. Variable $\bar{x}^k$ is the projected solution of $x^k - s^k\nabla g(x^k)$ onto the set $\mathcal{X}$. For a given solution x^k, Equation (3.35) finds a descent direction $x^k - s^k\nabla g(x^k)$ and if the resulting solution is outside of feasible region $\mathcal{X}$, it projects the solution onto the feasible region $\mathcal{X}$. Similar to gradient based algorithms, Equation (3.34) uses a step size to update the solution.

The projection operation can be computationally expensive depending on the characteristics of feasible region $\mathcal{X}$. If the feasible region is simple bound constraints, the projection operation is same as rounding operation. For other cases, the projection operation can be described as a quadratic programming problem, which often requires computational efforts.

4 Computational experiments

This section discusses the computational results against two sets of neural network inverse problem instances; One set is based on a material design problem. The other instances are randomly generated.

All algorithms are implemented in Python 3.6.5 along with Tensorflow 1.10.0. The linear programming problems are implemented in PuLP 1.6.8[1]. The main linear programming solver of the implementation is CbC 2.9.0[2], which is the default linear solver for PuLP. Computational experiments have been conducted on a Sandy Bridge dual-socket Linux machine with eight 2.6 GHz cores on each socket and 64GB of RAM.

4.1 Material design instances

The material design problem considered in this section is to find the optimal topological polymer structure for given desired rheological properties. The main approach consists of two phases; the first phase is developing a neural network that mimics the behavior of the forward simulation, bob-rheology[3], that predicts the rheological properties with a given topological structure [Read et al., 2011]. The second phase is solving an inverse optimization problem for a given target rheological property description with the trained neural

[1]LP modeler written in Python https://github.com/coin-or/pulp
[2]https://projects.coin-or.org/Cbc
[3]https://sourceforge.net/projects/bob-rheology/

network. The main goal of the second phase is to find many good configurations rather than find a good configuration. Therefore, the multi-start approach is adopted.

The best performing neural network has 5 dense layers with the structure of $7 \times 64 \times 128 \times 256 \times 534$ neurons. The first 7 neurons are inputs for the design parameters and the outputs of the last 534 neurons depicts the rheological properties. All intermediate neurons have a rectified linear activation unit for their output. The inverse optimization problem has two types of constraints. One is upper and lower bounds of input parameters. The other constraint is a ratio constraint of two input parameters such that the sum of the two input elements should be equal to 1.

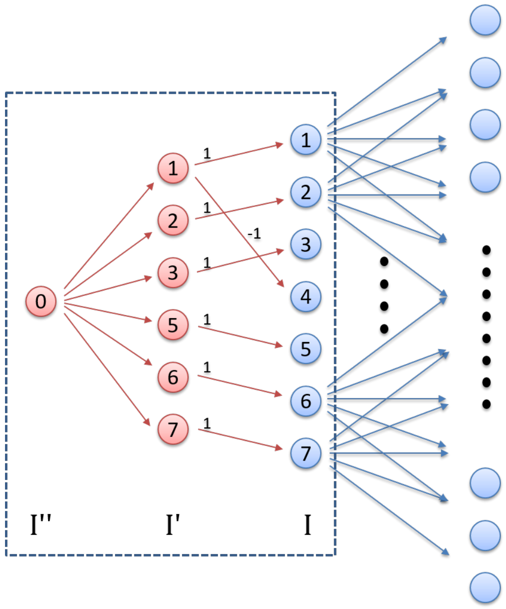


Figure 2: A diagram of the modified neural network: The neurons and arcs in blue are from the trained network and the rest are introduced to solve the inverse problem within the Tensorflow framework. Note that the only trainable parameters are biases in layer I'. The rest of weights and biases are fixed.

The projected gradient method is implemented within Tensorflow with the following modification. The main modification is introducing two additional layers prior to the original input layer in order to describe the ratio constraint and describe the input parameters as variables. Let I be the input layer of the original trained neural network. Let I' be the additional layer connected the input layer I and I'' be another layer adjacent to the layer I'. We create one fewer neurons in layer I' such that we duplicate all neurons in layer I except one neuron in the ratio constraint. In layer I'', we create only one neuron. We set the biases of layer I to 0 except that the bias of the neuron without a replication in layer I' is set to 1. The weights from layer I' to layer I are set to 1 if both neurons have a same neuron number. The weight from the ratio neuron in layer I' to the ratio neuron in layer I is set to -1. All other weights from layer I' to layer I are set to 0. The weights from neurons in layer I'' to neurons in layer I' are set to 0. The neuron in layer I'' is the input layer of the modified neural network. All neurons in layer I'', I' and I are a linear system without any activation function. The rest of neural network remains same as the original trained neural network. At last, we set only the biases of neurons in layer I' trainable and set all other weights and biases not trainable. Figure 2 depicts the modified neural network. Neurons and weight parameters in the dotted box are modified compared to the original trained neural network.

The bound constraints can be incorporated with a simple projection procedure. First, we optimize the biases in layer I' of the modified neural network with a Tensorflow optimizer. With a fixed number of iterations, we examine the solution if it is within the bounds. If it is outside of the bounds, we project it to the feasible bounds. Since it is a simple bound, the projection is equivalent to move the value to the closest

Label	Step size (γ^s)	Neig. size (γ^c)
OGO-0.01-0.1	10^{-2}	10^{-1}
OGO-0.01-0.05	10^{-2}	5×10^{-2}
OGO-0.01-0.01	10^{-2}	10^{-2}
OGO-0.1-0.5	10^{-1}	5×10^{-1}
OGO-0.1-0.05	10^{-1}	5×10^{-2}

(a) Outer approximation guided method

Label	Step size
PGO-0.01	10^{-2}
PGO-0.001	10^{-3}
PGO-0.0001	10^{-4}

(b) Projected gradient method

Table 1: Parameter settings for computational experiments

bound.

The projected gradient algorithm has two termination criteria; one condition is when the next solution is same as the previous solution, which means either the gradient of the current solution is equal to zero or the descent direction is infeasible respect to the bounds. The other is stalling behaviors such that the Tensorflow cannot find a better solution with multiple attempts.

4.1.1 Experimental design

The projected gradient approach and the proposed outer-approximation guided approach have been tested on 100 instances. All instances share the same trained neural network discussed in Section 4.1 with a given target and the mean squared error as the loss function. The only difference between instances is their starting points, which have been collected through examination of the training data set. For each training set, it can be easily to measure the loss between the corresponding output and the given target. The following results are a summary of the 100 optimization runs.

Table 1 summarizes the various optimization hyper-parameter configurations for the experiment. For the projected gradient method, Adam and RMSprop have been tested as the Tensorflow optimizer. For these specific instances, RMSprop outperforms the other optimizer. For the stalling termination criteria, the number of no improvement steps is set to 20, which is tuned to a balanced setting between the solution quality and the computational time. In order to improve the solution time of the projected gradient method, the projection operation is conducted at every 16 (epochs) steps of Tensorflow optimization instead of projection at each step. Table 1 (b) summarizes various step sizes that have been tested for the projected gradient method. Table 1 (a) summarizes various step size parameter γ^s and neighborhood size parameter γ^c of problem (DLOA) for the proposed outer-approximation guided method. All other hyper-parameters for Algorithm 3.1 are summarized in Table 2.

ϵ	$\gamma^s_{\min}$	$\gamma^s_{\max}$	ρ^c	ρ^e	N
$1E-5$	γ^c	$\epsilon \times \sqrt{n}$	0.9	1.5	$1E3$

Table 2: Optimization hyper-parameters for Algorithm 3.1. Parameter n denotes the number of input variables.

The percent-gap-closed to the best known solution are employed as the solution quality metrics. The percent-gap-closed metrics ρ_j^k for instance j and approach k is defined as follows;

$$\rho_j^k = \frac{v_j^0 - v_j^k}{v_j^0 - v_j^*}, \tag{4.1}$$

where v_j^0 and v_j^* denote the initial objective value and the best objective value among all approaches for instance j. v_j^k is the objective value of approach k for instance j. The denominator of Equation (4.1) computes the best known improvement for instance j and the numerator computes the improvement for the corresponding approach. The metrics measures how close to the best known solution the solution from each method is.

4.1.2 Overall solution time and quality

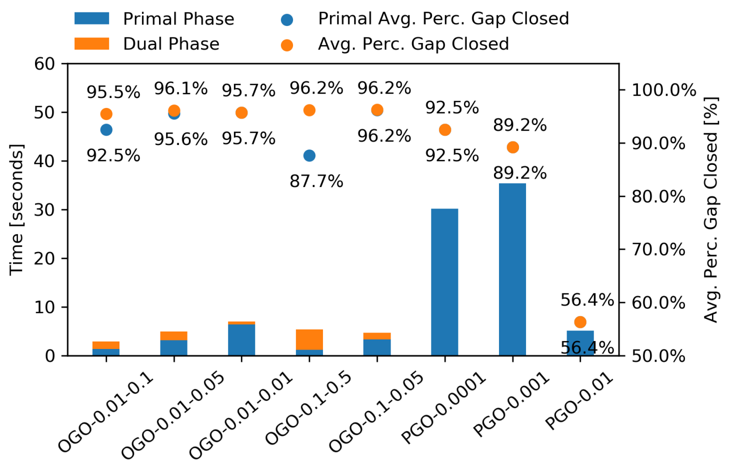


Figure 3: The average solution time and average percent-gap-closed metrics. The x-axis represents optimization approaches with various hyper-parameter settings. The bar charts are the average solution time of 100 instances. The dot charts show the average percent-gap-closed across 100 optimization runs at termination. The blue bars and dots are the results for the primal phase and the orange bars and dots are corresponding to the dual phase.

Figure 3 summarizes the average solution time and the average percent-gap-closed metrics across 100 instances in the primal and dual phases. The proposed outer-approximation algorithm can alter the status between primal and dual phases. However, in the computational summary, once the algorithm enters the dual phase, the subsequent procedure is treated as the dual phase. The projected gradient method has only the primal phase.

For the projected gradient method, as the step size decreases, the solution time increases and the solution quality is improved. There is no such clear trend for the proposed algorithm in terms of the solution quality and time. The computational result implies that the time spent in the primal phase is dependent on the ratio of the step size γ^s and the neighborhood size γ^c. If they are similar, the algorithm tends to spend more time in the primal phase.

The proposed outer-approximation guided method outperforms the projected gradient method in terms of the solution time and quality. The projected gradient method with a small step size (PGO-0.01) can solve the problems within comparable solution times as the proposed algorithm but it cannot achieve similar solution qualities. The solution quality from the proposed approach is superior to one from the projected gradient method.

The average improvement on the computational time ranges between 30 and 35 seconds, which can be considered as insignificant. However, in the material design context, it is required to solve many cases up to hundreds or thousands in order to produce many leading candidates and overcome the local optimality issues. In that regard, the percentage improvement, 1 - the average time for the proposed algorithm / the average time for the projected gradient method, is a more meaningful measure, which ranges between 77% and 92%. It implies that the proposed algorithm can solve the same number of problems with less than 23% of the computational efforts.

4.1.3 Solution progress profile

Figure 4 shows the progress profile of the solution for four approaches (OGO-0.01-0.1, OGO-0.1-0.05, PGO-0.001, PGO-0.0001). The graphs in Figure 4 show the average solution quality over time. It clearly shows

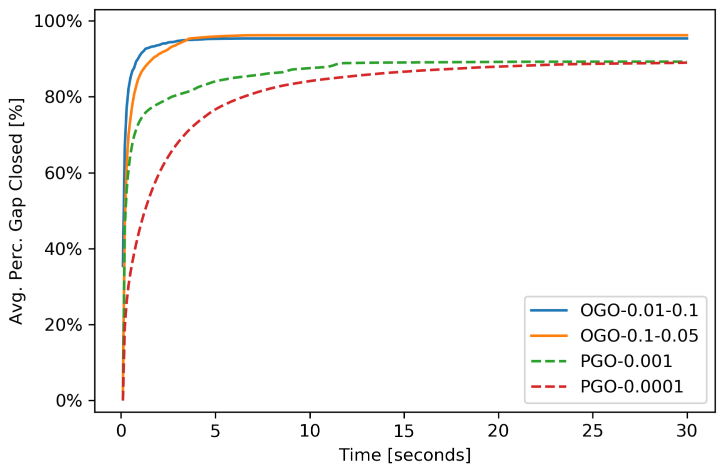


Figure 4: The progress profile of the average percentage gap closed metrics. The x-axis represents time spent in seconds. The y-axis is the average percentage gap closed.

that the proposed algorithm finds a better solution quickly. Both outer-approximation approaches achieve over 90% solution quality within 5 seconds whereas the projected gradient methods around 80% of the best known solution.

4.2 Randomly generated instances

The random instances are generated as follows; The process starts with a given neural network structure. The first step is randomly generating weights and biases for the given neural network structure. The initial randomization uses a normal distribution. Randomly generated neural networks tend to have a huge spread at outputs even with controlled inputs ranging between 0 and 1. The second step is to scale the output of the neural network. In the second step, the neural network is re-trained with many scaled outputs, which can be generated from the neural network from the first step. Many pairs of input and output data can be generated along with the neural network from the first step. After the output data is processed to range between 0 and 1, the processed data is used to retrain the neural network.

The two network architectures in Table 3 are considered as the base structures. The targets are randomly generated and are not part of the scaled training set. The starting points are randomly chosen with a standard uniform distribution. Only bound constraints for inputs are considered.

S1	100,256,128,64,128,64,32,8
S2	256,128,64,128,64

Table 3: Base neural network structures for random instances

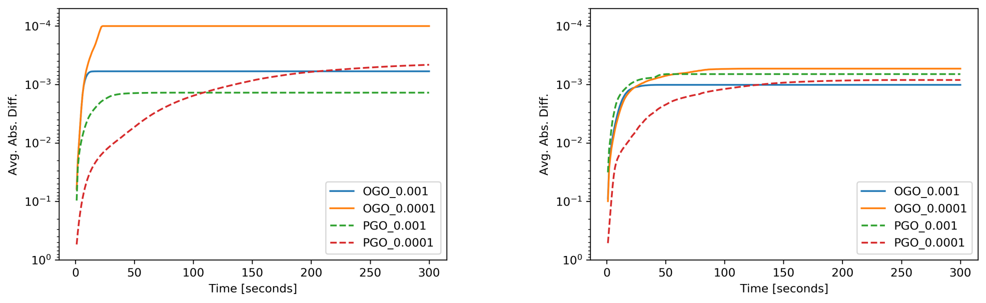
For the outer approximation guided approach, the hyper-parameter settings in Table 4 are tested. In this experiment, the various optimality gap criteria are considered. For the projected gradient approach, two step sizes are considered. PGO_0.001 and PGO_0.0001 refer to the projected gradient methods with step size 0.001 and 0.0001, respectively.

Because of random starting points, the percentage-gap-closed metrics does not distinguish the performance difference of approaches. The most of final solutions are within a range of 10^{-2} and initial solutions are typically far from the local optimal solution because of random starting points. These two factors make the denominator of the metrics large and the differences in the numerator small. In this experiment, the absolute

Label	ϵ	γ^s	γ^c	γ^s_{min}	γ^s_{max}	ρ^c	ρ^e	N
OGO_0.001	$1E-3$	0.01	0.5	γ^c	$\epsilon \times \sqrt{n}$	0.9	1.5	$1E3$
OGO_0.0001	$1E-4$	0.01	0.5	γ^c	$\epsilon \times \sqrt{n}$	0.9	1.5	$1E3$

Table 4: Optimization hyper-parameters of Algorithm 3.1 for random instances. Parameter n denotes the number of input variables.

difference to the best known solution is used instead of the percentage-gap-closed. The metrics measures the absolute difference between the best known solution and the solution of each instance and approach.



(a) S1 (b) S2

Figure 5: The progress profile of the average absolute difference. The x-axis represents time spent in seconds. The y-axis in a logarithmic scale is the average absolute difference to the best known solution.

Figure 5 shows the representative progress profiles of experiments. The y-axis in a logarithmic scale is the average absolute difference between the solution over time and the best known solution. Each line denotes the different approach. Note that the orange line is truncated at 10^{-4} for display purpose.

The outer approximation guided approach finds a good quality solution quickly for neural network structure S1 instances compared to the projected gradient method. For neural network structure S2 instances, the projected gradient method with step size of 0.001 shows better performance than two outer-approximation guided approaches at the earlier stage of the progress. After around 50 seconds, the outer-approximation guided approach with optimality tolerance of 10^{-4} achieves a better average solution quality. Note that the optimality gap ϵ is respect to a local optimal solution and not for the global optimality. Therefore, OGO_0.0001 can have solutions with more than 0.0001 absolute difference. It is hard to say that the proposed algorithm always outperforms the projected gradient methods. The numerical results show that the proposed algorithm is generally superior to the projected gradient method discussed.

5 Conclusion

This paper discusses constrained neural network inverse optimization problems that find the optimal set of input parameters for a desired output and proposes an outer-approximation guided method, especially when the inverse problem has additional constraints on input parameters. The proposed method is devised by exploiting the characteristics of the local optimal solution for neural network inverse optimization problems with rectified activation units. The proposed algorithm consists of primal and dual phases. In the primal phase, the algorithm incorporates neighbor gradients through outer approximations of neighbors to expedite the convergence to a good quality solution. In the dual phase, the algorithm exploits the convex structure of the local optimal solution to improve the speed of convergence. In addition, the paper proposes a method to generate feasibility cuts without solving an explicit dual problem. The superiority of the proposed algorithm is demonstrated with computational results compared to a projected gradient method for a material design problem and randomly generated instances.

References

[Anderson et al., 2019] Anderson, R., Huchette, J., Tjandraatmadja, C., and Vielma, J. P. (2019). Strong mixed-integer programming formulations for trained neural networks. In Lodi, A. and Nagarajan, V., editors, *Integer Programming and Combinatorial Optimization*, pages 27–42, Cham. Springer International Publishing.

[Ben-Tal and Nemirovski, 2005] Ben-Tal, A. and Nemirovski, A. (2005). Non-euclidean restricted memory level method for large-scale convex optimization. *Mathematical Programming*, 102(3):407–456.

[Benders, 1962] Benders, J. F. (1962). Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4(1):238–252.

[Bertsekas, 1999] Bertsekas, D. (1999). *Nonlinear Programming.* Athena Scientific.

[Chen and Gu, 2020] Chen, C. T. and Gu, G. X. (2020). Generative deep neural networks for inverse materials design using backpropagation and active learning. *Advanced Science*, page 10.

[Corts et al., 2009] Corts, O., Urquiza, G., and Hernndez, J. (2009). Optimization of operating conditions for compressor performance by means of neural network inverse. *Applied Energy*, 86(11):2487 – 2493.

[Duran and Grossmann, 1986] Duran, M. A. and Grossmann, I. E. (1986). An outer-approximation algorithm for a class of mixed-integer nonlinear programs. *Mathematical Programming*, 36(3):307–339.

[Eaves and Zangwill, 1971] Eaves, B. and Zangwill, W. (1971). Generalized cutting plane algorithms. *SIAM Journal on Control*, 9(4):529–542.

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

[Geoffrion, 1970] Geoffrion, A. M. (1970). Elements of large-scale mathematical programming part i: Concepts. *Management Science*, 16(11):652–675.

[Geoffrion, 1972] Geoffrion, A. M. (1972). Generalized benders decomposition. *Journal of Optimization Theory and Applications*, 10(4):237–260.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

[Hogan, 1973] Hogan, W. W. (1973). Applications of a general convergence theory for outer approximation algorithms. *Mathematical Programming*, 5(1):151–168.

[Hornik et al., 1990] Hornik, K., Stinchcombe, M., and White, H. (1990). Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3(5):551 – 560.

[Kelley, 1960] Kelley, Jr., J. (1960). The cutting-plane method for solving convex programs. *Journal of the Society for Industrial and Applied Mathematics*, 8(4):703–712.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[Linden and Kindermann, 1989] Linden, A. T. and Kindermann, J. (1989). Inversion of multilayer nets. *International 1989 Joint Conference on Neural Networks*, pages 425–430 vol.2.

[O'Malley et al., 2019] O'Malley, D., Golden, J. K., and Vesselinov, V. V. (2019). Learning to regularize with a variational autoencoder for hydrologic inverse analysis.

[Peters and Schaal, 2008] Peters, J. and Schaal, S. (2008). Natural Actor-Critic. *Neurocomputing.*

[Read et al., 2011] Read, D. J., Auhl, D., Das, C., den Doelder, J., Kapnistos, M., Vittorias, I., and McLeish, T. C. B. (2011). Linking models of polymerization and dynamics to predict branched polymer structure and flow. *Science*, 333(6051):1871–1874.

[Rezaee and Dadkhah, 2019] Rezaee, M. J. and Dadkhah, M. (2019). A hybrid approach based on inverse neural network to determine optimal level of energy consumption in electrical power generation. *Computers & Industrial Engineering*, 134:52–63.

[Schmidhuber, 2015] Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117.

[Silver et al., 2017] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Van Den Driessche, G., Graepel, T., and Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature.*

[Szegedy et al., 2014] Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. (2014). Intriguing properties of neural networks. In *International Conference on Learning Representations.*