IMPERIAL

Improving backpropagation in the PiShield package

Author
Arnav Kohli
CID: 02018546

Supervised by
Dr Eleonora Giunchiglia

Prof. Jeremy Pitt

A Thesis submitted in fulfillment of requirements for the degree of
Master of Engineering in Electronic and Information Engineering

Department of Electrical and Electronic Engineering
Imperial College London
2025

Abstract

Effective backpropagation is fundamental to neural network training, but its efficacy diminishes when outputs are required to satisfy complex constraints. PiShield is a PyTorch-based framework that enforces such constraints through shield layers, ensuring feasibility of network predictions. While PiShield guarantees constraint satisfaction via clamping-based operations, this approach introduces challenges during backpropagation, particularly in multivariate-regression tasks. The use of non-differentiable functions such as max() and min() can block gradient flow when constraints are active. Moreover, when clamped outputs are evaluated under a Mean Squared Error loss, gradient components may conflict: one term pulls predictions toward the ground truth, while another pushes them toward constraint boundaries. These opposing forces degrade convergence, destabilize optimization, and limit learning efficiency.

This thesis proposes two methods to address these limitations. The first introduces a masked, sign-aware loss function that selectively directs gradient flow through a constrained subset of outputs, determined by one-hot encoded masks. The second applies projection-based gradient correction that ensures updates remain within the tangent space defined by active constraints. This is done on the backward pass, while using the conventional shield layer on the forward pass. Both approaches attempt to preserve feasibility during the forward pass, which is a characteristic feature of the shield layer.

The proposed methods were evaluated on multiple regression datasets using various performance metrics and constraint structures. Results demonstrate that the projection-based gradient correction produces better test RMSE, and gradient direction alignment, particularly in deeper and wider networks, due to its reliance on dynamical isometry. Conversely, the masked method exhibits some limitations that require further improvement in future work.

Declaration of Originality

I hereby declare that the work presented in this thesis is my own unless otherwise stated. To the best of my knowledge the work is original and ideas developed in collaboration with others have been appropriately referenced. Selective use of the large language models `ChatGPT-4o` and `ChatGPT-4o-mini` by OpenAI has been employed to improve grammar and structure, in accordance with the guidelines established by the Department of Electrical and Electronic Engineering. Although these tools provided assistance in refining the language, I remain fully responsible for designing and implementing all the derivations, analysis, and conclusions presented in this work.

Copyright Declaration

Contents

List of Acronyms

DGM *Deep Generative Models*

CL *Constraint Layer*

GAN *Generative-Adversarial Network*

QFLRA *Quantifier-Free Linear Real Arithmetic*

DRL *Disjunctive Refinement Layer*

CP *Cutting Planes*

DC3 Deep Constraint Completion and Correction

RMSE Root Mean-Square Error

List of Figures

List of Tables

1 Introduction

PiShield [1] is a PyTorch-based package designed to integrate domain-specific requirements into neural networks. By introducing a novel neural network layer known as a Shield Layer, PiShield guarantees that a neural network's outputs adhere to specified constraints (continuous or propositional), irrespective of the input. This ensures safety and reliability in applications where adherence to predefined rules is critical. It is a package developed for the broader sphere of neuro-symbolic AI.

The foundations of PiShield were established in [1], [2], which introduced novel frameworks allowing the incorporation of either propositional or linear constraints into neural network topologies for both classification tasks and tabular data generation tasks. Their approach was extended to embed constraints directly into the network architecture, either during training or at inference time, using specialized PyTorch layers. These "Shield Layers," as illustrated in 1.1, enforce compliance with the requirements specified, ensuring that neural network outputs lie within a safe and permissible range based on them. For example, PiShield ensures "hierarchical coherence in gene function predictions within functional genomics" [1] and enforces logical consistency in "road event detection for autonomous driving" [1].

Despite its success in enforcing constraints, PiShield's current implementation has not been tested in the realm of regression. Due to multiple target variables being possible within the constraint space, this thesis' main focus is on *multivariate regression*.

Given this context, the package faces challenges in optimizing backpropagation under linear constraints. Consider a prediction model that enforces the constraint $y_1 \geq y_2$ to ensure that the

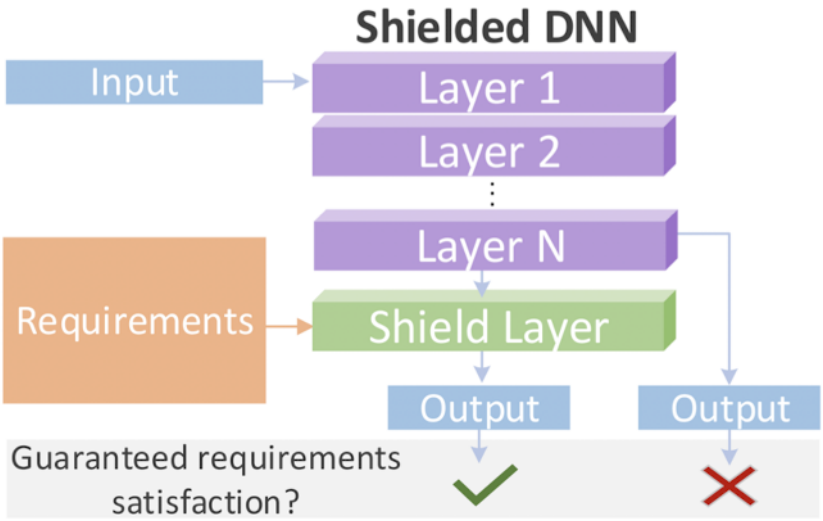


Figure 1.1: An illustration of the shield layers involved in ensuring constraints are respected in PiShield ([1]).

corrected predictions $\hat{y}'_1$ and $\hat{y}'_2$ satisfy this inequality. The shield layer enforces clamping operations through max() and min() (more details in Chapters 2 and 3) and produces corrected predictions from the unconstrained predictions $\hat{y}_1$ and $\hat{y}_2$ as follows:

$$\hat{y}'_1 = \max(\hat{y}_1, \hat{y}_2), \quad \hat{y}'_2 = \hat{y}_2 \tag{1.1}$$

This guarantees that $\hat{y}'_1 \geq \hat{y}'_2$, thereby satisfying the constraint. However, this approach complicates gradient computation during backpropagation, affecting the optimization process. The Mean Squared Error (MSE) is used as the loss function, defined as

$$\mathcal{L} = (\hat{y}'_1 - y_1)^2 + (\hat{y}'_2 - y_2)^2 \tag{1.2}$$

where y_1 and y_2 are the ground-truth values. This is a common loss function for regression tasks. The objective is to minimize this loss by adjusting $\hat{y}_1$ and $\hat{y}_2$ through gradient-based optimization.

When the unconstrained predictions violate the constraint $\hat{y}_1 < \hat{y}_2$, the corrected value $\hat{y}'_1$ depends entirely on $\hat{y}_2$, effectively decoupling $\hat{y}_1$ from the loss. This creates an asymmetry in how the model adjusts $\hat{y}_1$ and $\hat{y}_2$, leading to unstable optimization behavior.

To understand this issue, consider the gradient of the MSE loss with respect to $\hat{y}_2$ when the

model outputs satisfy:

$$y_1 \geq \hat{y}_2 \geq \hat{y}_1 \geq y_2 \tag{1.3}$$

In this scenario, the gradient of the loss with respect to $\hat{y}_2$ is derived as follows:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_2} = 2(\hat{y}_2 - y_2) + 2(\hat{y}_2 - y_1) \tag{1.4}$$

This expression reflects two competing forces acting on $\hat{y}_2$:

- The first term, $2(\hat{y}_2 - y_2)$, adjusts $\hat{y}_2$ to minimize the difference between the prediction and the ground truth y_2.
- The second term, $2(\hat{y}_2 - y_1)$, penalizes deviations of $\hat{y}_2$ from y_1, ensuring that $\hat{y}_1' = \hat{y}_2$ remains consistent with the constraint $y_1 \geq \hat{y}_2$.

These gradient components indicate that $\hat{y}_2$ must simultaneously decrease to align with y_2 and increase to satisfy the constraint. The two components thus compete with each other to satisfy different goals, whereas in an ideal scenario, the gradient directions would align or at least find a decent compromise. Another problem arises because the correction $\hat{y}_1' = \max(\hat{y}_1, \hat{y}_2)$ introduces a non-smooth (and thus non-differentiable) operation. This causes a sudden change in the gradient when $\hat{y}_1$ crosses $\hat{y}_2$.

The main contribution of this work is the design, analysis, and evaluation of two novel methods to resolve this gradient conflict while preserving PiShield's guarantee of constraint satisfaction. The two proposed methods are as follows:

1. **Mask and Modified MSE:** A method that isolates gradient flow to a single variable per constraint and uses a signed loss function to align the update direction with the supervised target.
2. **Projected Gradients:** An adaptation of classical gradient projection techniques integrated into the training process via a straight-through estimator and shield layer, enabling constraint-preserving gradient updates with differentiable gradients.

This thesis begins by providing background on the relevant optimization techniques and the foundations of PiShield. This is followed by a detailed mathematical and analytical discussion of

the two proposed methods along with their implementation. Finally, a comprehensive experimental analysis is presented to evaluate their effectiveness against several key performance metrics, including RMSE and gradient alignment.

2

Background

Contents

This chapter introduces some of the background knowledge already incorporated within Pishield, as well as some essential mathematical groundwork needed to improve back-propagation as described in 2.2.

2.1 Coherent Hierarchical Multi-Label Classification

A classification problem is a supervised learning task where a model learns to predict a categorical label or set of labels for a given input. Formally, given an input space $\mathcal{X}$ and an output space $\mathcal{Y}$, the goal is to learn a function $f : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{Y}$ consists of discrete categories. This includes:

- **Binary classification**, where $\mathcal{Y} = \{0, 1\}$, predicting whether an input belongs to a single class or not.
- **Multi-class classification**, where $\mathcal{Y} = \{y_1, y_2, \ldots, y_k\}$, assigning each input to one of k possible categories.

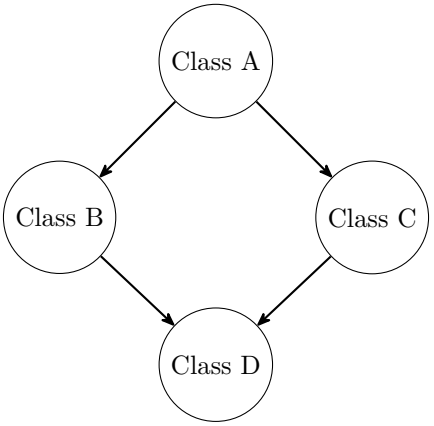


Figure 2.1: A Directed Acyclic Graph (DAG) representing hierarchical class relationships.

- **Multi-label classification**, where $\mathcal{Y} \subseteq \{y_1, y_2, \ldots, y_k\}$, allowing an input to belong to multiple classes simultaneously.

Many real-world classification tasks introduce additional constraints, such as class hierarchies or dependencies, which models must respect for logical consistency. These relationships are often represented as a **Directed Acyclic Graph (DAG)**, where each node is a class and directed edges indicate parent-child relationships. For example:

- A *car* is a type of *vehicle*, establishing a **parent-child relationship** between *vehicle* and *car*. If an object is predicted as a *car*, it must also be a *vehicle*.
- A *bike* is also a *vehicle* but is disjoint from *car*. Thus, *bike* and *car* are **siblings** under *vehicle*, and their predictions should not overlap.

Following the methodology in [2], a class A is predicted when its score $m_A(x)$, representing the model's confidence for input x, exceeds a user-defined threshold τ_A.

$$m_A(x) \geq \tau_A \implies \text{Class } A \text{ is predicted.} \tag{2.1}$$

Crucially, in a hierarchical setup, predictions must be *hierarchically consistent*. This means that for any descendant class B of A, its confidence score cannot exceed that of its parent:

$$m_B(x) \leq m_A(x) \quad \forall B \in \text{Descendants of } A. \tag{2.2}$$

This ensures that if a child class B is predicted, its parent A will also be predicted.

To enforce these hierarchical relationships, specialized post-processing and loss functions are

employed. While various methods exist for adjusting confidence scores to meet these constraints (for a detailed comparison of methods like f^+ and g^+, see Appendix B.1), a robust solution often involves a two-layer approach combined with a custom loss function.

This approach utilizes:

- An initial **output layer** h that provides scores (e.g., in $\{0, 1\}$) for each class.
- A **Max-Confidence Module (MCM)**. The MCM propagates confidence scores upwards through the hierarchy defined by the DAG (e.g., Figure 2.1). It ensures that a parent node's confidence is never lower than its most confident child node. This is achieved by taking the maximum confidence score from a class and its most confident descendant. The general definition of the output of this layer for a class A is:

$$MCM_A = \max_{B \in \mathcal{D}_A} (h_B) \tag{2.3}$$

  where $\mathcal{D}_A$ is the set of descendants of class A. This concept is often referred to as the *delegation mechanism*, as it ensures that the parent's confidence is effectively 'delegated' from its most confident child.

- A custom loss function, **MCLoss**. For a class $C \in \mathcal{S}$ (a set of hierarchically structured classes), MCLoss inherently accounts for hierarchical dependencies. Unlike standard categorical cross-entropy (CE), which treats each prediction independently, MCLoss directly incorporates consistency constraints. The aggregated MCLoss is given by:

$$MCLoss = \sum_{A \in S} \left(-y_A \ln(\max_{B \in \mathcal{D}_A} (y_B h_B)) - (1 - y_A) \ln(1 - MCM_A) \right) \tag{2.4}$$

  The formulation of MCLoss yields a more consistent gradient for adjusting class predictions, especially for parent classes with numerous children. This eliminates the problematic dependency on the number of parent classes found in CE, allowing the model to effectively learn and reduce hierarchical violations. For a detailed mathematical analysis of the MCLoss gradient compared to categorical cross-entropy, refer to Appendix B.2.

This robust methodology forms the basis for classification-based constraint enforcement within the PiShield package [3].

2.2 Enforcing Constraints for Continuous Values

The generation of realistic synthetic data to augment existing data is vital in the current machine learning landscape. *Deep Generative Models* (DGM)s have demonstrated significant capability in learning complex data distributions for this purpose. These models provide a useful unsupervised analogue to the supervised back-propagation layer that is the focus of this project. In real-world datasets, there are implicit restrictions and relationships which are unspoken and encoded within the data due to various factors including business requirements, laws of nature, etc. The following sections discuss papers which focus on tackling these problems to varying degrees. The implementation of these methods serve as the foundation for PiShield's [1] own linear and convex constraint enforcement method.

2.2.1 Constraint Layer for Linear Inequalities

Linear inequalities represent a fundamental form of constraint commonly imposed in various domains. These are mathematically expressed as:

$$\sum_k w_k x_k + b \trianglerighteq 0, \tag{2.5}$$

where $k \in \mathbb{N}$ and $1 \leq k \leq \mathcal{D}$. Here, $\mathcal{D}$ denotes the total number of constraints, w_k represents the weight or coefficient associated with the variable x_k, b is a bias term, and the operator $\trianglerighteq \in \{\geq, >\}$. A strict inequality ($>$) can be modeled as a non-strict inequality in the limiting case by using a small positive float, ϵ. The complete set of these constraints, as described in [3], is denoted as Π, defining a convex feasible region within the high-dimensional feature space.

The paper introduces a differentiable **Constraint Layer (CL)**, which minimally modifies the output of a Deep Generative Model (DGM) while ensuring compliance with the constraints laid out in Π. The CL achieves this by systematically adjusting each feature of the generated output, preserving its overall structure and realism. An example of the functioning of this layer can be seen in Figure 2.2, where it ensures that outputs from the Generator of a GAN respect the inequalities laid out in the requirements.

To simplify the enforcement of linear inequality constraints, the paper presents an iterative inductive process. This methodology systematically adjusts variables, propagating feasible adjustments across constraints in a structured manner. The core idea is to process the constraints

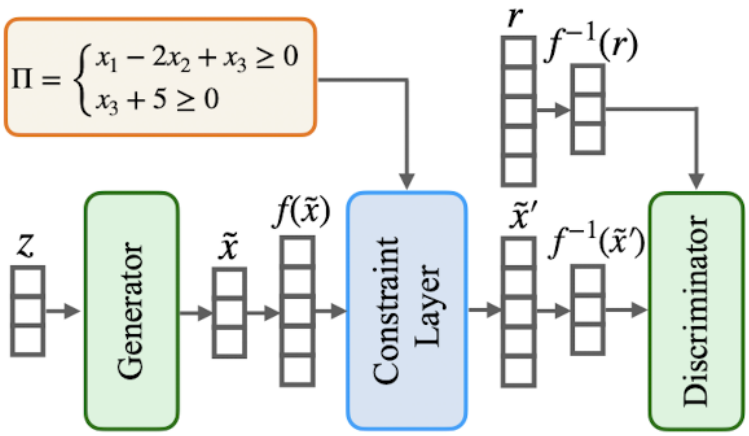


Figure 2.2: CL incorporated in a GAN, taken from [3]

one variable at a time, ensuring each adjustment maintains the overall validity of the feature. This iterative process addresses each variable sequentially, refining the constraints to eliminate already-processed variables while preserving the feasible region.

The process is defined as follows:

- A **variable ordering** is introduced for the set of constraints Π, using a mapping λ such that $\lambda(x_i) = i$. This ordering determines the sequence in which constraints are processed, and it can be adjusted for different computational priorities.
- Each variable x_i is associated with a set of constraints Π_i, defined inductively:
  - If $i = D$, the final variable in the order, then $\Pi_i = \Pi$, i.e., the full set of constraints.
  - If $i < D$, the constraints Π_i are derived from the constraints of the subsequent variable Π_j where $j = i + 1$, using the following rule:

$$\Pi_i = \Pi_j \setminus (\Pi_j^- \cup \Pi_j^+) \cup \{\text{red}_j(\phi^1, \phi^2) \mid \phi^1 \in \Pi_j^-, \phi^2 \in \Pi_j^+\}. \tag{2.6}$$

Here, Π_j^- and Π_j^+ represent subsets of Π_j where the variable x_j appears with a negative and positive coefficient, respectively. The function $\text{red}_j(\phi^1, \phi^2)$ is the **reduction operator** which eliminates variable x_j from two constraints while maintaining their logical consistency. For two constraints $\phi^1 = \sum_k w_k^1 x_k + b^1 \trianglerighteq^1 0$ and $\phi^2 = \sum_k w_k^2 x_k + b^2 \trianglerighteq^2 0$, the reduced constraint is:

$$\sum_{k \neq j} (w_k^1 |w_j^2| + w_k^2 |w_j^1|) x_k + b^1 |w_j^2| + b^2 |w_j^1| \trianglerighteq 0. \tag{2.7}$$

For a detailed numerical example of this reduction process, refer to Appendix B.3.

For a given variable x_i, the CL computes feasible upper and lower bounds based on its associated constraints in Π_i. These bounds are derived as:

$$ub_i = \min_{\phi \in \Pi_i^-} \left(\frac{-b_\phi - \sum_{k \neq i} w_k x_k}{w_i} \right), \quad lb_i = \max_{\phi \in \Pi_i^+} \left(\frac{-b_\phi - \sum_{k \neq i} w_k x_k}{w_i} \right). \tag{2.8}$$

For strict inequalities, x_i cannot equal lb_i (or ub_i), and it is adjusted by adding a small positive float, ϵ. The final value of x_i is then computed by clipping the input value $\tilde{x}_i$ within these feasible bounds:

$$\text{CL}(\tilde{x}_i) = \min\left(\max(\tilde{x}_i, lb_i), ub_i\right). \tag{2.9}$$

This ensures that x_i consistently lies within the valid range $[lb_i, ub_i]$. According to *Theorem 3.6* in [3], this process yields a solution that is as close to optimal as possible for $\text{CL}(\tilde{x}_i)$ given the constraints. A numerical example demonstrating the iterative adjustment of variables based on these bounds can be found in Appendix B.4.

This layer serves as the foundation for the shield layer, which is the main focus of a majority of this thesis. The shield layer implements the reduction operation and clamping to guarantee constraint satisfaction.

2.2.2 Non-Convex and/or Disconnected Disjunctive Constraints

Extending the ideas defined above to non-convex and disconnected spaces allows higher expressivity and the ability to capture higher dimensional relationships and differing modality of data generated. Giunchiglia et al. [4] introduce *Quantifier-Free Linear Real Arithmetic* (QFLRA) formulae, which enable features that are a combination of conjunctions, disjunctions or negations of linear inequalities. QFLRAs allow the aforementioned non-convex and disconnected spaces to be captured.

Linear inequalities define upper and lower bounds for each feature which must be respected, QFLRAs define multiple intervals where the background knowledge holds, each with their own upper and lower bounds (since they are linear inequalities). The paper exploits *De Morgan*'s laws to transform disjunctions, conjunctions and negations to/from one another, and generalizes to use disjunctions. QFLRAs have three components:

- Linear inequalities Φ_i over a set of features $\{x_k : 1 \leq k \leq \mathcal{D}\}$ with dimension $\mathcal{D} \in \mathbb{N}$ (similar to the definition/terminology used above).

$$\Phi_i = \sum_k w_k x_k + b \geq 0 \tag{2.10}$$

  [4]

- Constraints Ψ defined as a set of $n_\Psi \in \mathbb{N}$ disjunctions over the linear inequalities defined above.

$$\Psi = \Phi_1 \vee \Phi_2 \vee \cdots \vee \Phi_{n_\Psi} \tag{2.11}$$

  [4]

- Constraints Ψ defined as a set of $n_\Psi \in \mathbb{N}$ disjunctions over the linear inequalities defined above.

$$\Psi = \Phi_1 \vee \Phi_2 \vee \cdots \vee \Phi_{n_\Psi} \tag{2.12}$$

  [4] Due to this being a disjunction, a feature $\tilde{x}$ satisfies the constraints if it satisfies the conditions of any Φ_i.

- Set of constraints Π represented as $\{\Psi_j : 1 \leq j \leq Z\}$. Where $Z \in \mathbb{N}$ is the number of constraints in the set.

The set σ represents the domain of feasible solutions for a variable or a set of variables, as determined by the given constraints. Each constraint, whether a linear inequality or a disjunction of inequalities, defines a subset of the space where the constraints hold. σ is the union of all these subsets.

For a single constraint Ψ, which is a disjunction of linear inequalities:

$$\Psi = \Phi_1 \vee \Phi_2 \vee \cdots \vee \Phi_{n_\Psi}, \tag{2.13}$$

a feature $\tilde{x}$ satisfies Ψ if it satisfies at least one of the linear inequalities Φ_i. The corresponding feasible region for Ψ is defined as:

$$\Omega(\Psi) = \bigcup_{i=1}^{n_\Psi} \Omega(\Phi_i), \tag{2.14}$$

where $\Omega(\Phi_i)$ represents the feasible region defined by the linear inequality Φ_i.

For a set of constraints $\Pi = \{\Psi_j : 1 \leq j \leq Z\}$, the overall feasible region is the intersection of

the feasible regions of all constraints in Π:

$$\Omega(\Pi) = \bigcap_{j=1}^{Z} \Omega(\Psi_j). \tag{2.15}$$

The *Disjunctive Refinement Layer* (DRL) adjusts predictions to satisfy these complex disjunctive constraints, enabling the capture of non-convex and disconnected feasible regions. It refines values by identifying boundaries derived from constraints and ensuring the adjusted prediction minimizes deviation from the original value while satisfying all conditions, working as an analogue to the linear constraints method defined by Stoian et al. [3]. The process sequentially adjusts variables until all are assigned values within their feasible ranges. A detailed explanation of the DRL's operations, including its application in single and multi-variable scenarios and the underlying Cutting Planes (CP) rule, is provided in sections B.5 and B.6 of the appendix. This includes the specific formulae for both methods.

The analysis of \ac{QFLRA} formulae and the Disjunctive Refinement Layer (DRL) for non-convex and disconnected constraints provides a crucial theoretical backdrop, demonstrating the potential for sophisticated constraint satisfaction beyond the linear case. While the practical applications and detailed analyses in the remainder of this thesis concentrate exclusively on linear constraint scenarios, this high-level treatment is included as a stepping stone, because it is the logical next step towards extending solutions for linear constraints, and provides a clear direction for future work.

2.3 Related Work

The current implementation of the shield layer places limited emphasis on the geometrical aspects of constraint satisfaction. Its procedural approach relies on variable ordering to fix certain variables and incrementally adjust the remaining ones to maintain feasibility. This section aims to highlight alternative approaches that frame constraint satisfaction from a more geometrical perspective. In particular, this is directly relevant to the **projected gradient correction** method discussed in section 3.2.3.

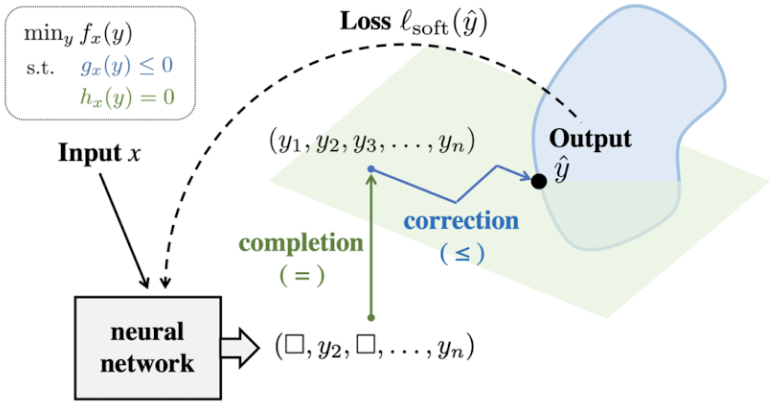


Figure 2.3: A schematic of the DC3 framework.

2.3.1 DC3

The foundation of Deep Constraint Completion and Correction (DC3) ([5]) lies in addressing the limitations of traditional neural networks, which, while effective function approximators, are inherently unconstrained and unable to ensure that their outputs comply with domain-specific rules, physical principles, or logical requirements. This issue becomes significant in scenarios where violating these constraints results in predictions that are invalid or inapplicable, such as in physics-informed systems, resource management, or structured decision-making processes.

DC3 focuses on implementing strict constraints, both linear and nonlinear, in a way that is fully differentiable and scalable. These constraints are directly embedded into the optimization process rather than being weakly enforced through penalties in the loss function or by introducing post-processing adjustments. By incorporating mechanisms to handle constraints within the neural network's training process, DC3 ensures that these requirements are met without compromising the efficiency or adaptability of the model.

To facilitate training while ensuring constraint satisfaction, DC3 employs a composite "soft" loss function. This loss combines the primary task objective $f_x(y)$ with penalties for violations of equality and inequality constraints:

$$\ell_{\text{soft}}(y) = f_x(y) + \lambda_g \|\text{ReLU}(g_x(y))\|^2 + \lambda_h \|h_x(y)\|^2, \tag{2.16}$$

where λ_g and λ_h determine the weight of the penalties for inequality and equality violations, respectively.

Equality Completion

DC3 enforces equality constraints $h_x(y) = 0$ during training and testing through a process called equality completion. These constraints arise in scenarios where specific relationships among variables must hold, such as physical laws or logical dependencies. Strict enforcement of these constraints ensures that the outputs are always feasible, avoiding the limitations of penalty-based or post-hoc correction methods.

Instead of directly outputting the full-dimensional solution $y \in \mathbb{R}^n$, the neural network $N_\theta(x)$ produces $z \in \mathbb{R}^m$, a subset of the variables known as the independent variables. The remaining $n-m$ dependent variables are computed using a function $\varphi_x(z)$, ensuring $y = [z^T, \varphi_x(z)^T]^T$ satisfies $h_x(y) = 0$. This reduces the computational complexity by limiting the network's task to predicting only the degrees of freedom permitted by the constraints.

An important assumption in DC3 is that the problem is not overdetermined, i.e., the number of equality constraints $n - m$ must not exceed the dimension of the decision variable $y \in \mathbb{R}^n$. Overdetermined systems typically do not have feasible solutions because the constraints conflict or overspecify the problem.

Equality constraints reduce the degrees of freedom in a problem. With $n - m$ constraints, only m variables can vary freely, while the rest must conform to the specified relationships. By focusing on the independent variables, DC3 simplifies learning and ensures feasibility when the dependent variables are computed using $\varphi_x(z)$. This function can be determined explicitly for simple constraints, such as linear systems, or implicitly using numerical methods like Newton's method.

When $\varphi_x(z)$ is not explicitly differentiable, DC3 applies the Implicit Function Theorem, which guarantees the existence of a differentiable function $\varphi_x(z)$ if the Jacobian of $h_x(y)$ with respect to the dependent variables is invertible. The derivative of $\varphi_x(z)$ with respect to z is:

$$\frac{\partial \varphi_x(z)}{\partial z} = -\left(J^h_{:,m:n}\right)^{-1} J^h_{:,0:m}, \tag{2.17}$$

where $J^h \in \mathbb{R}^{(n-m)\times n}$ is the Jacobian of $h_x(y)$ with respect to y, $J^h_{:,m:n}$ corresponds to the dependent variables, and $J^h_{:,0:m}$ corresponds to the independent variables.

The gradient of the loss with respect to z is computed as:

$$\frac{\partial \ell}{\partial z} = \frac{\partial \ell}{\partial z} + \frac{\partial \ell}{\partial \varphi_x(z)} \frac{\partial \varphi_x(z)}{\partial z}. \tag{2.18}$$

Substituting $\frac{\partial \varphi_x(z)}{\partial z}$, this becomes:

$$\frac{\partial \ell}{\partial z} = \frac{\partial \ell}{\partial z} - \frac{\partial \ell}{\partial \varphi_x(z)} \left(J^h_{:,m:n}\right)^{-1} J^h_{:,0:m}. \tag{2.19}$$

To improve efficiency, DC3 avoids computing the full Jacobian $\frac{\partial \varphi_x(z)}{\partial z}$ explicitly. Instead, the product $\frac{\partial \ell}{\partial \varphi_x(z)} \frac{\partial \varphi_x(z)}{\partial z}$ is computed directly, reducing computational and memory overhead while maintaining accurate gradient propagation. This ensures that training accounts for equality constraints while remaining computationally efficient.

Inequality Correction

In addition to completing equality constraints, DC3 enforces inequality constraints $g_x(y) \leq 0$ using a gradient-based correction procedure. This process is crucial because the completion procedure ensures feasibility with respect to equality constraints but does not inherently address inequality constraints. The correction step maps infeasible outputs into the feasible region iteratively, taking steps along the manifold defined by the equality constraints. This ensures that the final output satisfies both equality and inequality constraints.

DC3 defines a correction operation $\rho_x(y)$ to iteratively move the outputs $y = [z^T, \varphi_x(z)^T]^T$ closer to satisfying the inequality constraints. The operation is formally defined as:

$$\rho_x\left(\begin{bmatrix} z \\ \varphi_x(z) \end{bmatrix}\right) = \begin{bmatrix} z - \gamma \Delta z \\ \varphi_x(z) - \gamma \Delta \varphi_x(z) \end{bmatrix}, \tag{2.20}$$

where $\gamma > 0$ is the learning rate, and Δz and $\Delta \varphi_x(z)$ are corrections computed based on the gradient of the inequality violation.

The corrections Δz and $\Delta \varphi_x(z)$ are calculated using the gradients of the inequality violation:

$$\Delta z = \nabla_z \|\mathrm{ReLU}(g_x(y))\|_2^2, \tag{2.21}$$

$$\Delta \varphi_x(z) = \frac{\partial \varphi_x(z)}{\partial z} \Delta z. \tag{2.22}$$

Here:

- $g_x(y)$: The inequality constraint function, where $g_x(y) \leq 0$ represents feasibility.
- $\text{ReLU}(g_x(y))$: Penalizes violations by mapping positive violations to themselves and zero otherwise.
- $\frac{\partial \varphi_x(z)}{\partial z}$: The Jacobian of $\varphi_x(z)$ with respect to z, computed during the equality completion step.

The gradient-based correction moves z and $\varphi_x(z)$ closer to feasibility by iteratively reducing the violation measured by $\|\text{ReLU}(g_x(y))\|_2^2$.

At training time, DC3 minimizes a *soft loss*:

$$\ell_{\text{soft}}(y) = f_x(y) + \lambda_g \|\text{ReLU}(g_x(y))\|^2 + \lambda_h \|h_x(y)\|^2. \tag{2.23}$$

To compute $\lim_{t\to\infty} \rho_x^{(t)}(y)$, DC3 approximates this limit using $t = t_{\text{train}}$ for backpropagation and allows $t_{\text{test}} > t_{\text{train}}$ during testing to improve convergence when runtime permits.

The correction step effectively enforces feasibility for inequality constraints while maintaining computational efficiency, leveraging the Jacobian to ensure the outputs respect the equality manifold.

DC3's main advantage is the ability to embed constraint satisfaction directly into the forward pass of the network, avoiding the computational overhead involved in standard projection-based methods (as explored later in Chapter 3) and the limitations of penalty-based methodologies. This ensures compliance with constraints while maintaining the scalability required for large-scale, high-dimensional tasks.

The drawback of this method is that although it serves as an improvement over other techniques employed for such tasks, the constraint satisfaction is not guaranteed in general cases ([5]). The paper recommends different techniques to be employed for more niche applications, and isn't a one size fits all fix to the problem. Constraint satisfaction is mainly affected by the number of gradient steps and the step size, which require fine-tuning (according to the analysis in Giunchiglia et al.[6]).

2.3.2 Homogeneous Linear Inequality Constraints for Neural Network Activations

[7] address the challenge of enforcing homogeneous linear inequality constraints on neural network activations of the form:

$$Ax \leq 0, \tag{2.24}$$

where $A \in \mathbb{R}^{m \times d}$ encodes the constraints and $x \in \mathbb{R}^d$ is a vector of activations. These constraints ensure that neural network outputs always remain within a feasible region during both training and inference. By parameterizing this feasible region efficiently, the method eliminates the need for expensive projection steps or specialized solvers. Instead, it allows constraints to be enforced directly within the optimization process, leveraging standard stochastic gradient descent.

To represent the feasible set of activations, the polyhedron defined by $Ax \leq 0$ is transformed into a more flexible representation. For homogeneous constraints, the feasible set is:

$$C = \{z \in \mathbb{R}^d \mid Az \leq 0\}. \tag{2.25}$$

Using the decomposition theorem for polyhedra, this set can equivalently be written as:

$$C = \text{conv}(v_1, \ldots, v_n) + \text{cone}(r_1, \ldots, r_s), \tag{2.26}$$

where $\text{conv}(v_1, \ldots, v_n)$ is the convex hull of vertices $\{v_1, \ldots, v_n\}$ and $\text{cone}(r_1, \ldots, r_s)$ is the cone generated by rays $\{r_1, \ldots, r_s\}$. The convex hull represents the bounded part of the polyhedron, while the cone captures the unbounded directions. In the case of homogeneous constraints ($Ax \leq 0$), the feasible set simplifies to:

$$C = \text{cone}(r_1, \ldots, r_s), \tag{2.27}$$

as there are no bounded vertices when the constraints are homogeneous.

The decomposition theorem states that any polyhedron can be expressed as:

$$C = \left\{ z \mid z = \sum_{i=1}^{n} \lambda_i v_i + \sum_{j=1}^{s} \mu_j r_j,\ \lambda_i \geq 0,\ \mu_j \geq 0,\ \sum_{i=1}^{n} \lambda_i = 1 \right\}. \tag{2.28}$$

Here, λ_i are the convex combination parameters for the vertices v_i, and μ_j are the conical combination parameters for the rays r_j. The condition $\sum_{i=1}^{n} \lambda_i = 1$ ensures that the convex hull is properly

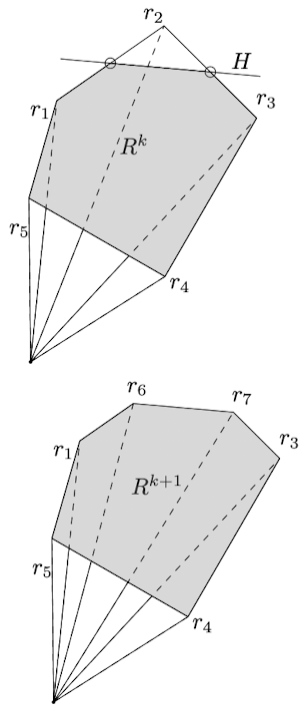


Figure 2.4: Illustration of the double description method.

defined. For homogeneous constraints, the convex combination parameters are not required, and the feasible set reduces to:

$$C = \left\{ z \mid z = \sum_{j=1}^{s} \mu_j r_j,\ \mu_j \geq 0 \right\}. \tag{2.29}$$

This representation captures the polyhedral cone as a combination of rays $\{r_1, \ldots, r_s\}$, where any point in the cone can be written as a non-negative linear combination of these rays.

The double description method is an iterative algorithm used to convert a system of linear inequalities from the H-representation to the V-representation. This process refines the feasible region step-by-step, adding one constraint (or half-space) at a time. At each step, the feasible region is represented as a polyhedral cone, characterized by a set of rays. Starting with an initial set of rays, each new constraint modifies the cone by adding new intersection rays and removing those that are no longer valid. This iterative process ensures that the feasible set is efficiently parameterized and remains compatible with gradient-based neural network training.

The current state of the system is represented by the pair (A^k, R^k), where A^k contains a subset of the constraints and R^k represents the rays defining the cone. When a new constraint is introduced, it defines a hyperplane H, which may cut through the existing polyhedron. This interaction between the hyperplane and the rays results in three possible outcomes:

- Rays that lie entirely within the new half-space remain unchanged.
- Rays that are intersected by the hyperplane generate new intersection rays, which are added to R^k.
- Rays that fall outside the new half-space are removed.

This process is illustrated in Figure 2.4. Initially, the polyhedral cone R^k is defined by rays r_1, r_2, r_3, r_4, r_5. When the hyperplane H is added:

- New rays r_6 and r_7 are created by calculating intersection points between H and the existing rays.
- Rays that no longer satisfy the constraint, such as r_2, are removed.

The updated polyhedron R^{k+1} satisfies all constraints introduced up to this point and is represented by the refined set of rays.

The paper describes how domain constraints are commonly expressed as box constraints, such as $\mathcal{B} := \{x \in \mathbb{R}^d \mid -1 \leq x_i \leq 1\}$, which define upper and lower limits for each dimension. For instance, in image generation tasks, pixel intensities are usually constrained within $[-1, 1]$. However, converting these constraints into the V-representation using the double description method becomes infeasible, as the number of vertices required grows exponentially with the dimension. To overcome this limitation, the method separates modeling constraints, which are converted into the V-representation, from domain constraints. Modeling constraints define the polyhedral cone, while domain constraints are handled separately. After computing a point x within the modeling constraint set $\mathcal{C}$, it is adjusted to also satisfy $\mathcal{B}$. This adjustment is achieved through normalization by the infinity norm:

$$\hat{x} = \frac{x}{\max\{\|x\|_\infty, 1\}}, \tag{2.30}$$

where $\|x\|_\infty = \max_i |x_i|$. This scaling ensures that the adjusted point $\hat{x}$ satisfies both $\mathcal{C}$ and $\mathcal{B}$, as scaling by a positive constant retains the point within the cone. In addition to this, the approach efficiently enforces domain constraints while avoiding the complexity of parameterizing the intersection of polyhedra and box constraints.

- If the point x already satisfies the box constraint $\mathcal{B}$, meaning $\|x\|_\infty \leq 1$, the normalization does nothing because $\max\{\|x\|_\infty, 1\} = 1$, and so $\hat{x} = x$. In this scenario, the point remains unchanged and continues to lie within both $\mathcal{C}$ and $\mathcal{B}$.

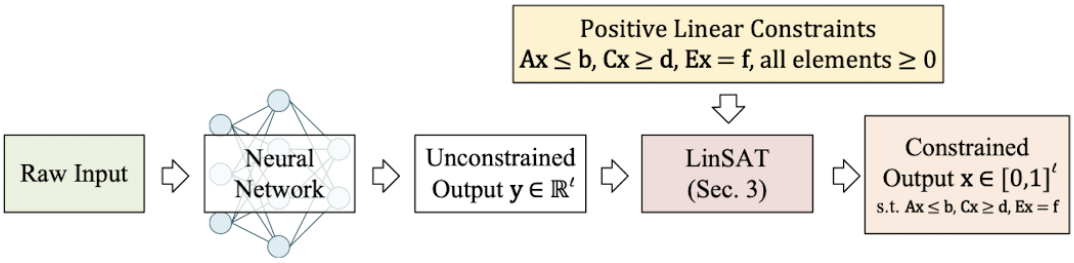


Figure 2.5: LinSATNet in action, enforcing positive linear constraints on the unconstrained output of a typical Neural Network [8]

- If x violates the box constraint, i.e., $\|x\|_\infty > 1$, the normalization scales x down by dividing it by $\|x\|_\infty$. This scaling ensures that the largest absolute value of any component of x is brought to exactly 1, satisfying the box constraint. Furthermore, since $\mathcal{C}$ is a cone, scaling a point within the cone by a positive constant retains it within the cone. Thus, the adjusted point $\hat{x}$ satisfies both constraints.

While the methodology described in [7] is effective, it can suffer from slow convergence due to vanishing gradients, when one parameter dominates significantly. In addition to this, integrating domain constraints (like the box constraints defined previously) in a general setting incurs extreme computational costs, because of the difficulty of parameterizing their intersection with polyhedra. Consequently, the double description method is most practical for constraints that are simple to model, like homogeneous or adjacent constraints.

2.3.3 LinSATNet

LinSATNet [8] proposes a novel approach for encoding positive linear constraints into neural networks. Figure 2.5 provides an overview of the layered architecture, illustrating how LinSATNet is structured and how it integrates cohesively with a typical neural network. Using an extension of the Sinkhorn algorithm [9], this framework enables seamless integration of constraints into the neural network architecture.

The Sinkhorn algorithm [9] is an iterative method for projecting a non-negative matrix onto the set of doubly stochastic matrices, where all rows and columns sum to one. This is achieved by alternately normalizing the rows and columns of the matrix. Given an initial non-negative matrix $S \in \mathbb{R}^{n \times n}$ and target row and column marginals $r, c \in \mathbb{R}^n$, the Sinkhorn algorithm updates the matrix to match these marginals as follows:

- The first update ensures that the rows of the matrix $S^{(k+1)}$ match the target row marginals r. This is achieved by:

$$S_{ij}^{(k+1)} = S_{ij}^{(k)} \cdot \frac{r_i}{\sum_j S_{ij}^{(k)}} \tag{2.31}$$

  Here, for each row i, the current row sum is calculated ($\sum_j S_{ij}^{(k)}$), and each element in the row is scaled by the ratio of the target row marginal r_i to the current row sum. This operation adjusts the matrix so that the row sums align with r.

- After row normalization, the second update ensures that the columns of the matrix $S^{(k+2)}$ match the target column marginals c. This is achieved by:

$$S_{ij}^{(k+2)} = S_{ij}^{(k+1)} \cdot \frac{c_j}{\sum_i S_{ij}^{(k+1)}} \tag{2.32}$$

  Here, for each column j, the current column sum is calculated ($\sum_i S_{ij}^{(k+1)}$), and each element in the column is scaled by the ratio of the target column marginal c_j to the current column sum. This operation adjusts the matrix so that the column sums align with c.

- The algorithm alternates between these two steps until the matrix converges, i.e., the row and column sums simultaneously match the target marginals r and c within a specified tolerance.

The primary contribution of this method is the generalization of said algorithm to handle multiple sets of marginal distributions (defined as the "multi-set" Sinkhorn algorithm). This generalization preserves the convergence properties of the original single-set version and provides a theoretical foundation for enforcing *positive linear constraints* in differentiable neural networks. These constraints are expressed as:

- $Ax \leq b$: *Packing constraints* to ensure the total usage of resources does not exceed available capacities.
- $Cx \geq d$: *Covering constraints* to guarantee minimum thresholds are met.
- $Ex = f$: *Equality constraints* to enforce balancing or conservation requirements.

Here, all elements of A, b, C, d, E and f are non-negative [8].

The multi-set Sinkhorn algorithm extends the classical Sinkhorn method to handle multiple sets of marginal distributions simultaneously. Given a non-negative matrix $S \in \mathbb{R}^{m \times n}$ and k sets of marginal distributions, each consisting of a row marginal $r^{(\eta)} \in \mathbb{R}^m$ and a column marginal

$c^{(\eta)} \in \mathbb{R}^n$ for $\eta = 1, \ldots, k$. The algorithm iteratively refines S to approximately satisfy all k constraint sets. The iterative process involves:

- The matrix S is initially normalized to ensure its total sum is one:

$$S_{ij}^{(0)} = \frac{S_{ij}}{\sum_{i,j} S_{ij}}. \tag{2.33}$$

- For the current marginal set $\eta = (t \mod k) + 1$, the algorithm scales the rows of S to match the row marginals $r^{(\eta)}$. This is achieved through:

$$S_{ij}^{(t+1)} = S_{ij}^{(t)} \cdot \frac{r_i^{(\eta)}}{\sum_{j=1}^{n} S_{ij}^{(t)}}. \tag{2.34}$$

- After row normalization, the columns of S are adjusted to match the column marginals $c^{(\eta)}$. This is done using:

$$S_{ij}^{(t+2)} = S_{ij}^{(t+1)} \cdot \frac{c_j^{(\eta)}}{\sum_{i=1}^{m} S_{ij}^{(t+1)}}. \tag{2.35}$$

- The algorithm alternates between these row and column normalization steps in a cyclic manner. At each iteration t, the marginal set $\eta = (t \mod k) + 1$ is selected, and the matrix is sequentially normalized with respect to the rows and columns of that set. This process ensures that all marginal sets are updated iteratively, and the algorithm continues until the approximation error (e.g., KL divergence or L_1 distance) converges to a pre-defined threshold.

This variation retains the convergence properties of the classical Sinkhorn method and ensures approximate satisfaction of all marginal constraints.

To enable the multi-set Sinkhorn algorithm to operate on the neural network's output, the vector $\mathbf{y} \in \mathbb{R}^l$, representing the network's predictions, must be encoded into a form compatible with the normalization process. This is achieved by constructing a matrix W of size $2 \times (l + 1)$, defined as:

$$W = \begin{bmatrix} y_1 & y_2 & \ldots & y_l & \beta \\ \beta & \beta & \ldots & \beta & \beta \end{bmatrix}, \tag{2.36}$$

where $\mathbf{y}$ is placed in the upper-left portion of W, and the remaining entries are filled with a dummy variable β. The parameter β can be set to zero or another small constant, depending

on the specific application. This matrix construction allows the neural network's output to be reformulated as a score matrix that can represent marginal distributions when normalized.

To ensure that the entries of W are non-negative and to handle any potential negative values in $\mathbf{y}$, the matrix is exponentiated as:

$$S = \exp\left(\frac{W}{\tau}\right), \tag{2.37}$$

where $\tau > 0$ is a regularization parameter controlling the smoothness of the output. The regularizer helps mitigate the effects of extreme values in W, ensuring that the matrix is well-conditioned for iterative normalization. The resulting matrix S is then passed as input to the multi-set Sinkhorn algorithm, as described above, where it undergoes row and column normalization to enforce the specified marginal constraints.

The LinSAT layer offers a practical and theoretically grounded mechanism for enforcing positive linear constraints within neural network architectures. Rather than incorporating constraints into the loss function via penalty terms or dual variables, LinSAT enforces them through a parameter-free, differentiable layer that operates directly on the network's output. This approach allows constraint satisfaction to be treated as part of the forward pass, rather than as a post hoc optimization step. Although the constraints are not explicitly coupled with the learning objective, they still exert an implicit influence on the optimization process, since the corrected (feasible) outputs are used to compute the loss and propagate gradients. By supporting a wide class of constraints (packing, equality and covering), LinSAT enables end-to-end training of decision models that can reliably produce feasible outputs. Its reliance on matrix-vector arithmetic ensures compatibility with modern auto-diff frameworks, making it both scalable and efficient.

The multi-set Sinkhorn algorithm, which serves as the basis of LinSATNet [8], has some limitations. Although it is theoretically guaranteed to converge to a solution that satisfies all marginal constraints, this convergence occurs only asymptotically—that is, in the limit of infinite iterations. In practice, the algorithm is terminated when an approximation error metric, such as the KL divergence or the L_1-distance between the achieved and target marginals, falls below a predefined threshold (see Equations (4a) and (4b), and Theorem 3.2 in [8]). This introduces a trade-off between computational cost and the precision of constraint satisfaction.

Moreover, as noted in Section 3.2 of [8], the convergence rate is sensitive to the initialization of the input matrix S, due to its influence on the convergence parameter $\alpha = \min_{i,j:S_{ij}>0} \frac{S_{ij}}{\max_{i,j} S_{ij}}$.

Poorly conditioned initializations (e.g., sparse or highly skewed matrices) can degrade convergence speed.

3

Analysis and Design

Contents

This chapter presents the two methods explored to improve the quality of gradient backpropagation through the shield layer, under linear constraints. The first method tweaks the loss function and the formulation of the clamping operations, attempting to exploit the underlying mechanics of these to work in favorr of back-propagation optimization. The second approach uses conventional geometric optimization techniques such as projection, to better guide the gradients.

3.1 Mask and Modified MSE

3.1.1 Formalism and Notation

Let $\mathbf{h} = (h_1, \ldots, h_n)$ denote the prediction vector output by a neural network, and let $\mathbf{y} = (y_1, \ldots, y_n)$ denote the corresponding ground-truth target vector. We consider linear inequal-

ity constraints that relate a left-hand-side (LHS) variable to a linear combination of a subset of other predicted variables, denoted the right-hand side (RHS).

Each constraint takes the following form:

$$h_L \geq \sum_{j \in \text{RHS}} a_j h_j + b, \tag{3.1}$$

where:

- $h_L \in \mathbf{h}$ is the LHS variable,
- RHS $\subset \{1, \ldots, n\}$ indexes the active RHS variables,
- $a_j \in \mathbb{R}$ are the associated linear weights, and
- $b \in \mathbb{R}$ is a scalar offset.

3.1.2 Initial Formulation

As discussed in Chapter 1, the shield layer ensures constraint satisfaction by clamping predictions to a feasible region. However, this clamping distorts gradient flow during training. When a prediction appears in both the constraint and the supervised loss, it may receive competing gradient signals, undermining convergence.

To address this, we introduce two mechanisms:

1. A masking strategy to isolate the gradient path to a single RHS variable per constraint.
2. A directional correction using sign terms to align gradient updates with the supervised target.

3.1.3 The Need for Directional Gradient Correction

We now revisit the issue of conflicting gradients introduced in Chapter 1 by considering a constraint of the form:

$$h_A \geq h_B + h_C, \tag{3.2}$$

To enforce this constraint during training, the shield layer computes a masked correction of the

RHS. For a given binary mask $\mathbf{m} = [m_B, m_C] \in \{0,1\}^2$, the corrected value for the LHS is:

$$h'_A = \max\left(h_A,\ m_B h_B + (1-m_B)y_B + m_C h_C + (1-m_C)y_C\right). \tag{3.3}$$

This formulation allows only a single RHS variable to receive gradient updates. For example, with $\mathbf{m} = [1, 0]$, we select h_B as the active variable and substitute h_C with y_C. The clamped value becomes:

$$h'_A = \max(h_A,\ h_B + y_C). \tag{3.4}$$

If the new state of the constraint is violated (i.e., $h_A < h_B + y_C$), the shield layer outputs:

$$h'_A = h_B + y_C. \tag{3.5}$$

The total loss (mean-square error) is defined as:

$$\mathcal{L} = (h'_A - y_A)^2 + (h_B - y_B)^2 + (h_C - y_C)^2. \tag{3.6}$$

Substituting the shielded value, the loss becomes:

$$\mathcal{L} = (h_B + y_C - y_A)^2 + (h_B - y_B)^2 + (h_C - y_C)^2. \tag{3.7}$$

Taking the gradient with respect to the active variable h_B, we obtain:

$$\frac{\partial \mathcal{L}}{\partial h_B} = 2(h_B + y_C - y_A) + 2(h_B - y_B). \tag{3.8}$$

This expression contains two components:

- The first term, $2(h_B + y_C - y_A)$, enforces constraint satisfaction,
- The second term, $2(h_B - y_B)$, aligns h_B with its ground-truth target.

When these terms act in opposite directions, for example, when $h_B < y_B$ but $h_B + y_C > y_A$ - the update for h_B becomes unstable. This gradient conflict mirrors the behavior in Equation 1.4, where the same variable participates in both constraint enforcement and prediction accuracy maximisation, leading to competing update signals.

3.1.4 Combining Masks and Sign Correction

To correct for this, we augment the loss with directional sign terms. For a single constraint, the loss is defined as:

$$\mathcal{L}_{AB} = (h'_A - y_A)^2 \cdot \text{sgn}(h'_A - y_A) \cdot \text{sgn}(h_B - y_B), \tag{3.9}$$

where h'_A is computed using the masked correction from Equation 3.3.

Let $z = h'_A - y_A$. Then the gradient with respect to the active variable h_B is:

$$\begin{aligned} \frac{\partial \mathcal{L}_{AB}}{\partial h_B} &= \frac{\partial}{\partial h_B} \left[z^2 \cdot \text{sgn}(z) \cdot \text{sgn}(h_B - y_B) \right] \\ &= 2|z| \cdot \text{sgn}(h_B - y_B). \end{aligned} \tag{3.10}$$

The gradient's direction is determined by the supervised error of the active variable, $\text{sgn}(h_B - y_B)$, and the magnitude is scaled by the severity of the constraint violation, $|h'_A - y_A|$. The update thus balances both constraint feasibility and prediction accuracy.

3.1.5 General Case with Structured Masking

In the general case, let there be m constraints and n prediction variables. Define a binary mask matrix $\mathbf{M} \in \{0,1\}^{m \times n}$, where each row $\mathbf{m}_i$ selects a single RHS variable $h_{j^*(i)}$ for constraint i. This means $M_{ij^*(i)} = 1$, and all other entries in row i are 0 for RHS variables (one-hot encoded masks)

Let $h_{L(i)}$ denote the LHS variable for constraint i, and define:

$$h_{L(i)} \geq \sum_{j \in \text{RHS}(i)} a_{ij} h_j. \tag{3.11}$$

We define a masked RHS value:

$$g_i = \sum_{j \in \text{RHS}(i)} a_{ij} \left(M_{ij} h_j + (1 - M_{ij}) y_j \right), \tag{3.12}$$

which simplifies to:

$$g_i = a_{i,j^*(i)} h_{j^*(i)} + \sum_{j \neq j^*(i)} a_{ij} y_j. \tag{3.13}$$

Let $\mathcal{C}_k$ denote the set of constraints in which variable h_k appears on the LHS. We define the shielded value of h_k as:

$$\hat{h}_k = \max\left(h_k, \max_{c \in \mathcal{C}_k} g_c\right). \tag{3.14}$$

For each constraint i, we define:

$$h'_{L(i),i} = \max(h_{L(i)}, g_i), \tag{3.15}$$

and let $y_{L(i)}$ be the target value for the LHS variable. The full loss is then:

$$\mathcal{L} = \sum_{i=1}^{m} \left(h'_{L(i),i} - y_{L(i)}\right)^2 \cdot \operatorname{sgn}(h'_{L(i),i} - y_{L(i)}) \cdot \operatorname{sgn}(h_{j^*(i)} - y_{j^*(i)}). \tag{3.16}$$

This formulation ensures that:

- Gradient updates are routed through a single selected RHS variable per constraint.
- The sign terms enforce directional alignment with both constraint violation and prediction error.
- Violated constraints with higher degrees of violation dominate the corrective signals, while satisfied ones contribute minimally.

3.1.6 Known Limitations

Potential negative loss (without a proven lower bound)

The masked and signed loss in Equation 3.16 includes a directional correction via sign terms. While the squared error component

$$(h'_{L(i),i} - y_{L(i)})^2 \tag{3.17}$$

is always non-negative, the product with

$$\operatorname{sgn}(h'_{L(i),i} - y_{L(i)}) \cdot \operatorname{sgn}(h_{j^*(i)} - y_{j^*(i)}) \tag{3.18}$$

can yield a final scalar loss term that is negative. This results in a global loss function $\mathcal{L}$ that is potentially not bounded below by zero

This behavior breaks a key assumption of optimizers such as Adam [10], which are designed under the expectation that the loss function is bounded below, and exhibits a monotonic relationship between parameter updates and performance improvement. Reddi et al. confirm this, in their OpenReview submission, by stating that **"convergence guarantees for Adam require the loss to be lower-bounded"** [11]. When the loss is allowed to take arbitrarily negative values, it is effectively bounded below by $-\infty$, which makes optimization unstable.

Many standard, well-behaved loss functions can take negative values. Their stability comes from having a well-defined floor, which provides the optimizer with a clear, achievable objective. For example, in representation learning, a primary goal is to map inputs to a vector space where similar items have similar embeddings. A widely used objective function is the negative cosine similarity (also used in Chapter 5), which encourages the angle between the vector embeddings of a pair (v_1, v_2) to be as small as possible. The loss is defined as:

$$\mathcal{L}_{\text{cosine}} = -\frac{v_1 \cdot v_2}{\|v_1\|\|v_2\|} \tag{3.19}$$

The range of this loss function is $[-1, 1]$. It is frequently negative during training, and is explicitly bounded below by -1 [12].

The key insight here is that the traditional mean-square error loss function is bounded by 0. This is not the case for an "inverted" version of the function.

Breaking guaranteed constraint satisfaction during training

The original design of the shield layer guaranteed that all model predictions satisfied the imposed linear constraints by clamping outputs to a feasible region.

However, the introduction of the masking strategy changes this. Instead of using only predicted values on the right-hand side (RHS) of a constraint, the masking mechanism substitutes all but one with their corresponding ground-truth targets. This design aims to isolate gradient flow to a single variable, enabling cleaner and more stable updates. Yet, this substitution breaks the guarantee that the final prediction vector satisfies the original constraint.

Consider a constraint of the form:

$$h_A \geq h_B + h_C. \tag{3.20}$$

Suppose h_B is chosen as the active RHS variable. Under masking, h_C is replaced by its ground-truth value y_C, and the constraint enforced by the shield becomes:

$$h'_A \geq h_B + y_C. \tag{3.21}$$

The shield outputs:

$$h'_A = \max(h_A,\ h_B + y_C). \tag{3.22}$$

This replacement implies that the shield ensures feasibility only with respect to the modified constraint, not the original one. The guaranteed constraint changes to:

$$h'_A \geq h_B + y_C \quad \text{but not necessarily} \quad h'_A \geq h_B + h_C. \tag{3.23}$$

Constraint satisfaction of the original form can be violated during training if the model's prediction h_C exceeds y_C. In such cases, we have:

$$h_B + y_C < h_B + h_C, \tag{3.24}$$

which means:

$$h'_A \geq h_B + y_C \ \not\Rightarrow\ h'_A \geq h_B + h_C. \tag{3.25}$$

This discrepancy implies that even after applying the shield, the output h'_A may fail to satisfy the original constraint if the predicted value h_C exceeds its ground-truth target y_C.

It is important to note, however, that masking is only applied during training. At inference time, the full prediction vector is used in constraint enforcement without substitution. As such, the original guarantees of constraint satisfaction are reinstated at test time, provided the learned predictions lie within the feasible region.

3.2 Projected Gradients

3.2.1 Clamping Operations and their effect on the gradient

A key insight in the application of the shield layer is that when a constraint becomes active, the mechanism enforces feasibility by applying a clamping operation to the model's unconstrained

predictions.

Constraints of the form $h_i \geq h_j$ are enforced by replacing the original prediction h_i^{raw} with the corrected value $\hat{h}_i = \max(h_i^{\text{raw}}, h_j^{\text{raw}})$, while constraints of the form $h_i \leq h_j$ use $\hat{h}_i = \min(h_i^{\text{raw}}, h_j^{\text{raw}})$. These operations ensure that the final output $\hat{h}_i$ strictly satisfies the required inequality before it is passed to the loss function.

It is this clamping operation that introduces instability during backpropagation. When using the chain rule, the total gradient with respect to a variable h_i^{raw} is computed as:

$$\frac{\partial \mathcal{L}}{\partial h_i^{\text{raw}}} = \frac{\partial \mathcal{L}}{\partial \hat{h}_i} \cdot \frac{\partial \hat{h}_i}{\partial h_i^{\text{raw}}}. \tag{3.26}$$

Suppose the constraint is active such that $h_i^{\text{raw}} < h_j^{\text{raw}}$. Then $\hat{h}_i = h_j^{\text{raw}}$, and:

$$\frac{\partial \hat{h}_i}{\partial h_i^{\text{raw}}} = 0. \tag{3.27}$$

As a result,

$$\frac{\partial \mathcal{L}}{\partial h_i^{\text{raw}}} = \frac{\partial \mathcal{L}}{\partial \hat{h}_i} \cdot 0 = 0, \tag{3.28}$$

which blocks gradient flow to h_i^{raw} entirely.

This behavior effectively removes h_i^{raw} from the learning process when the constraint is active, even though it contributed to the original prediction. Meanwhile, h_j^{raw} receives the full gradient signal.

The clamping operation, while enforcing feasibility in the forward pass, is not differentiable and thus does not provide a meaningful gradient in the backward pass. In its default form, `autograd` sets the derivative of functions like max() or min() to zero with respect to the inactive argument, as shown earlier.

3.2.2 Straight Through Estimators (Autograd)

To allow gradient flow through such non-differentiable operations, a common workaround is the use of a straight-through estimator (STE) [13]. In this approach, we assume:

$$\frac{\partial \hat{h}_i}{\partial h_i^{\text{raw}}} \approx 1 \quad \text{even if } \hat{h}_i = \max(h_i^{\text{raw}}, h_j^{\text{raw}}), \tag{3.29}$$

thereby enabling gradient propagation through the clamping function as if it were the identity. This is typically implemented by detaching the forward projection and manually overriding the backward gradient, treating the gradient from the shield layer as an identity function, instead of the sparse gradients obtained from the max() and min() functions.

While STE enables learning in the presence of discrete or clamped operations, it does not take the constraint surface into account during optimization. That is, the gradient $\frac{\partial \mathcal{L}}{\partial h_i^{\text{raw}}}$ is computed as if the prediction $\hat{h}_i$ were unconstrained, even though it was forcibly projected into a feasible region by the shield layer. The effect of this method, as well as the mitigation will be presented in the next section.

The optimizer effectively ignores the constraint, since it does not adjust its updates based on the underlying half-space. This can cause instability during training, particularly when learning rates are large enough to cause frequent re-violation of the constraint set.

3.2.3 Projected Gradient Correction

Before presenting the method used to improve the quality of gradient updates, it is essential to define some key terminology used in this section.

- **Active constraints** are exactly satisfied at a given prediction. For a set of linear inequality constraints of the form $Ay \leq b$, a constraint is said to be active at a point y if it satisfies the following condition:

$$A_i y = b_i \tag{3.30}$$

  for some row A_i of the matrix $A \in \mathbb{R}^{C \times N}$, where C is the number of constraints and N is the number of output variables. The vector $y \in \mathbb{R}^N$ contains the model's predictions, and $b \in \mathbb{R}^C$ is the corresponding bias vector. Using active constraints is paramount in limiting the scope of the projection problem, the core of which is inspired by the active set method [14], [15].

- **Tangent space** refers to the set of directions in which the prediction vector y can move infinitesimally without violating the currently active constraints. For a set of active constraints indexed by $\mathcal{I}_{\text{active}}$, the tangent space $\mathcal{T}_y$ at point y is defined as:

$$\mathcal{T}_y = \{v \in \mathbb{R}^n \mid A_i v = 0 \text{ for all } i \in \mathcal{I}_{\text{active}}\}. \tag{3.31}$$

This space consists of all update directions that lie along the boundary of the feasible region, preserving constraint satisfaction [15].

Although the actual implementation uses the Adam optimizer [10], we adopt the standard stochastic gradient descent (SGD) update rule [16] , in the following derivation for clarity and analytical convenience. In such settings, the two optimizers produce similar update directions, making SGD a reasonable approximation. The SGD update rule for model parameters $\theta \in \mathbb{R}^d$ at iteration t is given by:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta \mathcal{L}(\theta_t), \tag{3.32}$$

where $\eta > 0$ is the learning rate and $\mathcal{L}$ is the loss function.

Constraining the Output Step

To ensure that the parameter update in Equation 3.32 results in an output that remains within the constraint surface, we focus on the change in the output prediction, rather than the change in parameters. Since the constraints are enforced on the output vector y, it is vital that after applying Δy the new output value of y does not then violate constraints.

It is essential to analyze how the change in parameters θ affects the model's output y, which is the point at which constraints are enforced.

Define:

$$y = f(x; \theta) \tag{3.33}$$

where $f : \mathbb{R}^d \to \mathbb{R}^n$ is the composite function of the neural network that maps the parameters $\theta \in \mathbb{R}^d$, based on the inputs x, to the raw predictions $y \in \mathbb{R}^n$. These predictions are constrained by $Ay \leq b$, and sany update to θ must ensure that the updated prediction y_{new} remains in the feasible region, to prevent undoing the corrections from the shield layer (and the opposing gradients problem highlighted in Chapter 1).

To understand the effect of a parameter update $\Delta\theta$, we approximate the new prediction using a Taylor expansion of f around the current point θ:

$$f(\theta + \Delta\theta) \approx f(\theta) + J \cdot \Delta\theta \tag{3.34}$$

where $J = \frac{\partial f(\theta)}{\partial \theta} = \frac{\partial y}{\partial \theta} \in \mathbb{R}^{n \times d}$ is the Jacobian matrix of the outputs with respect to the parameters.

We can write the updated prediction as:

$$y_{\text{new}} \approx y + \Delta y = y + J \cdot \Delta\theta \tag{3.35}$$

This expression gives us a linear approximation of how the prediction vector changes due to a small step in the parameter space. Due to the update rule in equation 3.32, it is apparent that the step is incremental, and small enough that the taylor's approximation holds, provided that the learning rate is also small (typical range between 1e-3 and 1e-6). This is typically the case in well-formed and well-trained networks because it promotes stable convergence. Substituting the parameter update rule:

$$\Delta y = -\eta \cdot J \cdot \frac{\partial \mathcal{L}}{\partial y} \tag{3.36}$$

To prevent constraint violations, this change must lie entirely within the tangent space defined by the constraints at the current prediction. That is, for any active constraint $A_i y = b_i$, we must have:

$$A_i \cdot \Delta y = 0 \quad \forall i \in \mathcal{I}_{\text{active}} \tag{3.37}$$

In matrix form, this condition becomes:

$$A_{\text{active}} \cdot \Delta y = -\eta \cdot A_{\text{active}} \cdot J \cdot \frac{\partial \mathcal{L}}{\partial y} = 0 \tag{3.38}$$

This can be achieved by projecting the raw gradient obtained from the shield layer, $\frac{\partial \mathcal{L}}{\partial y}$, onto the null-space of $A_{\text{active}} \cdot J$ to ensure that the resulting update direction lies within the constraint-preserving tangent space. Let P denote the corresponding projection matrix. Then the final corrected update becomes:

$$\Delta y = -\eta \cdot P \cdot J \cdot \frac{\partial \mathcal{L}}{\partial y} \tag{3.39}$$

This ensures that the parameter update induces a change in y that stays within the constraint manifold.

Note: By definition, for a particular batch item, A_{active} is also the set of constraints that were part of the shield layer clamping process. Variables that violate one or more constraints get clamped to the boundary of the feasible region, where they exactly satisfy their most restrictive constraints as equalities.

Gradient Correction

The previous section highlights the potential benefits and performance improvements that can be gained via the gradient correction process. As mentioned in section 3.2.2, the forward pass of the shield layer proceeds as normal, producing h, which corresponds to the set of outputs respecting the entire constraint set A (note that A is used interchangeably as the constraint set and constraint tensor), from the output y generated by the preceding neural network.

In practice, the Jacobian $J = \frac{\partial y}{\partial \theta} \in \mathbb{R}^{n \times d}$ is not available during automatic differentiation, as it lies earlier in the computational graph than the straight-through estimator (STE) applied during the shield layer's forward projection. This limitation prevents us from computing the composite matrix $A_{\text{active}}J$, which would otherwise be used to project the raw gradient into the tangent space of the feasible set.

As a result, we adopt a simplifying assumption: that the gradient $g = \frac{\partial \mathcal{L}}{\partial y} \in \mathbb{R}^n$ already lies in the null space of the active constraint matrix, i.e.,

$$A_{\text{active}} g = 0. \tag{3.40}$$

This assumption enables us to analyze the feasibility of parameter updates without needing access to J directly. In the context of stochastic gradient descent (SGD), the parameter update for one step is:

$$\Delta\theta = -\eta \frac{\partial \mathcal{L}}{\partial \theta} = -\eta J^\top g, \tag{3.41}$$

$$\text{where} \quad J = \frac{\partial y}{\partial \theta}, \quad g = \frac{\partial \mathcal{L}}{\partial y}. \tag{3.42}$$

The corresponding change in the output prediction y is then approximated by:

$$\Delta y = J \cdot \Delta\theta = -\eta J J^\top g. \tag{3.43}$$

To maintain feasibility after the update, we require that Δy lies in the null space of A_{active}, i.e.,

$$A_{\text{active}} \Delta y = -\eta A_{\text{active}} J J^\top g \approx 0. \tag{3.44}$$

Substituting the identity matrix I, we rewrite this as:

$$A_{\text{active}}\Delta y = -\eta A_{\text{active}}(JJ^\top - I + I)g \tag{3.45}$$

$$= -\eta A_{\text{active}}(JJ^\top - I)g - \eta A_{\text{active}}g. \tag{3.46}$$

Given our assumption $A_{\text{active}}g = 0$, the second term vanishes, leaving:

$$A_{\text{active}}\Delta y = -\eta A_{\text{active}}(JJ^\top - I)g. \tag{3.47}$$

This expression reveals that constraint violations are second-order and controlled by the deviation of $JJ^\top$ from the identity. Provided J is well-conditioned and η is small, this residual remains negligible in practice. It is therefore important to consider the spectral behavior of J. In well-conditioned neural networks, the singular values of J often remain close to one, which helps ensure that signal propagation remains stable during both forward and backward passes. This property is known as dynamical isometry [17], [18].

Although exact dynamical isometry is not a strict requirement for the correctness of the method, it helps ensure that $JJ^\top \approx I$. Prior work has displayed that this condition can be encouraged through architectural choices such as orthogonal weight initialization [19] and the use of batch or layer normalization.

Deriving the projection

Given the assumption in the previous subsection, we want to find a vector g_{proj} that lies in the tangent space defined by the active constraints, i.e.,

$$A_{\text{active}}g_{\text{proj}} = 0.$$

Our goal is to find the vector g_{proj} that is closest to a given vector g while satisfying this constraint. Formally, this corresponds to the optimization problem

$$\min_{g_{\text{proj}}} \|g_{\text{proj}} - g\|, \quad \text{such that} \quad A_{\text{active}}g_{\text{proj}} = 0.$$

Since minimizing the norm is equivalent to minimizing its squared value, and for mathematical

convenience, we consider the equivalent problem

$$\min_{g_{\text{proj}}} \frac{1}{2}\|g_{\text{proj}} - g\|^2, \quad \text{such that} \quad A_{\text{active}} g_{\text{proj}} = 0.$$

This quadratic objective is differentiable and convex, and the factor $\frac{1}{2}$ is introduced purely for convenience, as it does not affect the location of the minimizer. To solve this, we use a Lagrange multiplier [15] vector λ, and construct the Lagrangian:

$$\mathcal{L}(g_{\text{proj}}, \lambda) = \frac{1}{2}\|g_{\text{proj}} - g\|^2 + \lambda^\top A_{\text{active}} g_{\text{proj}} \tag{3.48}$$

Taking the gradient of $\mathcal{L}$ with respect to g_{proj} and setting it to zero yields the optimality condition:

$$g_{\text{proj}} - g + A_{\text{active}}^\top \lambda = 0 \tag{3.49}$$

Substituting this into the constraint $A_{\text{active}} g_{\text{proj}} = 0$, we obtain the linear system:

$$A_{\text{active}} A_{\text{active}}^\top \lambda = A_{\text{active}} g \tag{3.50}$$

Solving for λ gives the optimal multiplier, and substituting back yields the projected gradient:

$$g_{\text{proj}} = g - A_{\text{active}}^\top (A_{\text{active}} A_{\text{active}}^\top)^{-1} A_{\text{active}} g \tag{3.51}$$

In practice, the matrix $A_{\text{active}} A_{\text{active}}^\top$ may be ill-conditioned or singular, particularly when active constraints are redundant or nearly linearly dependent. To address this, we can apply regularization [15] by adding a small multiple of the identity matrix:

$$g_{\text{proj}} = g - A_{\text{active}}^\top (A_{\text{active}} A_{\text{active}}^\top + \alpha I)^{-1} A_{\text{active}} g \tag{3.52}$$

where $\alpha > 0$ is a tunable parameter that stabilizes the inversion.

4

Implementation

Contents

Building upon the theoretical foundations laid out in Chapter 3, this chapter presents the practical implementation of the two constraint-aware training strategies: the **Masked** method and the **Projected Gradient Correction** method. Although explicit code listings are not provided, the discussion focuses on algorithmic design, implementation details, and relevant trade-offs encountered during development.

4.1 Mask and Modified MSE

This section details the implementation of the masking-based training strategy introduced in Section 3.1. The method modifies the standard shield layer by introducing a masking mechanism

that uses ground-truth values to pin selected variables while adjusting certain others one at a time. The key idea is to select a single variable per constraint to remain active, allowing gradients to flow through it, while the others are clamped to their ground-truth values. This selection is made *randomly* during each forward pass, introducing a controlled source of stochasticity into the optimization process.

4.1.1 Shield Layer Algorithmic Overview

To realize the mechanism described in Chapters 2 and 3, Stoian et al. [3] provide an implementation of the `ShieldLayer`, which relies on pre-computation for efficiency. During initialization, the layer parses the set of linear inequality constraints and compiles them into structured matrices that capture the relationships between variables. Specifically, for each variable x_i in the correction ordering, two matrices are constructed:

- A positive matrix M_i^+ representing all constraints that impose a lower bound on x_i,
- A negative matrix M_i^- representing all constraints that impose an upper bound on x_i.

Each row in these matrices corresponds to a single constraint, and the columns contain the normalized coefficients of the other variables and the constant offset (bias). This design allows the calculation of all potential bounds for x_i across an entire batch of predictions $\mathbf{h}$ to be performed with a single, highly optimized matrix multiplication.

The final lower bound lb_i and upper bound ub_i are then found by applying a reduction. As the effective bound is the most restrictive one, the implementation uses a `max` reduction over the results from the positive matrix and a `min` reduction for the negative matrix. This matrix-based approach transforms the complex task of solving systems of inequalities into efficient linear algebra, making it feasible for modern hardware.

4.1.2 Pre-processing of Constraints

The shield layer was originally developed for unsupervised tabular data generation tasks involving linear constraints. In such unsupervised settings, all variables are treated symmetrically, and no distinction is made between inputs and targets. However, in scenarios involving regression, this distinction becomes essential.

To reconcile this difference, a pre-processing routine was introduced to identify which variables in the constraint expressions are to be treated as prediction targets. The procedure accepts both an ordered list of variables and a set of linear constraints relying on the API provided by Stoian et al. [1]. It then extracts those variables that appear in constraint expressions, classifying them as output variables, while treating all others as inputs. For example:

```
ordering y1, y2, y3
y1 - y2 > 0
```

In this case, only `y1` and `y2` appear in the constraints and are thus designated as target variables. `y3` is considered an input feature. This logic is applied programmatically; the parser identifies all variables referenced in the constraint expressions and constructs a minimal set of output variables accordingly. To support efficient indexing and downstream compatibility, these variables are remapped to a compact zero-indexed form (e.g., `y1`, `y3`, `y7` become `y0'`, `y1'`, `y2'`). A reverse mapping is retained to recover the original variable names for post-processing.

4.1.3 The Masked Forward Pass

The core of the proposed method is a modification to the shield's forward pass that is active only during training. The goal is to create a more informative gradient signal by selectively using ground-truth information when calculating the corrective bounds. This is achieved through a masking mechanism in which, for each constraint, a single variable is randomly selected to remain active, while the others are clamped. This random selection is performed independently for each constraint during each forward pass, introducing controlled stochasticity into the training process.

When the `ShieldLayer` processes a variable x_i, it computes the associated bounds using a blended input that mixes predictions $\mathbf{h}$ with ground-truth targets $\mathbf{y}$. For each multi-variable constraint that contributes to a bound on x_i, the method selects one of the other participating variables to remain active. All remaining variables involved in the constraint are masked by substituting their predicted values with their corresponding ground-truth targets.

The inputs for the bound calculation are then assembled as follows:

- The network's prediction is used for the single active variable.
- The ground-truth values are used for all other variables in the constraint.

The masking operation is implemented using PyTorch's tensor broadcasting and `where` operations. For a given constraint, the logic of creating the masked input vector is equivalent to:

$$\mathbf{h}_{\text{masked}} = \mathbf{m} \odot \mathbf{h} + (1 - \mathbf{m}) \odot \mathbf{y},$$

where $\odot$ denotes element-wise multiplication. This blended tensor is then used in the bound calculations that follow.

After masking is applied, the shield computes the effective lower and upper bounds on each variable h_i using the precomputed constraint matrices M_{pos} and M_{neg}. A max-reduction is applied to the masked inputs with the positive matrix to obtain the tightest lower bound, and a min-reduction is applied with the negative matrix to obtain the tightest upper bound.

The final corrected value is obtained by clamping the prediction:

$$h_i' = \min\left(\max(h_i, \text{lb}_i), \text{ub}_i\right).$$

This logic is summarized in Algorithm 1.

Algorithm 1: The Masked Shield Forward Pass

Input: Predictions $\mathbf{h}$, Ground Truth $\mathbf{y}$, Constraint Matrices $\mathcal{M}$

Output: Corrected predictions $\mathbf{h}'$

$\mathbf{h}_{\text{corr}} \leftarrow \mathbf{h}.clone()$;

for *each variable* h_i *in the constraint ordering* λ **do**

  $M_{\text{pos}}, M_{\text{neg}} \leftarrow \mathcal{M}[h_i]$;

  if *training mode is active* **then**

    $\mathbf{h}_{\text{masked\_pos}} \leftarrow \text{CreateMaskedInput}(\mathbf{h}_{\text{corr}}, \mathbf{y}, M_{\text{pos}})$;

    $\mathbf{h}_{\text{masked\_neg}} \leftarrow \text{CreateMaskedInput}(\mathbf{h}_{\text{corr}}, \mathbf{y}, M_{\text{neg}})$;

  else

    $\mathbf{h}_{\text{masked\_pos}} \leftarrow \mathbf{h}_{\text{corr}}$;

    $\mathbf{h}_{\text{masked\_neg}} \leftarrow \mathbf{h}_{\text{corr}}$;

  end

  $\text{lb}_i \leftarrow \text{CalculateBound}(M_{\text{pos}}, \mathbf{h}_{\text{masked\_pos}}, \text{reduction='max'})$;

  $\text{ub}_i \leftarrow \text{CalculateBound}(M_{\text{neg}}, \mathbf{h}_{\text{masked\_neg}}, \text{reduction='min'})$;

  $h'_i \leftarrow \min(\max(h_i, \text{lb}_i), \text{ub}_i)$;

  $\mathbf{h}_{\text{corr}}[i] \leftarrow h'_i$;

end

return $\mathbf{h}_{corr}$

This masking mechanism has an important effect beyond bound computation: it shapes the gradient flow during training. By substituting the predictions of all non-selected variables in each constraint with ground-truth values, it effectively blocks gradient flow through those variables. Since ground-truth targets are treated as constants by the autograd system, no derivative flows through them. Consequently, only the active variable in the mask receives a non-zero gradient signal from the constraint correction.

This leads to a clean and directed learning signal. Rather than having gradients propagate through all terms in a multi-variable constraint (possibly in conflicting directions) the network receives clearer feedback, as illustrated above.

4.1.4 Adjusted Loss Function

In tandem with the masking strategy introduced in the forward pass, a custom loss function is employed to ensure that the gradient signal remains coherent with the underlying constraint

logic. This function, called `adjusted_constraint_loss`, modifies the standard mean squared error (MSE) calculation on a per-variable basis, depending on whether a masking adjustment was applied during the forward pass.

The function takes as input the model predictions, ground-truth targets, and a dictionary of masking matrices that track which variables were involved in each constraint. Specifically, the dictionary maps output indices i to binary mask tensors of shape (B, d), where B is the batch size and d is the output dimensionality. If the mask is set to `None`, the corresponding variable was not involved in any constraint adjustment, and a standard squared loss term is used. If a valid tensor is provided, then the mask specifies which other variables' errors should influence the gradient direction for the i-th output.

Let $\widehat{\mathbf{y}}$ denote the model's predictions and $\mathbf{y}$ the ground-truth targets. The basic error signal is computed as:

$$\mathbf{e} = \widehat{\mathbf{y}} - \mathbf{y}, \quad \mathbf{s} = \operatorname{sign}(\mathbf{e}), \quad \mathbf{L}_{\text{raw}} = \mathbf{e}^2.$$

Each variable's contribution to the total loss is initially given by $\mathbf{L}_{\text{raw}}$, the element-wise squared error. However, for each output index i with a valid adjustment mask $\mathbf{m}_i$, a further correction is applied: for each non-zero entry $\mathbf{m}_i[j, k] = 1$, the error sign of variable k in sample j is used to modify the loss term associated with variable i. This process is represented by:

$$L'_{j,i} \leftarrow L_{j,i} \cdot \operatorname{sign}(e_{j,i}) \cdot \operatorname{sign}(e_{j,k}).$$

This adjustment flips the sign of the gradient only when the active and influencing variables disagree in error direction, effectively damping updates that would otherwise move predictions further out of alignment. It ensures that, in adjusted terms, the gradient with respect to variable i flows in the same direction it would have flowed under an unmasked shield, while respecting the masking pattern imposed in the forward pass.

Finally, the adjusted loss tensor is aggregated using the specified reduction method (by taking the mean or sum) ensuring compliance with the expectation of standard PyTorch loss functions.

4.1.5 Implementation Trade-offs

While the masking strategy ensures localized gradient flow and prevents interference between variables involved in the same constraint, its stochastic nature introduces a non-trivial source

of training noise. For each constraint, one of the participating variables is randomly selected to remain active, while the others are clamped to their ground-truth values. This selection is performed independently across constraints and samples in each forward pass. As a result, two identical prediction vectors may yield different loss signals and gradient directions solely due to a different masking configuration. Consequently, the optimizer may struggle to attribute performance fluctuations to the network's parameters, potentially confusing masking effects with model-induced behavior.

Additionally, the current implementation is challenging to fully vectorized. The masking logic relies on per-sample, per-constraint combinatorial decisions based on which variables appear in the constraint expressions. The resulting computational bottleneck contrasts with the otherwise fully vectorized design of the shield layer and the neural network training pipeline.

Another limitation, as discussed in section 3.1.6, is the potential for the adjusted loss function to produce negative values. This behavior interacts poorly with conventional machine learning optimizers, which often assume non-negative losses (as previously discussed).

4.2 Projected Gradient Correction

The Projected Gradient Correction mechanism enforces constraint satisfaction during training by altering gradient directions to align with the geometry of the feasible set. This is accomplished through a custom PyTorch module that integrates seamlessly into the computational graph.

The core idea is to modify each sample's gradients such that they lie in the null space of its active constraint set. By projecting onto the tangent space of the feasible region, the corrected gradients ensure that optimization steps do not violate constraint boundaries, thus preserving feasibility throughout the training process.

The implementation prioritizes both numerical stability and computational efficiency. Specifically, the projection operations are performed independently for each sample, preventing cross-sample interference and allowing fine-grained control over constraint satisfaction. Batched matrix operations are employed wherever possible to maintain scalability.

4.2.1 Algorithm Overview

The overall goal of the implementation is to ensure that, during optimization, the neural network's outputs remain feasible with respect to a set of linear constraints. This is achieved by:

1. Projecting the raw network outputs to the feasible set during the forward pass.
2. Projecting the gradients onto the tangent space of the active constraints during the backward pass.

A high-level pseudocode for the projected gradient correction is as follows:

Algorithm 2: Projected Gradient Correction Procedure

Input: Predictions y, Constraints (A, b)

Output: Corrected outputs $\tilde{y}$, corrected gradients g_{proj}

`// Forward pass`

$\tilde{y} \leftarrow \text{ShieldLayer}(y, A, b)$;

Save $(\tilde{y}, A, b)$ for backward pass;

`// Backward pass`

Given incoming gradient g;

Identify active constraints based on $A\tilde{y}$ and b;

Solve $A_{\text{active}} A_{\text{active}}^{\top} \lambda = A_{\text{active}} g$;

Compute $g_{\text{proj}} = g - A_{\text{active}}^{\top} \lambda$;

4.2.2 Forward Pass: Constraint Enforcement

The forward pass enforces constraint feasibility by correcting the network's raw predictions using a *shield layer*. The shield layer as described in 1 and [1], clamps outputs to the feasible set defined by a system of linear inequalities:

$$A\tilde{y} \leq b, \tag{4.1}$$

where $\tilde{y}$ denotes the corrected output. It operates by adjusting the predictions minimally to satisfy all active constraints.

In this implementation, the shield operation is performed without gradient tracking via a `torch.no_grad()` context, since the gradients are manually handled during the backward pass.

The corrected outputs $\tilde{y}$, along with the constraint matrix A and bias vector b, are saved in the PyTorch autograd context for use during the backward computation.

Choice of Tolerance Parameter ϵ A tolerance ϵ is introduced to account for numerical inaccuracies when determining active constraints. ϵ can be tuned within the API, and was selected to be the same value as the PiShield package [3].

4.2.3 Constraint Matrix and Bias Vector Construction

Constraints are supplied as text files specifying inequalities of the form:

$$\sum_{k=1}^{\mathcal{D}} w_k x_k + b \trianglerighteq 0, \tag{4.2}$$

where $\trianglerighteq \in \{\geq, >, \leq, <\}$. To convert these constraints into a machine-readable form suitable for efficient computation, each constraint is parsed and mapped into:

- A row in the constraint matrix $A \in \mathbb{R}^{m \times \mathcal{D}}$, where m is the number of constraints and $\mathcal{D}$ is the number of output variables.
- A corresponding element in the bias vector $b \in \mathbb{R}^m$.

The construction process initializes a zero row vector for each constraint and assigns coefficients to the appropriate indices based on the constraint specification. Inequalities are standardized: for constraints specified as $\geq$ or $>$, the row and bias are multiplied by -1 to convert them into the standard form $Ax \leq b$. For strict inequalities ($<$ or $>$), a small numerical offset ε is added or subtracted from the bias term to enforce feasibility [3]. Once all constraints are processed, the rows are stacked to form the global matrix A, and the biases are collected into the vector b.

4.2.4 Backward Pass: Tangent Space Projection

The backward pass modifies the incoming gradients to ensure that updates stay within the tangent space of the feasible set.

Active Set Identification

The active constraints for each sample are identified by checking:

$$|A\tilde{y}_i - b| \leq \epsilon, \tag{4.3}$$

where $\tilde{y}_i$ is the corrected output for sample i. This identifies constraints that are tight up to the numerical tolerance ϵ.

Projection via Lagrange Multipliers

Once the active constraints are identified, the incoming gradient g is projected onto the tangent space by solving:

$$A_{\text{active}} A_{\text{active}}^{\top} \lambda = A_{\text{active}} g, \tag{4.4}$$

where λ are the Lagrange multipliers. The projected gradient is then given by:

$$g_{\text{proj}} = g - A_{\text{active}}^{\top} \lambda. \tag{4.5}$$

This ensures that the projected gradient lies within the feasible tangent space, preventing infeasible updates.

4.2.5 Numerical Stability Considerations

The linear system in the projection step can be ill-conditioned if the active constraints are nearly linearly dependent. The condition number κ of the system matrix:

$$\text{lhs} = A_{\text{active}} A_{\text{active}}^{\top} \tag{4.6}$$

is computed as:

$$\kappa(\text{lhs}) = \|\text{lhs}\| \cdot \|\text{lhs}^{-1}\|, \tag{4.7}$$

where $\|\cdot\|$ denotes the spectral norm [20].

If the condition number exceeds a threshold value of 10^7, a regularization term is added to the system:

$$\text{lhs} \leftarrow \text{lhs} + \alpha I, \tag{4.8}$$

where α is typically set to 10^{-7}. This improves the conditioning of the matrix and prevents numerical instability. If the system remains ill-conditioned after regularization, the projection problem is solved using a robust least-squares solver (`torch.linalg.lstsq`) rather than a direct solver. The threshold for the condition number was set to 10^7, following common practice in machine learning applications to avoid numerical instability. This choice is supported by the analysis of floating-point error magnification proportional to the condition number, as discussed in [21].

4.2.6 Custom Autograd Function

The projection correction mechanism is encapsulated within a custom `torch.autograd.Function` referred to as `ProjGradSTE`. This function overrides PyTorch's automatic differentiation mechanism by supplying customized `forward` and `backward` methods that implement the correction logic described above.

4.2.7 Implementation Trade-offs

The implementation performs projection independently for each sample in a batch. This design ensures correctness, as different samples may activate different constraint sets, making a single global projection infeasible. However, this per-sample projection limits the degree of parallelism that can be exploited on GPUs. In particular, while standard neural network operations can leverage batch matrix operations for acceleration, the sample-wise correction introduces a serial bottleneck during the backward pass.

From a computational complexity perspective, solving the projection for each sample involves a system of size $m \times m$, where m is the number of active constraints. The dominant cost per sample scales cubically as $O(m^3)$, making the overall complexity for a batch of size B proportional to $O(Bm^3)$. This cost becomes significant when the number of active constraints exceeds approximately 50 to 100 per sample, depending on hardware capabilities and the efficiency of the underlying linear algebra routines.

Despite the trade-offs involved, the chosen approach strikes a balance between theoretical correctness and computational feasibility.

5

Experimental Analysis

Contents

This chapter presents an analysis of the methods proposed in Chapter 3, and implemented in Chapter 4. To ensure reproducibilty of the results, the complete source code and instructions are provided in Appendix A.

5.1 Experimental Setup

5.1.1 Models Compared

Three models were evaluated to assess the efficacy of the proposed methods. The **Baseline MLP** is a standard multi-layer perceptron (MLP) trained without constraint enforcement via the shield layer and serves as a reference point for performance The **Shielded MLP** augments the baseline with a shield layer that enforces linear constraints during the forward pass. This serves as a

reference point for constraint enforcement when evaluating the architectures/techniques. The **Improved Shielded MLP** is a blanket term for the two enhancements being tested:

- Projected gradients using a straight-through estimator.
- Masked shield layer with masked mean square error.

All models share identical architectural hyper-parameters so that any observed differences arise solely from the constraint-enforcement strategy.

5.1.2 Model Architectures

Two base architectures were employed as the baseline MLPs. The **Shallow MLP** contains three fully connected layers with hidden dimensions 32 and 64. LeakyReLU activations are chosen to mitigate the *dying-neuron* problem, discovered by [22]. This allows even negative inputs to receive a non-zero gradient and remain trainable:

$$f(x) = \begin{cases} x, & x \geq 0, \\ \alpha x, & x < 0, \end{cases}$$

with $\alpha = 0.01$ in the default PyTorch implementation.

Unlike sigmoidal activations, which are prone to saturation and vanishing gradients in deep models, `LeakyReLU` activations maintain non-saturating behavior and provide non-zero gradients throughout the input space. This choice helps ensure stable backpropagation and faster convergence in deeper architectures. Although ReLU-family activations cannot achieve perfect dynamical isometry at initialization, their practical advantages, including mitigation of vanishing gradients, support for sparse activations, and avoidance of dead neurons, outweigh the theoretical benefits of `Sigmoid` in large-scale settings. Recent theoretical work on residual architectures shows that dynamical isometry can be achieved universally across activation functions when proper scaling is used [23].

The **Deep MLP** extends this structure with hidden layers of 64, 128, and 128 units and inserts a Layer Normalisation layer after each activation. Increasing the depth and width of each layer helps concentrate the singular values of the input-output Jacobian around one, improving gradient and helps approximate *dynamical isometry*, promoting stable gradient propagation and supporting

the projected-gradient descent described in Chapter 3. In line with the conditions for this discussed in section 3.2.3 this MLP is initialized using orthogonal weight matrices and zero biases.

5.1.3 Training Configuration

Training was conducted using the Adam optimizer for all data-sets, with a fixed learning rate of 10^{-3} across 30 epochs, unless otherwise specified. In addition to this, a batch size of 64 was used across all experiments. For models employing constraints, linear inequality conditions were parsed from standardized configuration files as discussed in section 4.1.2, and constraint satisfaction was monitored throughout training and evaluation. For models incorporating the masking mechanism, dynamic masks were computed during the forward pass and applied during loss computation. The specifics for this can be found in the accompanying code repository, linked in Appendix A.

5.1.4 Evaluation Metrics

Model performance was evaluated using the following key metrics:

- **RMSE** RMSE is a standard metric used to evaluate the accuracy of a model by computing the square root of the average squared differences between the predicted outputs and the true target values. It is most commonly employed in regression tasks. For a set of n predictions $\{\hat{y}_i\}$ and ground-truth values $\{y_i\}$, RMSE is defined as:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}. \tag{5.1}$$

- **Constraint Satisfaction Rate:** This measures the proportion of predictions that satisfy all specified linear constraints. It is an indicator of how well the model maintains feasibility with respect to the constraint set. A high satisfaction rate implies that the model consistently produces outputs that respect the given constraints.

- **Statistical Significance Testing:** To assess whether performance improvements between the Shielded MLP and the Improved Shielded MLP are statistically significant, paired t-tests were conducted. As described by Kutner, Nachtsheim, Neter, *et al.* (2005), the paired t-test compares the means of two related groups across multiple trials, testing the null hypothesis

that their population means are equal. The test statistic t for a paired t-test is calculated as:

$$t = \frac{\bar{d} - \mu_d}{s_d/\sqrt{n}}$$

Here, $\bar{d}$ represents the mean of the differences between the paired observations (e.g., the performance difference between the two MLP variants for each random seed). μ_d is the hypothesized population mean difference under the null hypothesis, which is typically set to 0 (indicating no difference). The term s_d is the standard deviation of these differences, and n is the number of paired observations, corresponding to the 10 random seeds. The null hypothesis is rejected if the computed p-value (derived from this t-statistic) falls below the significance threshold $\alpha = 0.05$, indicating that the observed differences are unlikely to have occurred by chance.

- **Effect Size (Cohen's *d*):** While statistical significance tests determine the likelihood that an observed event occurred by chance, effect size quantifies the magnitude or strength of that effect. Cohen's d is computed as the difference between the sample means of two models divided by their pooled standard deviation. Let $\overline{x}_1$ and $\overline{x}_2$ denote the sample means of Model 1 and Model 2 across multiple runs, respectively. Then, Cohen's d [25] is defined as:

$$d = \frac{\overline{x}_1 - \overline{x}_2}{s_p}, \tag{5.2}$$

where s_p is the pooled standard deviation, computed from the standard deviations and sample sizes of the two groups. Larger values of d indicate stronger effects, with conventional thresholds (originally proposed by Cohen in [25]) suggesting that $d = 0.2$ represents a small effect, $d = 0.5$ a medium effect, and $d = 0.8$ a large effect.

- **Time per Epoch:** To assess the computational efficiency and scalability of the proposed methods, the average time required to complete a single training epoch was measured. This metric captures the runtime overhead introduced by the additional constraint-handling mechanisms and serves as a practical indicator of each method's computational cost.

- **Gradient Direction Consistency (Cosine Similarity):** Another key indicator of success for backpropagation improvement is *Gradient Alignment*. This parameter measures how close, in both sign and magnitude, the average gradient from a particular method is to the baseline gradient. Gradient alignment is quantified using cosine similarity between the corrected gradient and the original baseline gradient. For the purposes of this section, the

baseline is chosen as the MLP's raw gradient when using mean-square error loss. Cosine similarity between two tensors (which are treated as vectors for this calculation) $\mathbf{u}$ and $\mathbf{v}$ is defined as:

$$\cos(\theta) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|} \tag{5.3}$$

where θ is the angle between the two vectors. The value of cosine similarity ranges from -1 to 1. A value of 1 signifies perfect alignment, meaning the vectors point in the same direction. In the context of gradients, this implies the corrective mechanism fully preserves the original optimization direction. Conversely, a value of -1 indicates that the vectors point in exactly opposite directions, while a value of 0 means the vectors are orthogonal, indicating no linear relationship in their direction. Therefore, a cosine similarity close to 1 is the most desirable outcome for each data-point. Lower or negative values, indicate greater deviation from the original gradient direction and potential optimization inefficiency.

Note: To ensure robust and reliable assessment, all experimental configurations were repeated across 10 distinct random seeds. Final results for each metric are reported as the mean and standard deviation across these repetitions.

5.1.5 Evaluation Pipeline

The evaluation pipeline was designed in two stages. First, the performance of each method was assessed by full model training, measuring root mean squared error and constraint satisfaction rates across multiple random seeds for each dataset. Second, gradient alignment was evaluated at initialization to isolate the intrinsic effects of each constraint handling method on optimization dynamics, independent of model parameter updates. Since gradient directions are primarily determined by the architecture and loss landscape at initialization and not by training progress, measuring cosine similarity on the first 100 batches provides a statistically robust yet computationally efficient estimate of gradient alignment, without incurring excessive runtime overhead.

For reproducibility, the complete set of command-line scripts used for performance evaluation and gradient alignment analysis has been made publicly available. Further details, including repository access and usage instructions for this whole chapter, are provided in the appendix A.

5.1.6 Datasets

This section evaluates the proposed methods on four datasets: URL, News, LCLD, and Faulty Steel Plates. The first three are sourced from the MOEVA dataset collection, which is available via Figshare[1], while the Faulty Steel Plates dataset originates from the UCI Machine Learning Repository[2]. These datasets were selected due to their alignment with the constraint-learning tasks studied in Stoian et al. [3][3].

The MOEVA datasets (URL, News, and LCLD) originate from real-world applications in cybersecurity and classification and were previously used in adversarial robustness research by Simonetto et al. [26]. They are characterized by structured tabular features and include between five and ten inequality constraints. The Faulty Steel Plates dataset, by contrast, is a regression dataset concerning physical measurements of manufacturing defects. Although it is unconstrained in its original form, the accompanying constraint files enable it to be repurposed for multi-output regression under linear inequality conditions

All constraint files used in this thesis follow the structure defined by Stoian et al. [3] and are included in their public repository found in Appendix A.

Preprocessing and Transformation

The datasets were processed through a unified transformation pipeline. Each dataset was paired with a linear inequality constraint file, which defines a set of target variables and constraint expressions in a format consistent with the code accompanying Stoian et al [1], as described in section 4.1.2.

The pipeline also supports `.npy` data files as input, which can be specified using the `--numpy-data` and `--data-list` flags.

Input features are partitioned into continuous and categorical subsets. Continuous features are identified heuristically based on cardinality: any feature with more than two unique values is treated as continuous. While this rule-of-thumb may misclassify edge cases, it is sufficient for this comparative study, where the objective is not to optimize performance but to evaluate the performance of several constraint-enforcing strategies under identical conditions.

[1] `https://figshare.com/s/84ae808ce6999fafd192?file=42088389`
[2] `https://www.kaggle.com/datasets/uciml/faulty-steel-plates`
[3] `https://github.com/mihaela-stoian/ConstrainedDGM/tree/main/data`

Continuous features are standardized to zero mean and unit variance using a `StandardScaler`. Text-based features, inferred from columns with string data types, are mapped to fixed-size integer buckets via hashing.

The dataset is then partitioned into training, validation, and test sets using a 70-10-20 split. Prior to this, an optional sampling mechanism allows a configurable fraction of the training data to be used, which is particularly useful for quick verification on large datasets.

All features and targets are converted into PyTorch tensors, and the resulting splits are loaded using reproducible `DataLoader` instances. Constraints parsed from the configuration files are passed into the training loop and used to enforce feasibility during both the forward and backward passes, ensuring that constraint satisfaction is preserved throughout training regardless of model architecture.

The entire pre-processing pipeline is modular and dataset-agnostic, and is used uniformly across all experiments.

5.2 Results and Discussion

5.2.1 Test Set Performance

Table 5.1 summarizes the test set RMSE performance and computational efficiency of the proposed methods compared to the Shielded MLP across multiple datasets and architectures. The table reports the mean RMSE, the percentage change relative to the Shielded MLP, the statistical significance (p-value) of the improvement, the effect size (Cohen's d), and the average training time per epoch.

Table 5.1: Comparison of Test Set RMSE, Statistical Significance, Effect Size, and Training Time per Trial across Methods. Negative values in the % Change column indicate an improvement over the Shielded MLP baseline.

Dataset	Architecture	Method	Shielded RMSE	Method RMSE	% Change	p-value	Cohen's d	Shielded Time (s)	Method Time (s)
URL	Shallow	Projected	24.3193	697929.6969	2869760.5	0.0224	0.87	38.3	209.67
URL	Deep	Projected	11.6844	11.5025	-1.56	0.0	-4.092	39.98	103.38
Faulty Steel Plates	Shallow	Projected	1317167.55	1316893.0875	-0.02	0.0815	-0.62	5.31	35.78
Faulty Steel Plates	Deep	Projected	1322607.925	1322609.95	0.0	0.0001	2.149	9.49	38.77
LCLD	Shallow	Projected	7987.5182	8942.0342	11.95	0.0361	0.778	288.33	2233.63
LCLD	Deep	Projected	7911.995	7891.153	-0.26	0.19	-0.448	334.11	1425.68
News	Shallow	Projected	11703.1976	11455.604	-2.12	0.3447	-0.315	108.72	709.68
News	Deep	Projected	14388.523	14324.6592	-0.44	0.0	-16.925	120.72	714.99
URL	Shallow	Masked	24.3193	20.8442	-14.29	0.4746	-0.236	37.25	320.12
URL	Deep	Masked	11.6844	14.1661	21.24	0.0	12.144	40.15	323.87
Faulty Steel Plates	Shallow	Masked	1317167.55	1317283.8625	0.01	0.6392	0.153	5.36	18.11
Faulty Steel Plates	Deep	Masked	1322607.925	1322608.125	0.0	0.4482	0.251	7.24	39.23
News	Shallow	Masked	11703.1976	15321.2551	30.92	0.0	5.492	107.41	407.19
News	Deep	Masked	14388.523	15300.3178	6.34	0.0	40.829	121.35	418.62

The results summarized in Table 5.1 demonstrate that the performance of the proposed gradient correction methods is sensitive to network depth and architectural capacity.

The projected gradient method produced statistically significant reductions in test set RMSE for deep architectures on the URL and NEWS datasets. For the URL-DEEP configuration, RMSE decreased from 11.6844 to 11.5025, representing a -1.56% improvement with a p-value below 0.001 and a large effect size ($d = -4.092$). Similarly, for the NEWS-DEEP model, RMSE dropped by -0.44% from 14388.523 to 14324.6592, again with a significant p-value (< 0.001) and a very large effect size ($d = -16.925$). These results suggest that projected gradients offer a stable optimization signal when used with sufficiently expressive models that approximate dynamical isometry. The method preserves feasibility constraints while improving convergence properties.

However, this benefit does not extend to all architectures. In shallow networks, the same method led to severe performance degradations. On the URL-SHALLOW model, RMSE increased dramatically from 24.3193 to 697929.6969 which is a staggering increase of over 2.8 million% with $p = 0.0224$ and a moderate effect size ($d = 0.87$). A similar degradation occurred in the LCLD-SHALLOW model, where RMSE increased by 11.95% ($p = 0.0361$, $d = 0.778$).

These failure modes highlight the method's limitations when applied to networks that lack sufficient capacity. The reduced depth of shallow models likely contributes to ill-conditioned Jacobians, making them more sensitive to projection-induced distortions. The observed instability is therefore consistent with theoretical expectations regarding the fragility of constrained gradient flows in poorly conditioned regimes.

The computational cost of the projection step presents an additional hurdle. For the LCLD-SHALLOW model, average time per epoch increased from 288.33 s to 2233.63 s which is almost an $8\times$ increase. Even in the more stable deep networks, the overhead remains substantial. The NEWS-DEEP model required 714.99 s per epoch with projection compared to 120.72 s without. This result clearly indicates the scalability problems highlighted in section 4.2.7.

The masked MSE method, failed to produce statistically significant improvements across any of the evaluated datasets. Instead, it frequently led to performance degradation. On the URL-DEEP model, RMSE increased by 21.24% ($p < 0.001$, $d = 12.144$). For NEWS-DEEP, the degradation was 6.34% with a large effect size ($d = 40.829$). Similar trends emerged in the shallow NEWS architecture, where RMSE rose by 30.92% ($p < 0.001$, $d = 5.492$).

These results suggest that the masked approach introduces instability into the optimization

process. As described in Chapter 3, the mechanism relies on randomly selecting a single output to propagate gradients through, which introduces high variance into the update dynamics. Furthermore, the modified MSE that may produce negative values disrupts standard convergence behavior.

Despite its conceptual simplicity, the masked method also incurs meaningful computational overhead. On the NEWS-DEEP configuration, epoch time increased from 121.35 s to 418.62 s which is more than a 3× increase.

5.2.2 Gradient Direction Consistency (alignment) and Constraint Satisfaction

To evaluate the optimization behavior of each method independent of prolonged training, gradient alignment and constraint satisfaction were measured on the first 100 batches of the training run. Since both metrics stabilize quickly, limiting the evaluation to a small amount of batches instead of going through multiple epochs, provides a reliable yet computationally efficient estimate.

Gradient alignment between the corrected gradients and the baseline MSE gradients at initialization is reported in Table 5.2. Re-stating the objective of this metric, cosine similarity values closer to 1 indicate better preservation of the original optimization direction.

Table 5.2: Cosine Similarity between Baseline Gradients and Corrected Gradients at Initialization.

Dataset	Shielded MLP	Masked Shielded MLP	Projected Gradients
URL	0.0355 ± 0.0025	0.3624 ± 0.0464	0.995 ± 0.0016
Faulty Steel Plates	0.788 ± 0.0698	0.788 ± 0.0698	0.9962 ± 0.006
LCLD	1.0 ± 0.0	1.0 ± 0.0	0.9382 ± 0.0366
News	0.7743 ± 0.1726	0.7743 ± 0.1726	0.9539 ± 0.0448
Botnet	0.8159 ± 0.2801	0.9332 ± 0.1255	0.9916 ± 0.0229

The mean number of constraint violations on the test set is reported in Table 5.3. Lower values indicate better compliance with the specified linear convex constraints.

From table 5.2, the problem with regards to poor gradient alignment is apparent in the raw shield layer. The Projected Gradients method is effective in exacerbating this issue, and consistently outperforms both the Shielded MLP and the Masked Shielded MLP in terms of gradient alignment. Across all datasets, it achieves near-perfect cosine similarity with the baseline gradients. This is in line with expectations; the essence of the method is finding the closest gradient lying within the tangent space of the original raw gradient in the back-propagation step.

Table 5.3: Mean Constraint Violations on Test Set.

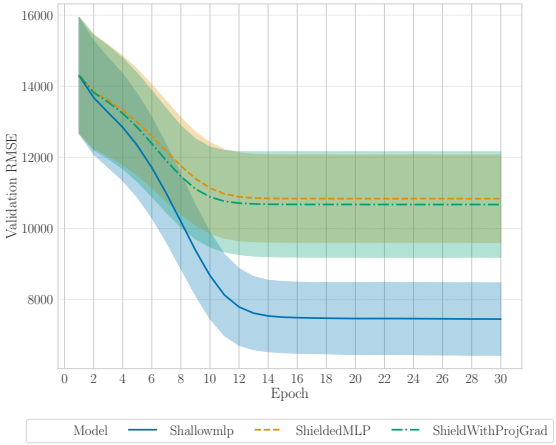
Dataset	Baseline MLP	Shielded MLP	Masked Shielded MLP	Projected Gradients
URL	3.0 ± 0.0	0.0 ± 0.0	0.8333 ± 0.6969	0.0 ± 0.0
Faulty Steel Plates	3.8571 ± 0.378	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
LCLD	3.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
News	5.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Botnet	362.0 ± 0.0	0.0 ± 0.0	4.0 ± 0.0	0.0 ± 0.0

With respect to constraint violations highlighted in table 5.3, both the Shielded MLP and the Projected Gradients method achieve perfect satisfaction across all datasets, validating their effectiveness in enforcing feasibility. However, the baseline MLP, which lacks any constraint mechanism, exhibits substantial violations, particularly on complex datasets like Botnet.

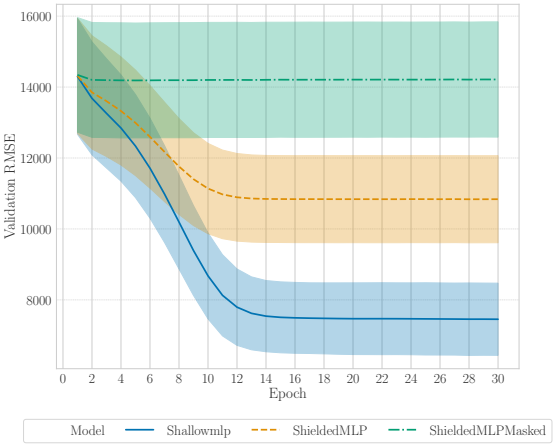
The Masked Shielded MLP reduces violations relative to the baseline but fails to eliminate them entirely, the reasons for which were presented in the limitations of the method laid out in section 3.1.6.

5.2.3 Training Stability

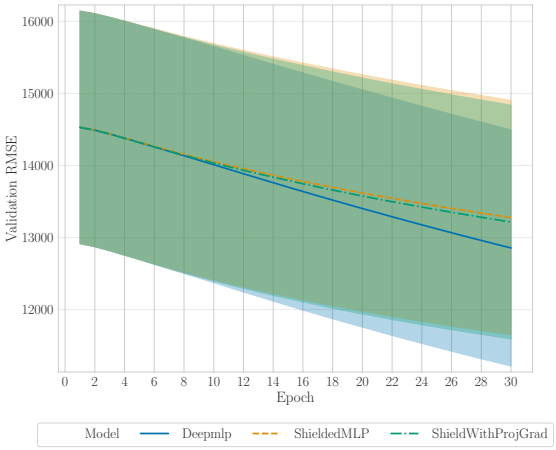
This section details the training stability of the four architectures/methods over 30 epochs. The mean and standard deviation are calculated across the 10 seeds, with the standard deviation being shaded around the mean. This is presented in the figures below.



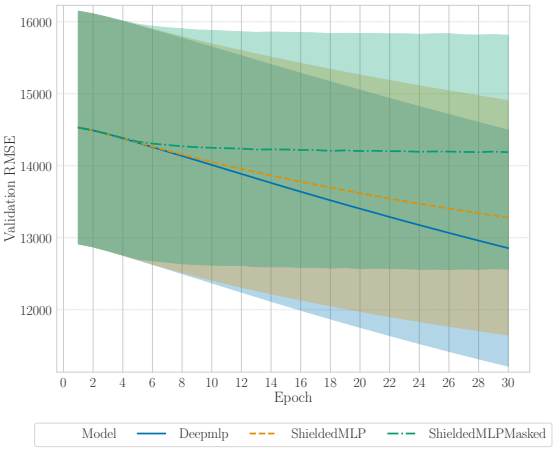
(a) Shallow architecture with projection



(b) Shallow architecture with masked training



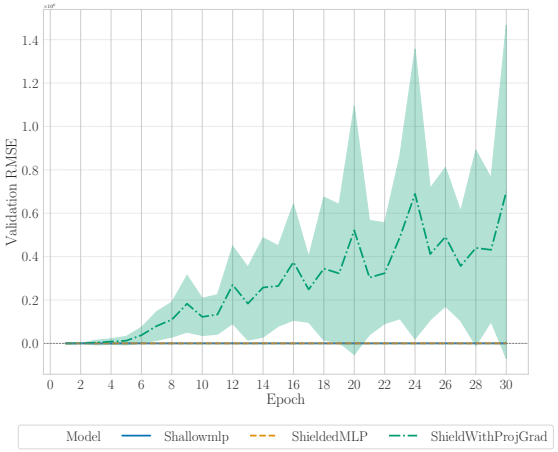
(c) Deep architecture with projection



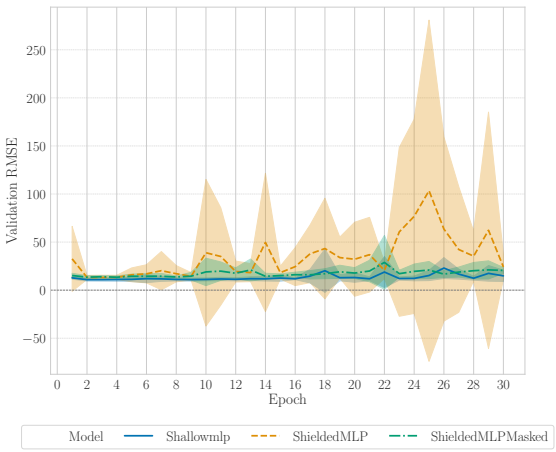
(d) Deep architecture with masked training

Figure 5.1: Training stability on the NEWS dataset across shallow and deep architectures.

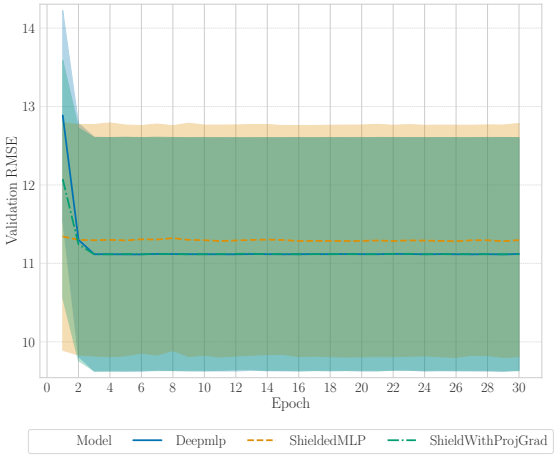
As shown in Figure 5.1, the projected gradient method exhibits greater training stability across runs compared to the masking-based approach. Moreover, the final RMSE achieved by the projected method is consistently closer to that of the unconstrained MLP baseline than that obtained using the mask-based method.



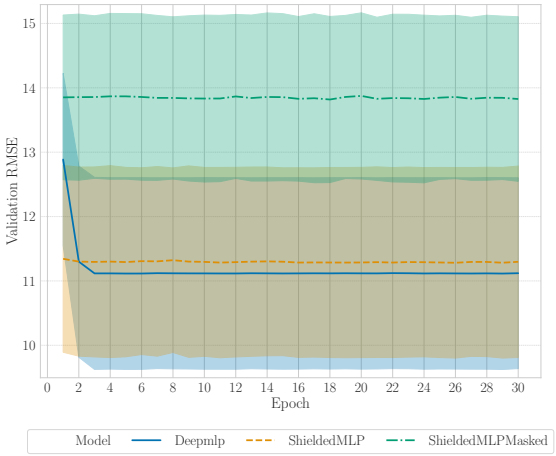
(a) Shallow architecture with projection



(b) Shallow architecture with masked training



(c) Deep architecture with projection



(d) Deep architecture with masked training

Figure 5.2: Training stability on the URL dataset across shallow and deep architectures.

Figure 5.2 presents the most illustrative results in terms of training instability. In both shallow configurations, the RMSE diverges under the masking and projection methods, highlighting the fragility of these methods in low-capacity networks. It is interesting to note that the degree of divergence of the masked method is less than that of the projected method as is visible from the magnitudes of the various RMSEs. In contrast, deep architectures display more stable results, with lower variance and convergence to lower final errors.

The masking method consistently yields worse final RMSE than the projected method, reinforcing the trends observed in the statistical analysis.

For conciseness, training stability plots for the remaining datasets (LCLD and Faulty Steel Plates) are excluded from the main text, as they do not demonstrate any interesting or alarming trends. These results are provided in Appendix C for completeness.

6

Conclusions and Future Work

Contents

This thesis addressed the fundamental challenge of performing effective gradient-based learning in neural networks subject to hard linear constraints. While existing frameworks like the PiShield package guarantee constraint satisfaction in the forward pass, the non-differentiable clamping operations used for enforcement disrupts gradient flow during back-propagation. This can lead to sub-optimal convergence and poor predictive performance compared to a baseline model, as illustrated comprehensively in Chapter 5 The central objective of this work was to design, analyze, and evaluate novel algorithmic techniques to improve the quality of gradient updates.

Two distinct methods were proposed and investigated. The first, a Masked Mean Squared Error (MSE) approach, sought to isolate gradient paths and align updates with the supervised objective. However, extensive evaluation revealed this method to be unstable. The introduction of a potentially negative loss function conflicted with the assumptions of the Adam optimizer, and the stochastic nature of the masking introduced high variance, ultimately leading to performance degradation. While this exploratory path did not yield an optimal solution, it provided a valuable negative result, highlighting the nuanced interaction between custom loss functions and the requirements of modern optimizers.

The second method, based on Projected Gradients combined with a Straight-Through Estimator (STE), proved to be more effective under certain architectural configurations. By projecting

the raw gradient onto the tangent space of the active constraints, this method successfully preserved the original optimization direction while respecting the geometry of the feasible region. Our experiments demonstrated that this technique yields statistically significant improvements in RMSE and achieves near-perfect gradient alignment on deeper architectures displaying a greater degree of dynamical isometry. When applied to shallow architectures, this method also failed, emphasizing the link between its stability and the spectral properties of the network's Jacobian detailed in Chapter 3.

6.1 Principal Contributions

The primary contributions of this thesis are threefold:

1. A detailed analysis and empirical demonstration of how standard shield layers, despite their utility, can impede learning by generating conflicting or zeroed gradients.
2. The novel design and successful implementation of a Projected Gradients method integrated with an STE. We identified the necessary architectural conditions (sufficient depth and normalization) for its success, thereby providing a practical guide for its application.
3. The design and investigation of the masking approach, offering valuable insights into the practical challenges and nuanced considerations involved in introducing extra sources of stochasticity, and designing custom loss functions for constrained optimization.

6.2 Limitations and Future Work

Despite its success, the Projected Gradients method has notable limitations. Its primary drawback is computational complexity; the matrix inversion required for the projection scales cubically with the number of active constraints (3), rendering it impractical for problems with high constraint density. Furthermore, its reliance on specific deep network architectures limits its "plug-and-play" applicability.

These limitations motivate several directions for future research. To address scalability, work could focus on developing heuristics to approximate the projection. For instance, operating on subsets of active constraints, thereby calculating projection less frequently, could be favorable trade-off between computational efficiency and gradient precision.

The logical next step, as introduced earlier in this thesis, would involve extending these methods to more complex constraint landscapes, such as handling the non-convex and disjunctive constraints presented by Giunchiglia et al. [4], and described in detail in Chapters 2, B, potentially by integrating these gradient correction techniques into advanced frameworks like the DRL (2).

7 Reflections

Contents

7.1 Communication

The construction of both the report and presentation prioritized brevity, clarity, and accessibility to effectively communicate complex engineering concepts. The report, with its analytical and mathematical derivations, served to convey the project's technical depth to readers with a strong grasp of optimization and machine learning mathematics, acting as a comprehensive, lasting reference.

The presentation, on the other hand, focused on conveying the project's essence and key contributions to a broader audience. It emphasized high-level intuition and visual clarity over detailed proofs, making it more accessible and engaging for those less familiar with the details.

The effectiveness of both communication methods was significantly improved by continuous feedback from my supervisor. Their general perspective was crucial in identifying areas of the report that, while clear to me, might not be clear to a new reader. This iterative refinement ensured the comprehensibility of the report and the impact of the presentation. Ultimately, the report excelled at providing thorough justification and detailed data, while the presentation was superior for conveying core problems and solutions in an engaging, high-level format.

7.2 Environmental & Social Impact

This project has the potential to create a positive impact on both society and the environment. It introduces methods that improve the convergence of neural networks under linear constraints, particularly in multi-output regression tasks. These methods are relevant in safety-critical settings such as medical diagnostics and autonomous systems, where reliable and rule-abiding model outputs are essential for practical deployment.

From an environmental perspective, the project was carried out with minimal resource consumption. Training was limited to small-scale regression models on publicly available datasets, with a small amount of epochs and seeds. The architectures covered involved lightweight and relatively simple MLP architectures to reduce computational load. All experiments were run on existing GPUs, with no additional equipment used.

The significant computational overhead associated with solving matrix equations represented the sole major processing concern. This challenge was effectively addressed by restricting the number of random seeds, training epochs, and datasets used for evaluation. Moreover, the *gradient alignment* metric (quantified in Chapter 5) proved to converge quickly, thus nullifying the need for extensive training epochs to achieve reliable results.

A

Relevant Repositories

The source code developed for this thesis is publicly available at: `github.com/arnavxkohli/shield-projected-gradient`. The repository includes:

- Implementation of the `ProjGradSTE` function used for projected gradient updates.
- Raw log files and CSV outputs from individual experimental runs.
- Plotting and analysis scripts for reproducibility and figure generation.
- A data extraction script, with links to the processed and raw datasets.
- A `README.md` with detailed instructions about the layout of the repository, and to reproduce the experiments described in Chapter 5.

In addition, a customized fork of the PiShield package was used to develop the masking-based constraint enforcement method.

The implementation of the shield layer builds on the original `ConstrainedDGM` codebase introduced by Stoian et al. [3].

The data files used in all the experiments in Chapter 5 can be found here.

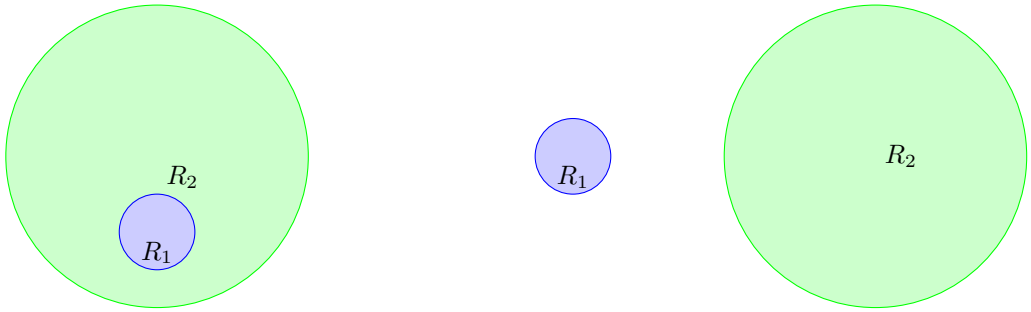
B

Proofs and Examples

This appendix provides the supplementary examples to 2.

B.1 Comparison of f^+ and g^+ Methods

To illustrate alternative methodologies for enforcing hierarchical relationships, we can examine the f^+ and g^+ post-processing methods, as discussed in [2]. These methods adjust confidence scores to reflect class hierarchies, particularly in scenarios where class A is a subclass of class B ($A \subseteq B$).



(a) $R_1 \cap R_2 = R_1$ (A is a subset of B)

(b) $R_1 \cap R_2 = \emptyset$ (A and B are disjoint)

Figure B.1: Comparison of f^+ and g^+ methods. The left figure illustrates an overlapping scenario where f^+ is suitable. The right figure depicts disjoint regions, where g^+ excels by explicitly modeling disjoint subsets.

- f^+ **method:** The f^+ method directly models the hierarchical relationship where class A is a subclass of class B ($A \subseteq B$), implying maximal overlap. It adjusts confidence scores using:

$$f_A^+ = f_A, \quad f_B^+ = \max(f_B, f_A). \tag{B.1}$$

Here, f_A^+ retains the original confidence for the subclass, while f_B^+ is adjusted to ensure that the confidence for the parent class B is never less than its child, A. This approach is computationally efficient and performs well when A is indeed a subset of B, as the natural overlap aligns the predictions. However, f^+ struggles when B includes regions disjoint from A, such as when $B = A \cup (B \setminus A)$ and A and $B \setminus A$ do not overlap. The method's assumption of a continuous relationship can lead to inaccurate classifications in the conceptual 'gap' between A and $B \setminus A$.

- g^+ **method:** The g^+ method explicitly partitions class B into A and $B \setminus A$, where $B \setminus A$ represents the portion of B not overlapping with A. This approach is better suited for disjoint or partially overlapping relationships, ensuring global consistency:

$$g_A^+ = g_A, \quad g_B^+ = \max(g_A, g_{B \setminus A}), \tag{B.2}$$

where $g_{B \setminus A}$ accounts for the confidence in the part of B outside A. By explicitly separating A and $B \setminus A$, g^+ effectively handles more complex hierarchical relationships or cases with disjoint subsets. This explicit representation comes at the cost of increased computational complexity compared to f^+.

In scenarios where $R_1 \cap R_2 \notin \{R_1, \emptyset\}$, the effectiveness of f^+ versus g^+ depends on the specific degree of overlap between the regions. This highlights the need for more generalized solutions capable of handling diverse hierarchical structures robustly.

B.2 `MCLoss` Gradient Analysis

This section provides a detailed mathematical comparison of the `MCLoss` gradient with that of traditional categorical cross-entropy (CE) to illustrate how `MCLoss` overcomes challenges in hierarchical classification, more details can be found in the original paper by Giunchiglia et al. [2]

Consider a scenario with $n+1$ classes where a class A is a subclass of $A_1, A_2, \ldots, A_n$. Suppose the true labels are $y_A = 0$ and $y_{A_1} = y_{A_2} = \cdots = y_{A_n} = 1$. Let the model's predictions be $h_A, h_{A_1}, \ldots, h_{A_n}$.

B.2.1 Categorical Cross-Entropy (CE) Gradient

The standard categorical cross-entropy loss is defined as:

$$L = -\sum_i y_i \ln(\hat{y}_i). \tag{B.3}$$

For our scenario, the gradient for h_A depends on its own error term and the influence of its parent classes $A_1, \ldots, A_n$ through the overall loss function. If h_A is designed to be a parent to multiple classes, the gradient computation for h_A in CE can become problematic:

$$\frac{\partial L}{\partial h_A} = \frac{1}{1 - h_A} - \frac{n}{h_A}. \tag{B.4}$$

This formulation reveals a critical issue: for the model to correctly learn that $y_A = 0$ (i.e., for $\frac{\partial L}{\partial h_A}$ to be negative, driving h_A down), h_A needs to be sufficiently small. However, as the number of parent classes n increases, the term $-\frac{n}{h_A}$ dominates. This leads to a situation where $\frac{\partial L}{\partial h_A} > 0$ (meaning h_A is encouraged to increase) only if $h_A > \frac{n}{n+1}$. For example, if $n = 10$, h_A would need to be greater than $\frac{10}{11} \approx 0.91$ before the gradient changes direction. This makes it extremely difficult for the model to learn to reduce h_A when it is incorrectly high, especially in deep or broad hierarchies.

B.2.2 MCLoss Gradient

In contrast, `MCLoss` explicitly incorporates hierarchical dependencies. For a class $C \in \mathcal{S}$, the $MCLoss_C$ is defined as:

$$MCLoss_C = -y_C \ln(\max_{B \in \mathcal{D}_C}(y_B h_B)) - (1 - y_C) \ln(1 - MCM_C). \tag{B.5}$$

The crucial difference lies in how MCM_C is computed and how the loss is formulated. For h_A in our example ($y_A = 0$), the gradient for h_A in `MCLoss` becomes:

$$\frac{\partial \text{MCLoss}}{\partial h_A} = \frac{1}{1 - h_A}. \tag{B.6}$$

This formulation eliminates the dependency on n (the number of parent classes that A is a subclass of). The gradient for h_A is now consistently $\frac{1}{1-h_A}$, which is always positive if $h_A < 1$, ensuring that if $y_A = 0$ and h_A is high, the gradient will always push h_A downwards. This provides a

consistent and effective gradient for adjusting h_A, regardless of the complexity or depth of the hierarchy. The model can thus effectively reduce hierarchical violations without being constrained by the increasing number of related classes.

B.3 Example of Reduction Operator red_j

This section provides a detailed numerical example of the reduction operator $\text{red}_j(\phi^1, \phi^2)$ for eliminating a variable from linear inequality constraints, as discussed in Section 2.2.1.

Consider the following set of linear inequality constraints:

$$\phi^1 : x_1 - x_2 + x_3 > 0, \tag{B.7}$$

$$\phi^2 : x_2 - 5x_3 > 5, \tag{B.8}$$

$$\phi^3 : 2x_1 - x_3 > 3. \tag{B.9}$$

Define the initial set of constraints for the last variable in our chosen order (x_3 in this case, implying $D = 3$ and $\lambda(x_3) = 3$):

$$\Pi_3 = \{\phi^1, \phi^2, \phi^3\}. \tag{B.10}$$

First, we apply the reduction operator $\text{red}_2(\phi^1, \phi^2)$ to eliminate x_2. We identify $\Pi_3^- = \{\phi^1\}$ (where x_2 has a negative coefficient) and $\Pi_3^+ = \{\phi^2\}$ (where x_2 has a positive coefficient).

For ϕ^1: $w_1^1 = 1, w_2^1 = -1, w_3^1 = 1, b^1 = 0$. For ϕ^2: $w_1^2 = 0, w_2^2 = 1, w_3^2 = -5, b^2 = -5$.

Using Equation 2.7, the coefficients for x_k ($k \neq 2$) in the reduced constraint are:

- Coefficient of $x_1 : w_1^1|w_2^2| + w_1^2|w_2^1| = (1)(1) + (0)(1) = 1$.
- Coefficient of $x_3 : w_3^1|w_2^2| + w_3^2|w_2^1| = (1)(1) + (-5)(1) = -4$.

The constant term is:

- $b^1|w_2^2| + b^2|w_2^1| = (0)(1) + (-5)(1) = -5$.

Thus, the reduced constraint from ϕ^1 and ϕ^2 is:

$$x_1 - 4x_3 > -5. \tag{B.11}$$

After eliminating x_2, the updated set of constraints, Π_2 (for variable x_2), becomes:

$$\Pi_2 = \{x_1 - 4x_3 > -5, \phi^3 : 2x_1 - x_3 > 3\}. \tag{B.12}$$

Next, we apply the reduction operator red_3 to eliminate x_3 from the relevant constraints in Π_2. We consider the newly formed constraint $x_1 - 4x_3 > -5$ (let's call it ϕ^A) and $\phi^3 : 2x_1 - x_3 > 3$. Here, x_3 appears negatively in both.

For ϕ^A: $w_1^A = 1, w_3^A = -4, b^A = -5$. For ϕ^3: $w_1^3 = 2, w_3^3 = -1, b^3 = -3$.

The coefficient of x_1 in the final reduced constraint is:

- $w_1^A|w_3^3| + w_1^3|w_3^A| = (1)(1) + (2)(4) = 9$.

The constant term is:

- $b^A|w_3^3| + b^3|w_3^A| = (-5)(1) + (-3)(4) = -5 - 12 = -17$.

Thus, the final reduced constraint for Π_1 (for variable x_1) is:

$$9x_1 > -17 \implies x_1 > -\frac{17}{9}. \tag{B.13}$$

So, the final set of constraints is:

$$\Pi_1 = \{x_1 > -\frac{17}{9}\}. \tag{B.14}$$

The constraints are reduced sequentially following the variable order:

$$\Pi_3 = \Pi, \quad \Pi_2 = \text{Reduction after eliminating } x_2, \quad \Pi_1 = \text{Reduction after eliminating } x_3. \tag{B.15}$$

B.4 Numerical Example of Constraint Layer Adjustment

This section provides a numerical example of how the Constraint Layer (CL) adjusts generated feature values to satisfy linear inequality constraints, using the reduced constraint sets derived in Appendix B.3.

Let the initial generated values be $\tilde{x}_1 = 2$, $\tilde{x}_2 = 3$, and $\tilde{x}_3 = 4$. We will apply the CL functions iteratively using the derived constraint sets:

- $\Pi_1 = \{x_1 > -\frac{17}{9}\}$
- $\Pi_2 = \{x_1 - 4x_3 > -5,\ 2x_1 - x_3 > 3\}$
- $\Pi_3 = \{x_1 - x_2 + x_3 > 0,\ x_2 - 5x_3 > 5\}$

1. **Adjusting x_1 using Π_1**: For $\Pi_1 = \{x_1 > -\frac{17}{9}\}$, the bounds for x_1 are:

$$lb_1 = -\frac{17}{9} \approx -1.89, \quad ub_1 = \infty. \tag{B.16}$$

Since $\tilde{x}_1 = 2 > lb_1$, the value is within bounds:

$$\text{CL}(\tilde{x}_1) = \min\left(\max(2, -1.89), \infty\right) = 2. \tag{B.17}$$

We denote the adjusted value as $x_1^* = 2$.

2. **Adjusting x_3 using Π_2**: For $\Pi_2 = \{x_1 - 4x_3 > -5,\ 2x_1 - x_3 > 3\}$, we substitute the adjusted $x_1^* = 2$. The constraints become:

- $2 - 4x_3 > -5 \implies -4x_3 > -7 \implies x_3 < \frac{7}{4}$
- $2(2) - x_3 > 3 \implies 4 - x_3 > 3 \implies -x_3 > -1 \implies x_3 < 1$

The bounds for x_3 are:

$$lb_3 = -\infty, \quad ub_3 = \min\left(\frac{7}{4}, 1\right) = 1. \tag{B.18}$$

Since $\tilde{x}_3 = 4$ and $ub_3 = 1$:

$$\text{CL}(\tilde{x}_3) = \min\left(\max(4, -\infty), 1\right) = 1. \tag{B.19}$$

We denote the adjusted value as $x_3^* = 1$.

3. **Adjusting x_2 using Π_3**: For $\Pi_3 = \{x_1 - x_2 + x_3 > 0,\ x_2 - 5x_3 > 5\}$, we substitute the adjusted $x_1^* = 2$ and $x_3^* = 1$. The constraints become:

- $2 - x_2 + 1 > 0 \implies 3 - x_2 > 0 \implies -x_2 > -3 \implies x_2 < 3$
- $x_2 - 5(1) > 5 \implies x_2 - 5 > 5 \implies x_2 > 10$

The bounds for x_2 are:

$$lb_2 = 10, \quad ub_2 = 3. \tag{B.20}$$

In this scenario, $lb_2 > ub_2$, which means the original set of constraints is **infeasible** for the given values of x_1^* and x_3^*, or perhaps the original values. When $lb_i > ub_i$, it indicates that there is no value of x_i that can satisfy all remaining constraints simultaneously. In a practical CL implementation, this condition implies a violation of the feasible region, and the CL would need to manage this, potentially by returning a value that minimizes constraint violation or by signaling infeasibility. For the sake of this example, we proceed with the clipping logic, acknowledging the infeasibility:

$$\mathrm{CL}(\tilde{x}_2) = \min\left(\max(3, 10), 3\right) = \min(10, 3) = 3. \tag{B.21}$$

We denote the adjusted value as $x_2^* = 3$.

After processing all constraints, the final adjusted values based on this example are:

$$\mathrm{CL}(\tilde{x}_1) = 2, \quad \mathrm{CL}(\tilde{x}_2) = 3, \quad \mathrm{CL}(\tilde{x}_3) = 1. \tag{B.22}$$

Since the given inequalities are strict ($>$), the solutions found need to be adjusted by adding a small error margin, ϵ (the smallest representable float, which is architecture-dependent). For example, x_3 would be $1 + \epsilon$. As stated by *Theorem 3.6* in [3], this adjustment yields a solution as close to an optimal as possible for $\mathrm{CL}(\tilde{x}_i)$.

B.5 Single Variable Case

In this setting, each constraint $\Psi \in \Pi$ defines a left boundary l_i^Ψ and a right boundary r_i^Ψ for the variable x_i. These boundaries are derived from the linear inequalities that constitute Ψ, ensuring the constraints are satisfied. For a constraint $\Psi = \bigvee_{j=1}^{n_\Psi}(w_j x_i + b_j \geq 0)$, the boundaries are determined as follows:

$$l_i^\Psi = \max_{(w_i x_i + b \geq 0) \in \Psi : w_i < 0} \left(\frac{-b}{w_i}\right), \quad r_i^\Psi = \min_{(w_i x_i + b \geq 0) \in \Psi : w_i > 0} \left(\frac{-b}{w_i}\right). \tag{B.23}$$

[4]

Here, w_j and b_j are the coefficients and constants associated with the individual linear inequalities in Ψ. A prediction $\tilde{x}_i$ satisfies the constraint Ψ if it lies within the allowable interval defined by these boundaries: $x_i \in [l_i^\Psi, r_i^\Psi]$. If no valid $w_j < 0$ or $w_j > 0$ exists, the respective boundary l_i^Ψ or r_i^Ψ is unbounded ($-\infty$ or $+\infty$).

For a set of constraints $\Pi = \{\Psi_j : 1 \leq j \leq Z\}$, the left and right boundaries for x_i are obtained by aggregating the individual boundaries from each $\Psi_j \in \Pi$. Specifically:

$$l_i^{\Pi} = \max\{l_i^{\Psi} : \Psi \in \Pi\}, \quad r_i^{\Pi} = \min\{r_i^{\Psi} : \Psi \in \Pi\}. \tag{B.24}$$

[4]

To refine a prediction $\tilde{x}_i$ that violates the constraints, the DRL adjusts x_i to the nearest boundary of the feasible region defined by $[l_i^{\Pi}, r_i^{\Pi}]$. The boundaries $l_i^{\Pi}(\tilde{x})$ and $r_i^{\Pi}(\tilde{x})$ are computed by identifying the closest satisfying left and right boundaries within the set of constraints Π, as follows:

$$l_i^{\Pi}(\tilde{x}) = \max_{\Psi \in \Pi} \left(\{ l_i^{\Psi} : \tilde{x}_i > l_i^{\Psi}, l_i^{\Psi} \in \Omega(\Pi) \} \right), \quad r_i^{\Pi}(\tilde{x}) = \min_{\Psi \in \Pi} \left(\{ r_i^{\Psi} : \tilde{x}_i < r_i^{\Psi}, r_i^{\Psi} \in \Omega(\Pi) \} \right). \tag{B.25}$$

[4]

Here, l_i^{Ψ} and r_i^{Ψ} represent the left and right boundaries defined by individual constraints $\Psi \in \Pi$, and $\Omega(\Pi)$ denotes the overall feasible region defined by all constraints.

Once $l_i^{\Pi}(\tilde{x})$ and $r_i^{\Pi}(\tilde{x})$ are determined, the DRL adjusts $\tilde{x}_i$ as follows:

$$DRL(\tilde{x})_i = \begin{cases} \tilde{x}_i, & \text{if } \tilde{x} \in \Omega(\Pi), \\ l_i^{\Pi}(\tilde{x}), & \text{if } \tilde{x} \notin \Omega(\Pi) \text{ and } |\tilde{x}_i - l_i^{\Pi}(\tilde{x})| < |\tilde{x}_i - r_i^{\Pi}(\tilde{x})|, \\ r_i^{\Pi}(\tilde{x}), & \text{otherwise.} \end{cases} \tag{B.26}$$

[4]

This adjustment ensures that the modified prediction satisfies the constraints in Π while minimizing its deviation from the original value $\tilde{x}_i$. In cases where $\Omega(\Pi)$ consists of multiple disjoint intervals, the refinement operation ensures that the nearest boundary is chosen on the basis of the Euclidean distance.

Consider a variable x constrained by the following set of inequalities:

$$(x - 2 \geq 0 \wedge 5 - x \geq 0) \vee (x - 8 \geq 0 \wedge 10 - x \geq 0). \tag{B.27}$$

Breaking it down:

- The first inequality set, $x - 2 \geq 0 \wedge 5 - x \geq 0$, corresponds to the interval $[2, 5]$.
- The second inequality set, $x - 8 \geq 0 \wedge 10 - x \geq 0$, corresponds to the interval $[8, 10]$.

This can be converted into two disjoint feasible intervals:

$$x \in [2, 5] \cup [8, 10]. \tag{B.28}$$

The feasible regions for the variable x are defined as the intervals $[2, 5]$ and $[8, 10]$. Given a predicted value $\tilde{x} = 6.5$, the closest left boundary is $l^{\Pi}(x) = 5$, and the closest right boundary is $r^{\Pi}(x) = 8$. The distances to these boundaries are calculated as

$$|\tilde{x} - l^{\Pi}(x)| = |6.5 - 5| = 1.5, \quad |\tilde{x} - r^{\Pi}(x)| = |6.5 - 8| = 1.5. \tag{B.29}$$

Since the distances are equal, the convention is to select the right boundary, $r^{\Pi}(x) = 8$. Therefore, DRL modifies the predicted value $\tilde{x}$ to $x = 8$, ensuring the adjusted value satisfies the constraints and lies within the feasible region.

B.6 General Case: The Cutting Planes Rule and Variable Elimination

In the Disjunctive Refinement Layer (DRL), the entire refinement process is designed to occur in a single forward pass. To achieve this, a specific ordering of the variables, $x_1, x_2, \ldots, x_D$, is considered. This ordering determines the sequence in which the values of the variables are refined. Importantly, this order can either be arbitrarily chosen or selected based on user preferences, particularly when certain variables are considered more critical or should be adjusted first. [4]

When refining the value of a variable x_i, the values of all preceding variables, $x_1, \ldots, x_{i-1}$, are treated as immutable for that step. This ensures that the refinement process for x_i respects the adjustments already made to earlier variables. For x_i to be computed, it must satisfy:

- The constraints involving x_i, given the fixed values of $x_1, \ldots, x_{i-1}$.
- The possibility of extending this solution to satisfy all remaining constraints for subsequent variables $x_{i+1}, \ldots, x_D$.

The refinement process starts with the full set of constraints $\Pi = \Pi_D$ (where $i = D$, the last variable) and works backward. For each variable x_i, the goal is to derive a subset of constraints, denoted as Π_{i-1}, that involves only the variables $x_1, \ldots, x_{i-1}$. The key idea is to ensure that any valid assignment of $x_1, \ldots, x_{i-1}$ can be extended to find a valid value for x_i satisfying the remaining constraints.

Mathematically, this amounts to ensuring $\exists x_i \left(\bigwedge_{\Psi \in \Pi_i} \Psi \right)$. [4] In simpler terms, this means that for any assignment of $x_1, \ldots, x_{i-1}$, there must exist a value of x_i such that all constraints in Π_i are satisfied.

Let constraints Ψ and Ψ' be defined as:

$$\Psi' = \bigvee_{j=1}^{m} (w'_j x_i + \varphi'_j \geq 0) \vee \Phi', \quad \Psi = \bigvee_{k=1}^{n} (w_k x_i + \varphi_k \geq 0) \vee \Phi \tag{B.30}$$

Here:

- m and n are the number of disjunctions (logical ORs) in Ψ and Ψ', respectively.
- w'_j, w_k are coefficients for x_i, and φ'_j, φ_k are offsets.
- Φ and Φ' are additional constraints that do not involve x_i.

The *Cutting Planes* (CP) rule is a refinement technique used to eliminate variables from a system of constraints, ensuring that the resulting system satisfies all the original constraints while simplifying their structure. It operates by combining existing constraints to derive new ones, progressively reducing the dimensionality of the problem. This process ensures that every solution to the refined system corresponds to a solution of the original system, thus preserving logical equivalence. By systematically applying the CP rule, the resulting constraints maintain the relationships among variables while becoming easier to solve or verify. This methodology is particularly useful in scenarios where the constraints involve disjunctions, non-convex relationships, or other complex interactions, such as those encountered in generative models like the Disjunctive Refinement Layer (DRL). The goal is to iteratively resolve pairs of constraints, eliminate variables, and ensure that the refined system adheres to the background knowledge encoded in the original constraints.

Given constraints Ψ and Ψ' involving a variable x_i, defined above, the CP rule derives a new set of constraints that no longer depend on x_i. The conclusion of the CP rule, denoted $\text{CPres}_{x_i}(\varphi, \varphi')$,

is:

$$\text{CPres}_{x_i}(\varphi, \varphi') = \bigvee_{j=1}^{m} \bigvee_{k=1}^{n} \left(\frac{\varphi_k}{w_k} - \frac{\varphi'_j}{w'_j} \geq 0 \right) \vee \Phi \vee \Phi', \tag{B.31}$$

where w_k and w'_j are the coefficients of x_i in the constraints, φ_k and φ'_j are offsets, and Φ and Φ' represent additional constraints that do not involve x_i. This equation encapsulates the process of resolving bounds on x_i and combining them with the remaining constraints. The resulting system eliminates x_i while maintaining the relationships defined by the original constraints.

Using values from *Example 3* ([4]) to illustrate this method, it is possible to reduce the constraints $\Pi = \{\Psi_1, \Psi_2, \Psi_3\}$ where:

- $\Psi_1 = x_5 \geq x_1$
- $\Psi_2 = (x_5 \leq x_2) \bigvee (x_5 \geq x_3)$
- $\Psi_3 = x_5 \leq x_4$

such that the dependence on x_5 is removed. As stated before, this is represented as $\exists x_5 \bigwedge_{\Psi \in \Pi} \Psi$.

To resolve Ψ_1 and Ψ_2, consider that Ψ_1 can be written as $x_5 \geq x_1$, which is equivalent to $w_1 x_5 + \varphi_1 \geq 0$, where $w_1 = 1$ and $\varphi_1 = -x_1$. Ψ_2 is a disjunction over $(x_5 \leq x_2) \vee (x_5 \geq x_3)$, where the first part corresponds to $w_2 x_5 + \varphi_2 \leq 0$ with $w_2 = -1$ and $\varphi_2 = x_2$, and the second part corresponds to $w_3 x_5 + \varphi_3 \geq 0$ with $w_3 = 1$ and $\varphi_3 = -x_3$.

The resolution of Ψ_1 with the first part of Ψ_2, $(x_5 \leq x_2)$, involves computing the CP between these constraints. The relationship between their coefficients and constants gives:

$$\frac{\varphi_2}{w_2} - \frac{\varphi_1}{w_1} = \frac{x_2}{-1} - \frac{-x_1}{1} = -x_2 + x_1. \tag{B.32}$$

This yields $x_1 \leq x_2$.

Similarly, resolving Ψ_1 with the second part of Ψ_2, $(x_5 \geq x_3)$, results in:

$$\frac{\varphi_3}{w_3} - \frac{\varphi_1}{w_1} = \frac{-x_3}{1} - \frac{-x_1}{1} = -x_3 + x_1. \tag{B.33}$$

This gives $x_1 \leq x_3$.

Combining these results:

$$CPres_5(\Psi_1, \Psi_2) = (x_1 \leq x_2) \vee (x_5 \geq x_3). \tag{B.34}$$

Note that x_5 has not been completely eliminated yet.

The next step involves resolving Ψ_1 and Ψ_3. Ψ_3 is expressed as $x_5 \leq x_4$, or $w_4 x_5 + \varphi_4 \leq 0$, where $w_4 = -1$ and $\varphi_4 = x_4$. Resolving Ψ_1 and Ψ_3 produces the cutting plane:

$$\frac{\varphi_4}{w_4} - \frac{\varphi_1}{w_1} = \frac{x_4}{-1} - \frac{-x_1}{1} = -x_4 + x_1. \tag{B.35}$$

This gives:

$$CPres_5(\Psi_1, \Psi_3) = x_1 \leq x_4. \tag{B.36}$$

Finally, the result from the previous resolution of Ψ_1 and Ψ_2, $(x_1 \leq x_2) \vee (x_5 \geq x_3)$, is combined with Ψ_3 to eliminate x_5. Resolving these constraints yields:

$$(x_1 \leq x_2) \vee (x_3 \leq x_4). \tag{B.37}$$

The final reduced set of constraints, after eliminating x_5, consists of:

$$\{x_1 \leq x_4,\, (x_1 \leq x_2) \vee (x_3 \leq x_4)\} \tag{B.38}$$

Which can be re-written as the disjunction $(x_1 \leq x_4) \wedge ((x_1 \leq x_2) \vee (x_3 \leq x_4))$. This derived set of constraints needs to be satisfied among the features involved for a solution involving x_5 to be admitted.

When attempting to eliminate a variable x_i from a set of constraints, direct resolution often generates intermediate constraints that involve both positive and negative occurrences of x_i. These new constraints can themselves act as premises for further resolutions, creating a cascading effect that significantly increases computational complexity. This issue is addressed through:

- Partitioning the constraints into subsets based on how x_i appears (positively, negatively, or both), reducing redundancy in resolutions.
- Using recursive application of the CP resolution rule to focus on resolving specific subsets of constraints, thereby simplifying the process while preserving logical equivalence.
- Constructing the reduced set of constraints in a way that eliminates x_i without reintroducing unnecessary complexity.

The constraints are divided into three subsets:

- Π_i^+: Constraints where x_i appears positively but not negatively.
- Π_i^-: Constraints where x_i appears negatively but not positively.
- $\Pi_i^\pm$: Constraints where x_i appears both positively and negatively (flipping the sign to $\geq$ and using .

To efficiently handle resolutions involving constraints with x_i, a new set of constraints, $\Pi_i^\star$, is introduced. This set is defined by recursively applying the CP resolution rule between constraints from Π_i^+ and $\Pi_i^\pm$, producing only constraints with positive occurrences of x_i. Formally, the recursion is defined as:

$$\Pi_i^\star = \bigcup_{k=0}^{|\Pi_i^\pm|} \Pi_i^k, \tag{B.39}$$

where:

$$\Pi_i^0 = \Pi_i^+, \quad \Pi_i^{k+1} = \{\text{CPres}_i(\Psi, \Psi') \mid \Psi \in \Pi_i^k,\, \Psi' \in \Pi_i^\pm\}. \tag{B.40}$$

This process ensures that $\Pi_i^\star$ contains only positive occurrences of x_i and captures all necessary information from Π_i^+ and $\Pi_i^\pm$. Once $\Pi_i^\star$ is constructed, the constraints in Π_i^- are resolved with $\Pi_i^\star$ to produce the final reduced set Π_{i-1}, where x_i is eliminated. The reduced set is given by:

$$\Pi_{i-1} = \big(\Pi_i \setminus (\Pi_i^+ \cup \Pi_i^- \cup \Pi_i^\pm)\big) \cup \{\text{CPres}_i(\Psi, \Psi') \mid \Psi \in \Pi_i^\star,\, \Psi' \in \Pi_i^-\}. \tag{B.41}$$

Continuing from the example, $\Pi_5 = \{\Psi_1, \Psi_2, \Psi_3\}$, where:

$$\Psi_1 = x_5 \geq x_1, \quad \Psi_2 = (x_5 \leq x_2) \vee (x_5 \geq x_3), \quad \Psi_3 = x_5 \leq x_4 \tag{B.42}$$

The constraints are partitioned as follows:

- $\Pi_5^+ = \{\Psi_1 = x_5 \geq x_1\}$: x_5 appears positively but not negatively.
- $\Pi_5^- = \{\Psi_3 = x_5 \leq x_4\}$: x_5 appears negatively but not positively.
- $\Pi_5^\pm = \{\Psi_2 = (x_5 \leq x_2) \vee (x_5 \geq x_3)\}$: x_5 appears both positively and negatively.

The Cutting Planes (CP) resolution rule is applied to resolve these constraints. Resolving Π_5^+ with $\Pi_5^\pm$ (Ψ_1 with Ψ_2) yields:

$$(x_1 \leq x_2) \vee (x_5 \geq x_3) \tag{B.43}$$

which forms part of Π_5^{++}. Next, resolving $(x_1 \leq x_2) \vee (x_5 \geq x_3)$ with Π_5^{-} (Ψ_3) produces:

$$(x_1 \leq x_2) \vee (x_3 \leq x_4) \tag{B.44}$$

Additionally, resolving Π_5^{+} (Ψ_1) directly with Π_5^{-} (Ψ_3) gives:

$$x_1 \leq x_4 \tag{B.45}$$

Finally, the reduced set of constraints, Π_4, is constructed by combining:

- Constraints from $\Pi_5 \setminus (\Pi_5^{+} \cup \Pi_5^{-} \cup \Pi_5^{\pm})$, which in this case is empty.
- Resolutions between Π_5^{++} and Π_5^{-}. Resolving $(x_1 \leq x_2) \vee (x_5 \geq x_3)$ with $\Psi_3 = x_5 \leq x_4$ produces $(x_1 \leq x_2) \vee (x_3 \leq x_4)$

Finally, with x_5 eliminated:

$$\Pi_4 = \{x_1 \leq x_4, (x_1 \leq x_2) \vee (x_3 \leq x_4)\} \tag{B.46}$$

which is the same result as before, but produced in a systematic, repeatable manner.

Using this result, the paper describes how CP simplifies multi-variable systems into manageable single-variable cases by iteratively isolating variables and resolving their relationships within the constraints. Starting with the first variable x_1, and proceeding sequentially to $x_2, x_3, \ldots, x_n$, each variable is assigned a value based on the boundaries derived from its associated constraints.

Similar to the single variable case, for a variable x_i, the constraints in Π_i are resolved to define:

$$l_i^{\Pi_i}(\tilde{x}_i) = \max_{\Psi \in \Pi_i} \{l_i^{\Psi} : \tilde{x}_i > l_i^{\Psi},\, l_i^{\Psi} \in \Omega(\Pi_i)\}, \quad r_i^{\Pi_i}(\tilde{x}_i) = \min_{\Psi \in \Pi_i} \{r_i^{\Psi} : \tilde{x}_i < r_i^{\Psi},\, r_i^{\Psi} \in \Omega(\Pi_i)\} \tag{B.47}$$

The only difference being that Π_i is considered instead of Π (the whole set of constraints).

The adjustment process also follows a similar method to the single variable case:

$$\mathrm{DRL}(\tilde{x}_i) = \begin{cases} \tilde{x}_i, & \text{if } \tilde{x}_i \in \Omega(\Pi_i), \\ l_i^{\Pi_i}(\tilde{x}_i), & \text{if } \tilde{x}_i \notin \Omega(\Pi_i) \text{ and } |\tilde{x}_i - l_i^{\Pi_i}(\tilde{x}_i)| < |\tilde{x}_i - r_i^{\Pi_i}(\tilde{x}_i)|, \\ r_i^{\Pi_i}(\tilde{x}_i), & \text{otherwise.} \end{cases} \tag{B.48}$$

Once the value of x_i is determined, it is propagated forward to resolve the constraints for subsequent variables $x_{i+1}, x_{i+2}, \ldots$. For variables x_j with $j > i$, their values are updated directly based on the corrected value of $\tilde{x}_i$, maintaining consistency across all constraints. The process sequentially adjusts variables until all are assigned values within their feasible ranges.

C Figures

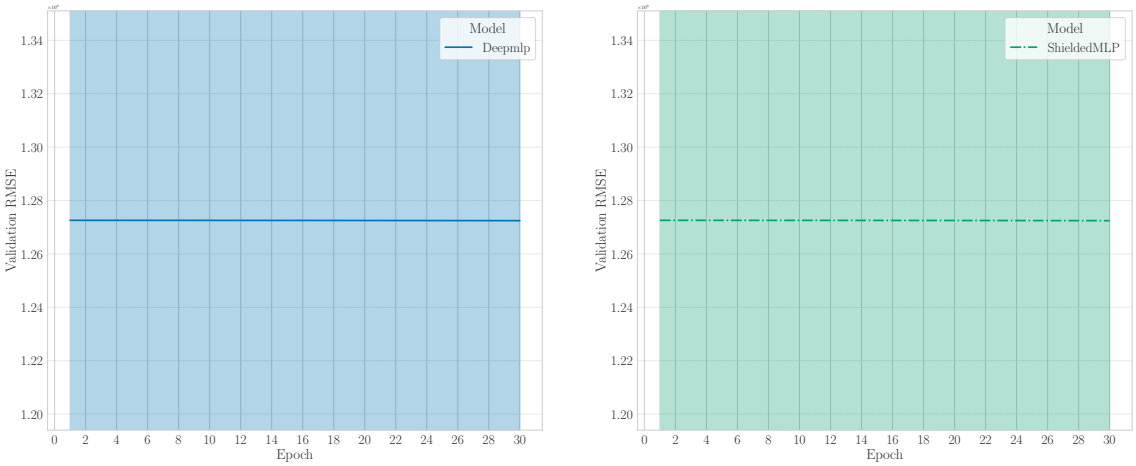
This appendix provides supplementary plots to the explanation provided in Chapter 5. Each section rovides the plots for the RMSE values obtained on the corresponding dataset, for the projected gradient method, as well as the mask case. It presents these as separate plots, as well as a combined plot for comparison.

C.1 Faulty Steel Plates

C.1.1 Projected Gradient Plots

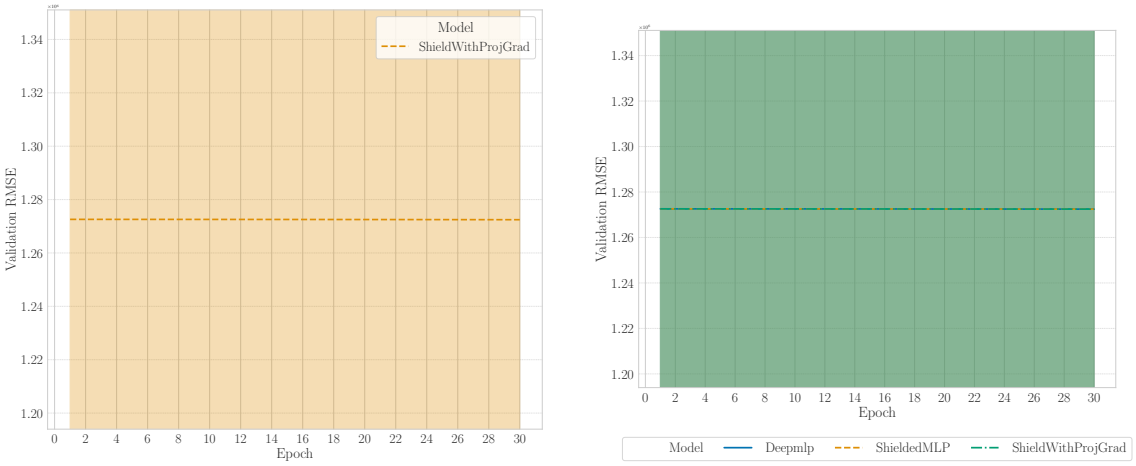
The high variance in the RMSE results for the Faulty Steel Plates as shown in Figures C.5, C.2b, C.7 and C.4b datasets is due to the high magnitude of the target values being regressed over. Scaling these down would break the constraint satisfaction guarantees in the original target dataset. From the combined plot, it is clear that this is not a very good representation of the performance of the method, as all of the RMSE values are quite close to each other.

Deep MLP



(a) Deep MLP (b) Deep Shielded MLP

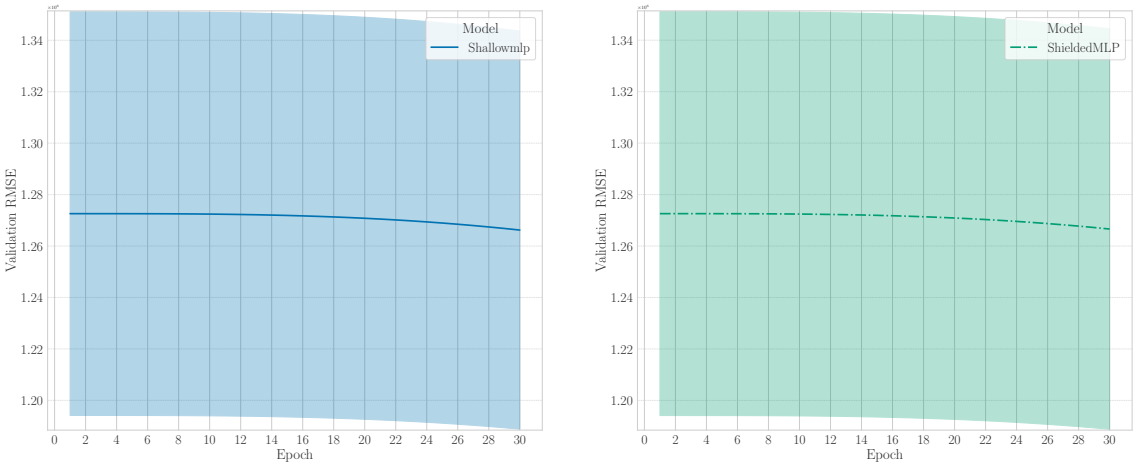
Figure C.1: Performance comparison between the Deep MLP and Deep Shielded MLP models on the Faulty Steel Plates dataset.



(a) Deep Shielded MLP with Projected Gradient (b) Combined Deep Unmasked Models

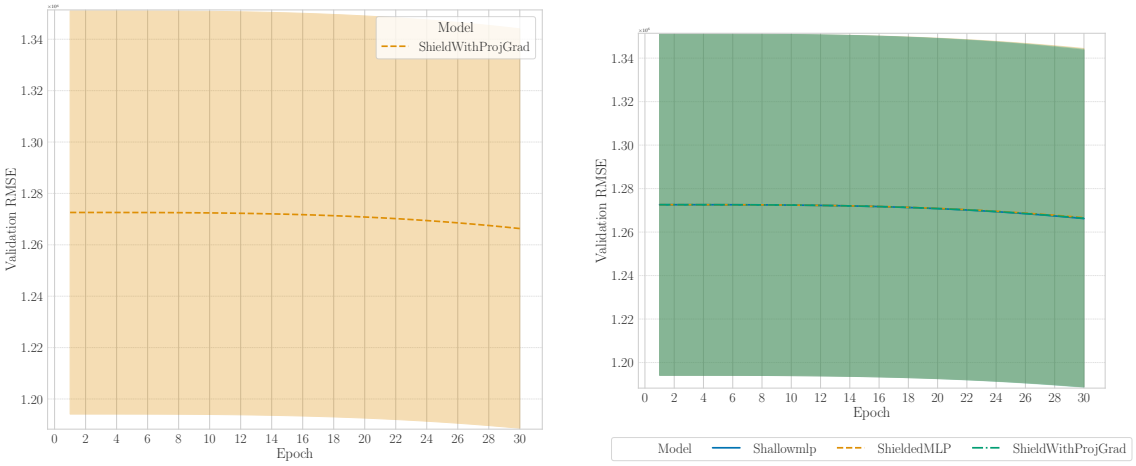
Figure C.2: Performance of the Deep Shielded MLP with Projected Gradient and the combined view of all Deep Unmasked Models on the Faulty Steel Plates dataset.

Shallow MLP



(a) Shallow MLP (b) shallow Shielded MLP

Figure C.3: Performance comparison between the Shallow MLP and Shallow Shielded MLP models on the Faulty Steel Plates dataset.



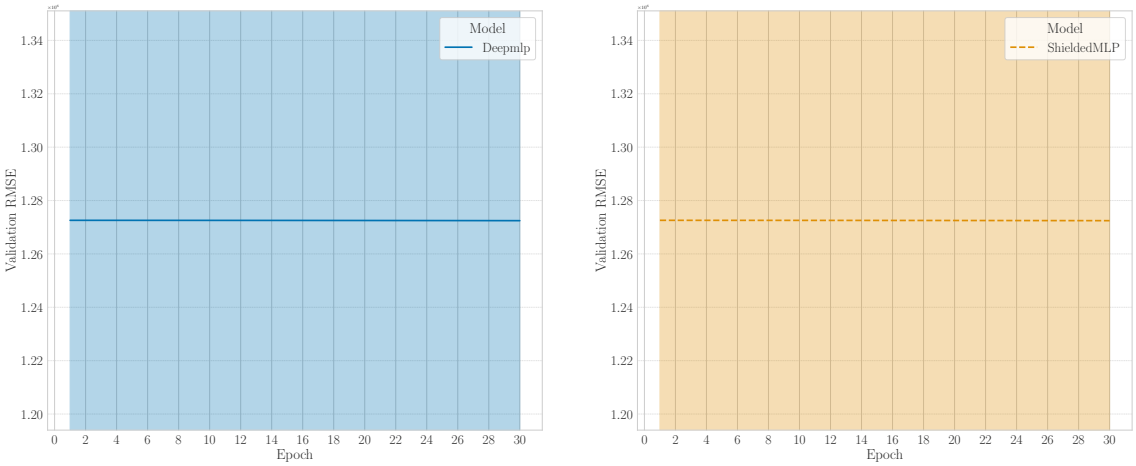
(a) Shallow Shielded MLP with Projected Gradient (b) Combined Shallow Unmasked Models

Figure C.4: Performance of the Shallow Shielded MLP with Projected Gradient and the combined view of all Shallow Unmasked Models on the Faulty Steel Plates dataset.

C.1.2 Masked Plots

Similar to the projected gradients case, making conclusions around stability through the below plots is difficult because of the scaling issues.

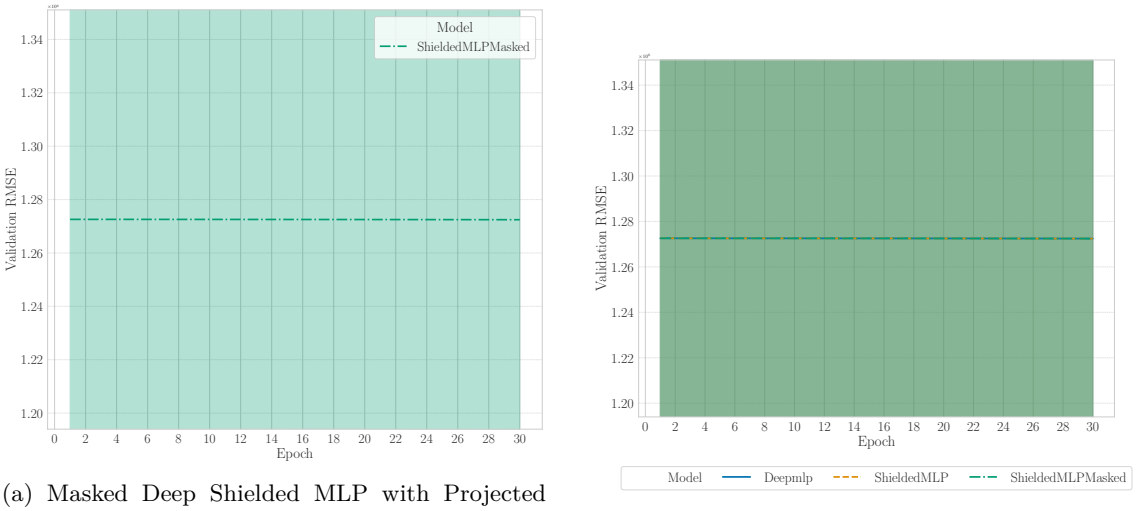
Deep MLP



(a) Masked Deep MLP

(b) Masked Deep Shielded MLP

Figure C.5: Performance comparison between the Masked Deep MLP and Masked Deep Shielded MLP models on the Faulty Steel Plates dataset.

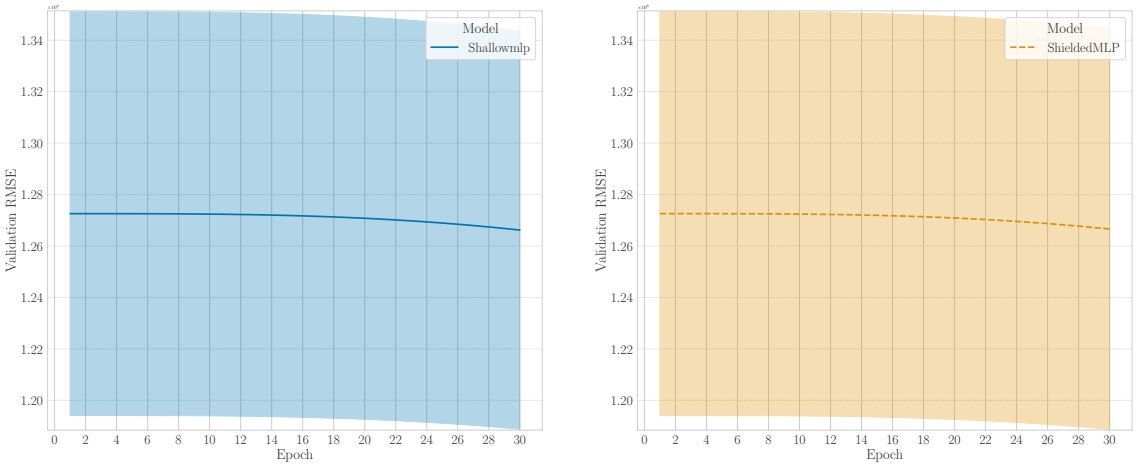


(a) Masked Deep Shielded MLP with Projected Gradient

(b) Combined Masked Deep Models

Figure C.6: Performance of the Masked Deep Shielded MLP with Projected Gradient and the combined view of all Masked Deep Models on the Faulty Steel Plates dataset.

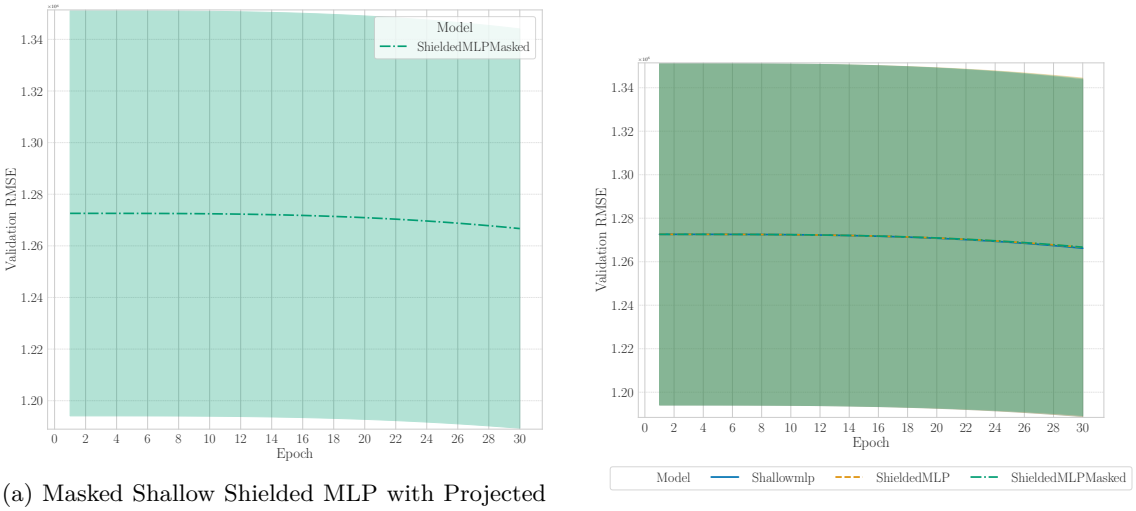
Shallow MLP



(a) Masked Shallow MLP

(b) Masked Shallow Shielded MLP

Figure C.7: Performance comparison between the Masked Shallow MLP and Masked Shallow Shielded MLP models on the Faulty Steel Plates dataset.



(a) Masked Shallow Shielded MLP with Projected Gradient

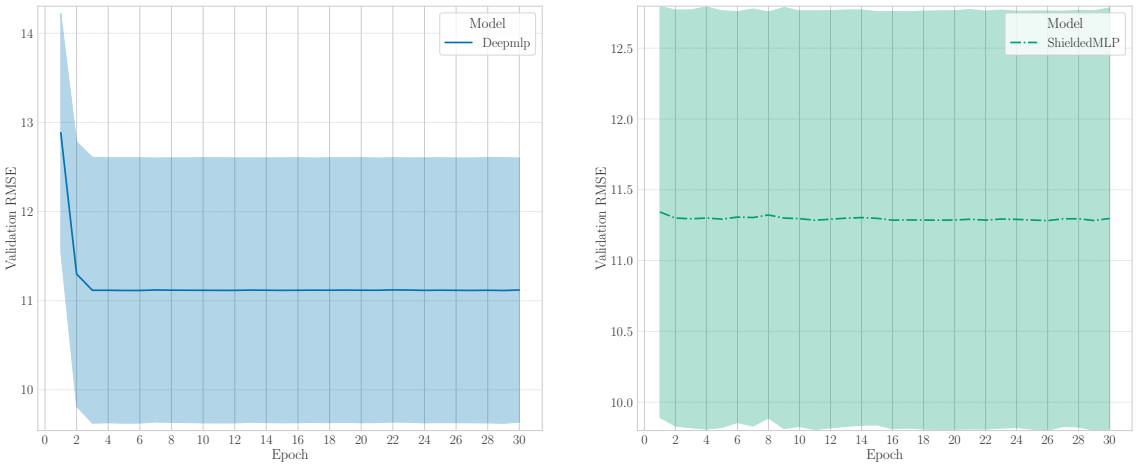
(b) Combined Masked Shallow Models

Figure C.8: Performance of the Masked Shallow Shielded MLP with Projected Gradient and the combined view of all Masked Shallow Models on the Faulty Steel Plates dataset.

C.2 URL

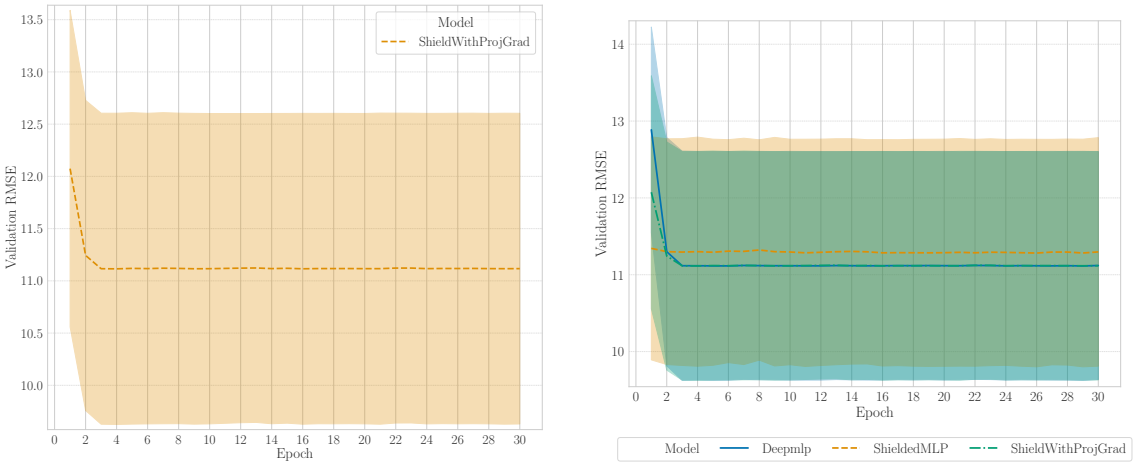
C.2.1 Projected Gradient Plots

Deep MLP



(a) Deep MLP (b) Deep Shielded MLP

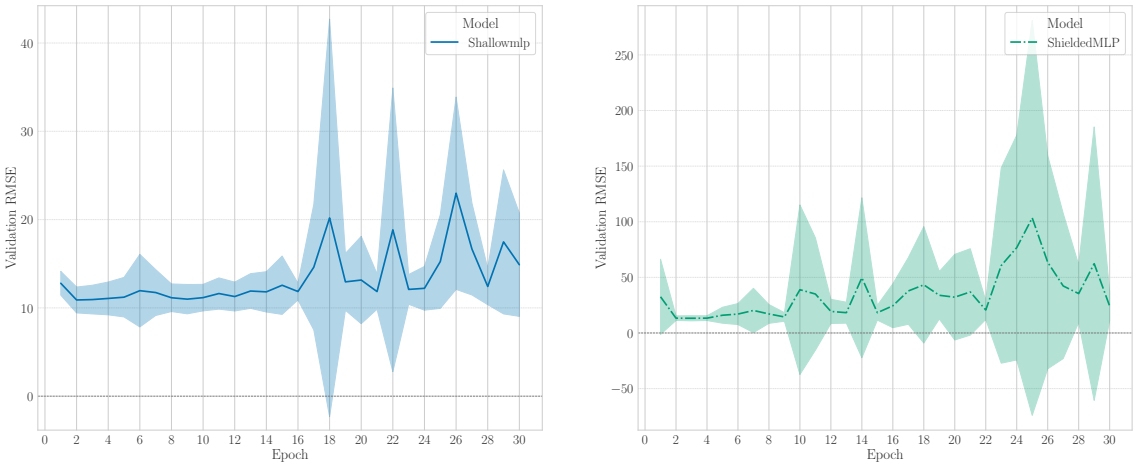
Figure C.9: Performance comparison between the Deep MLP and Deep Shielded MLP models on the URL dataset.



(a) Deep Shielded MLP with Projected Gradient (b) Combined Deep Unmasked Models

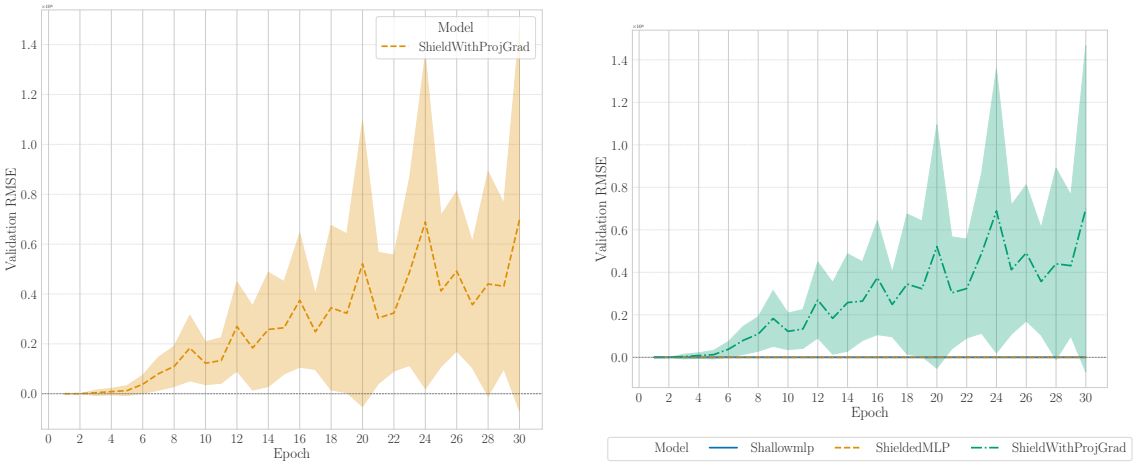
Figure C.10: Performance of the Deep Shielded MLP with Projected Gradient and the combined view of all Deep Unmasked Models on the URL dataset.

Shallow MLP



(a) shallow MLP (b) shallow Shielded MLP

Figure C.11: Performance comparison between the shallow MLP and shallow Shielded MLP models on the URL dataset.

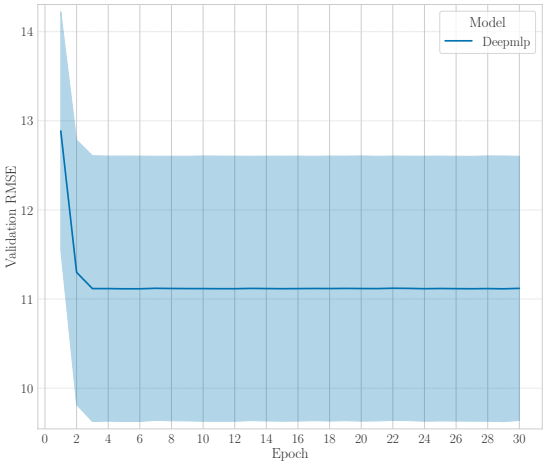


(a) Shallow Shielded MLP with Projected Gradient (b) Combined shallow Unmasked Models

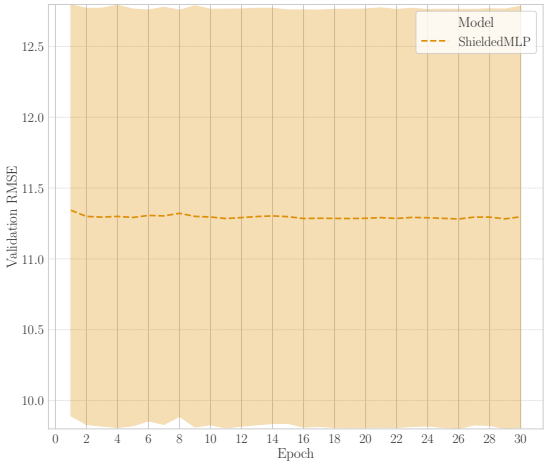
Figure C.12: Performance of the shallow Shielded MLP with Projected Gradient and the combined view of all shallow Unmasked Models on the URL dataset.

C.2.2 Masked Plots

Deep MLP

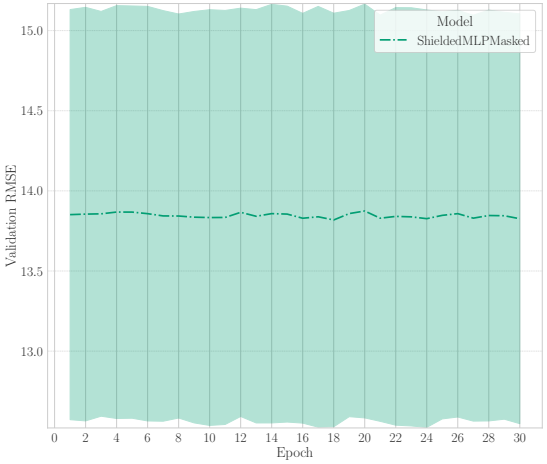


(a) Masked Deep MLP

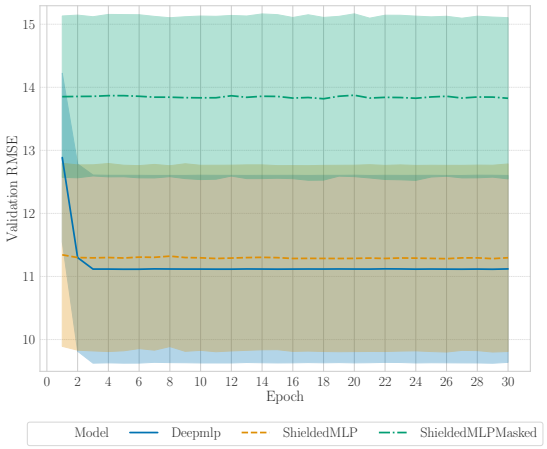


(b) Masked Deep Shielded MLP

Figure C.13: Performance comparison between the Masked Deep MLP and Masked Deep Shielded MLP models on the URL dataset.



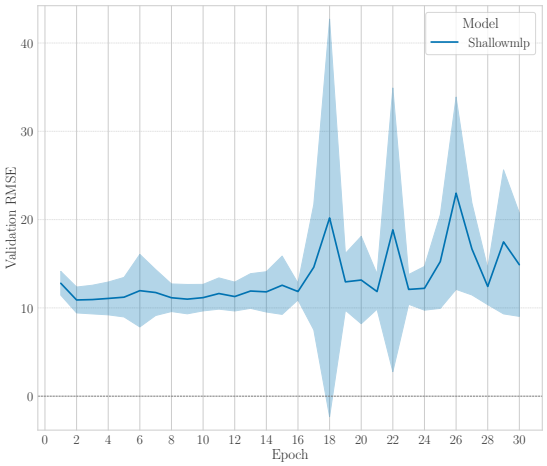
(a) Masked Deep Shielded MLP with Projected Gradient



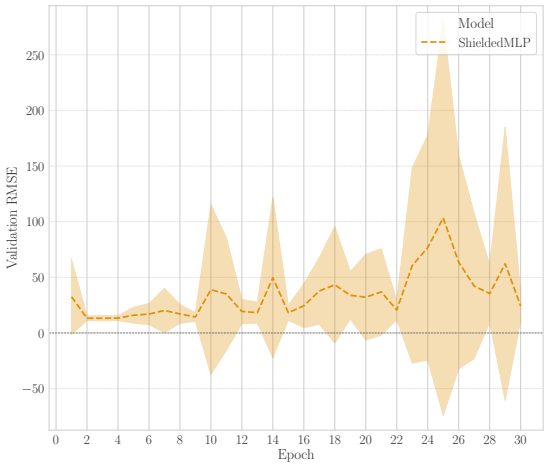
(b) Combined Masked Deep Models

Figure C.14: Performance of the Masked Deep Shielded MLP with Projected Gradient and the combined view of all Masked Deep Models on the URL dataset.

Shallow MLP

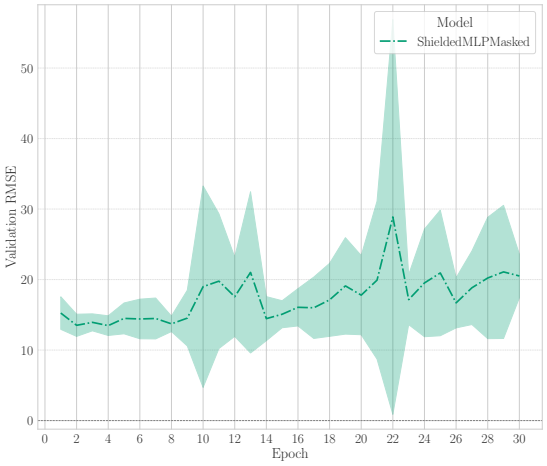


(a) Masked Shallow MLP

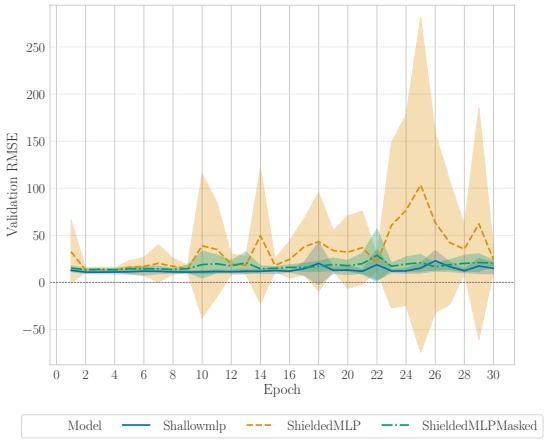


(b) Masked Shallow Shielded MLP

Figure C.15: Performance comparison between the Masked Shallow MLP and Masked Shallow Shielded MLP models on the URL dataset.



(a) Masked Shallow Shielded MLP with Projected Gradient



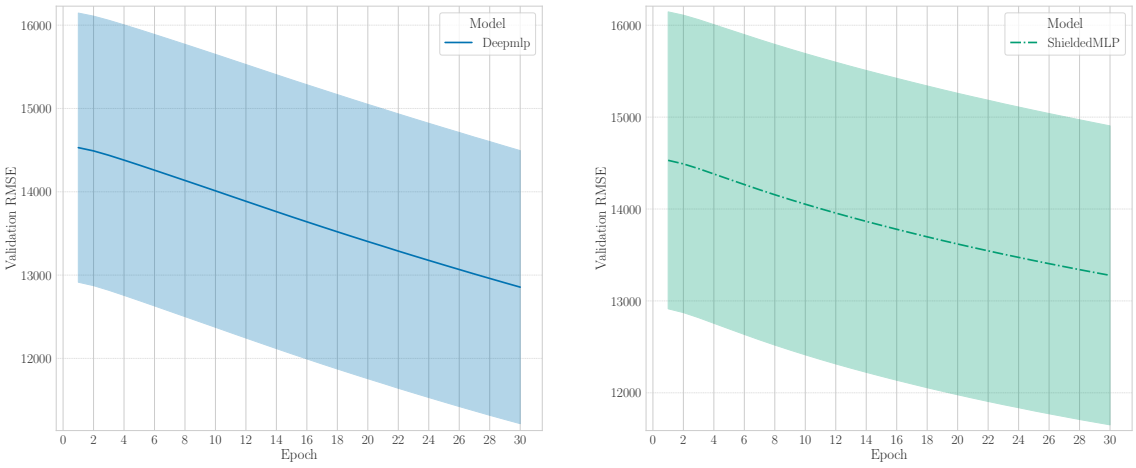
(b) Combined Masked Shallow Models

Figure C.16: Performance of the Masked Shallow Shielded MLP with Projected Gradient and the combined view of all Masked Shallow Models on the URL dataset.

C.3 News

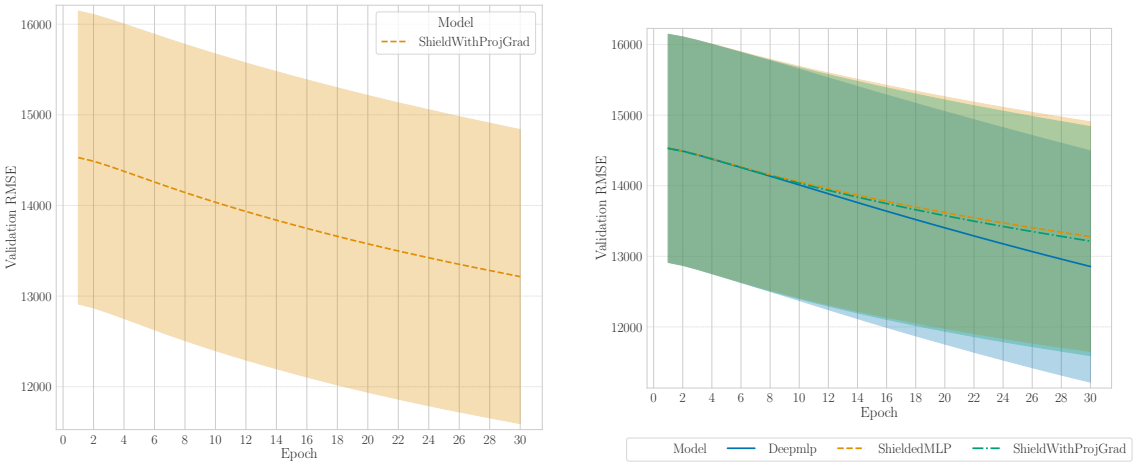
C.3.1 Projected Gradient Plots

Deep MLP



(a) Deep MLP (b) Deep Shielded MLP

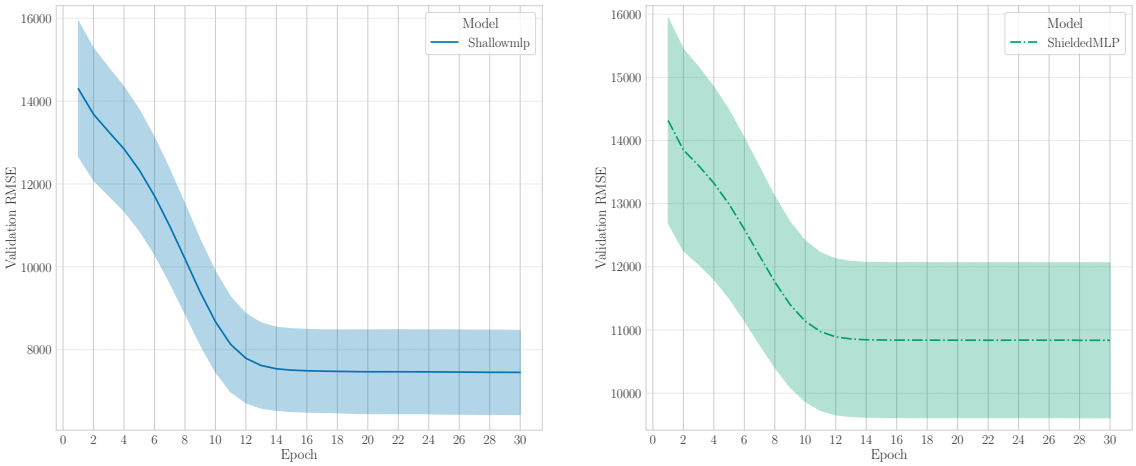
Figure C.17: Performance comparison between the Deep MLP and Deep Shielded MLP models on the News dataset.



(a) Deep Shielded MLP with Projected Gradient (b) Combined Deep Unmasked Models

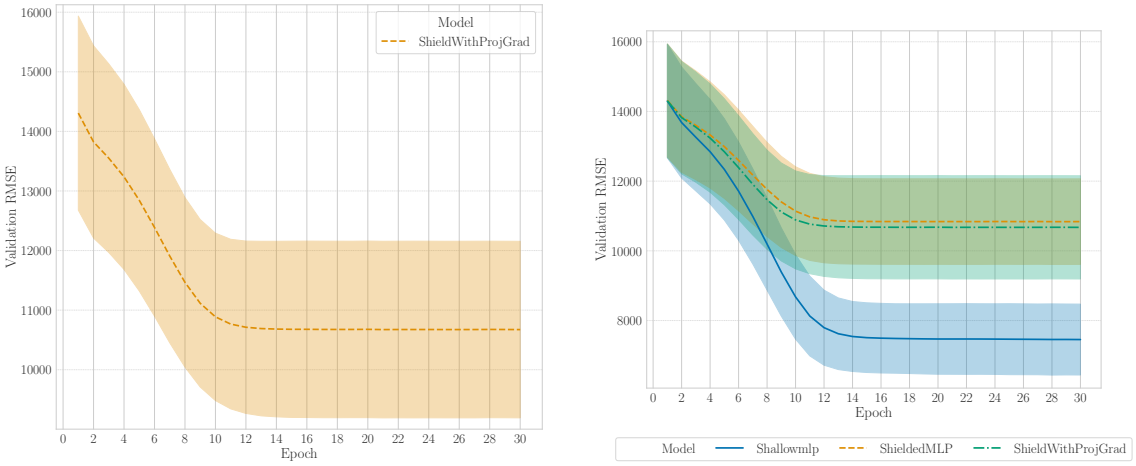
Figure C.18: Performance of the Deep Shielded MLP with Projected Gradient and the combined view of all Deep Unmasked Models on the News dataset.

Shallow MLP



(a) Shallow MLP (b) Shallow Shielded MLP

Figure C.19: Performance comparison between the Shallow MLP and Shallow Shielded MLP models on the NEWS dataset.

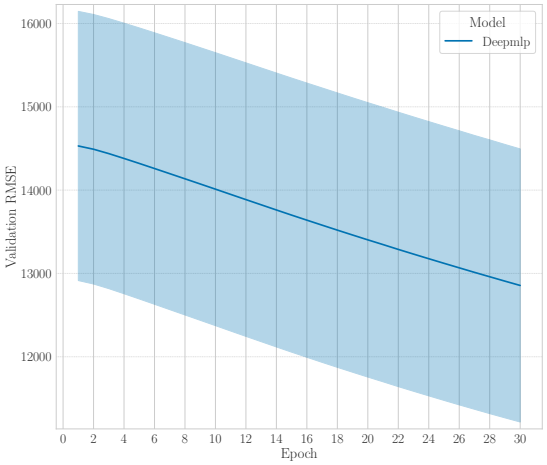


(a) Shallow Shielded MLP with Projected Gradient (b) Combined Shallow Unmasked Models

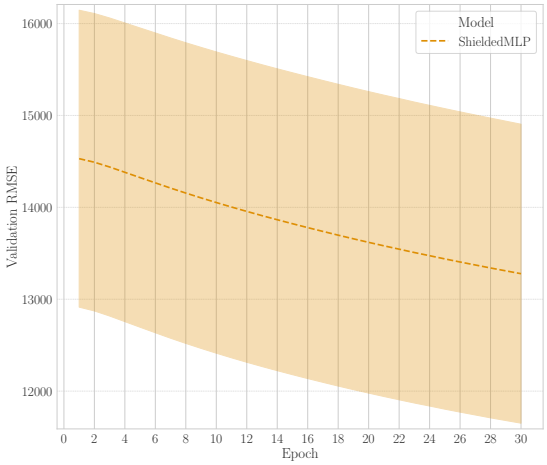
Figure C.20: Performance of the Shallow Shielded MLP with Projected Gradient and the combined view of all shallow Unmasked Models on the NEWS dataset.

C.3.2 Masked Plots

Deep MLP

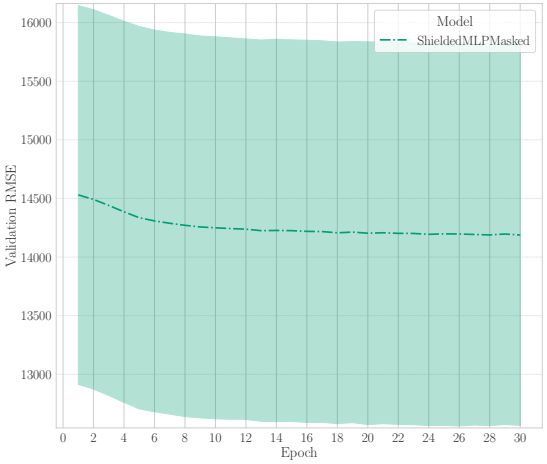


(a) Masked Deep MLP

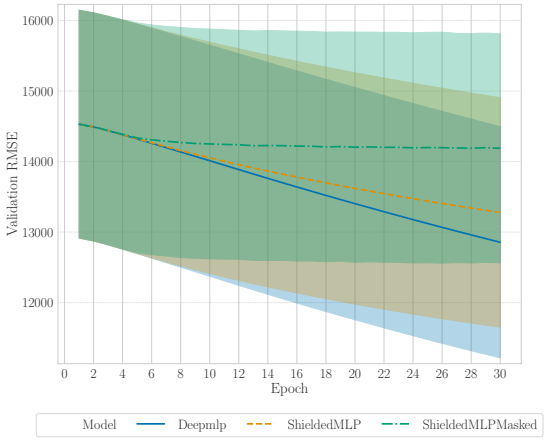


(b) Masked Deep Shielded MLP

Figure C.21: Performance comparison between the Masked Deep MLP and Masked Deep Shielded MLP models on the News dataset.



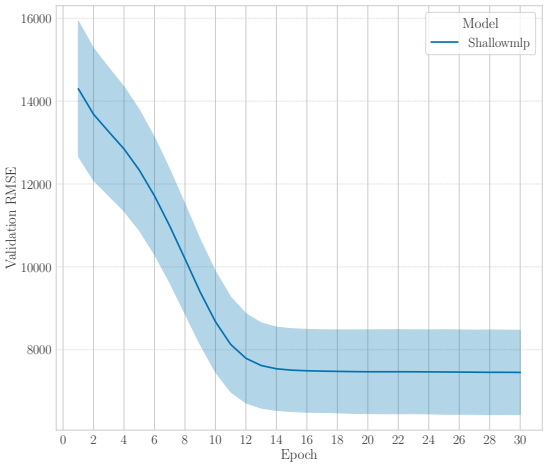
(a) Masked Deep Shielded MLP with Projected Gradient



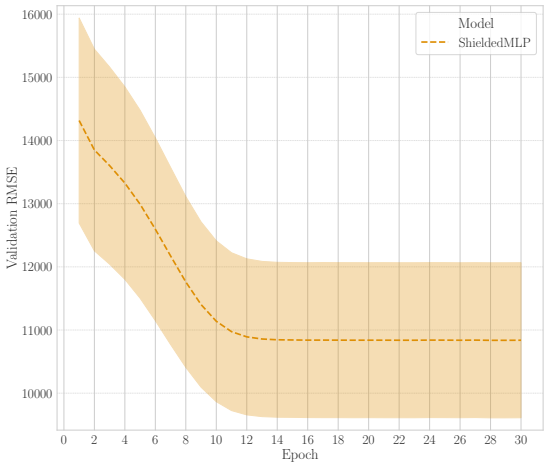
(b) Combined Masked Deep Models

Figure C.22: Performance of the Masked Deep Shielded MLP with Projected Gradient and the combined view of all Masked Deep Models on the News dataset.

Shallow MLP

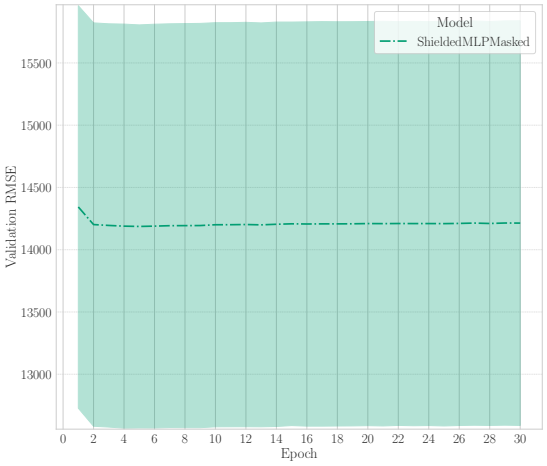


(a) Masked Shallow MLP

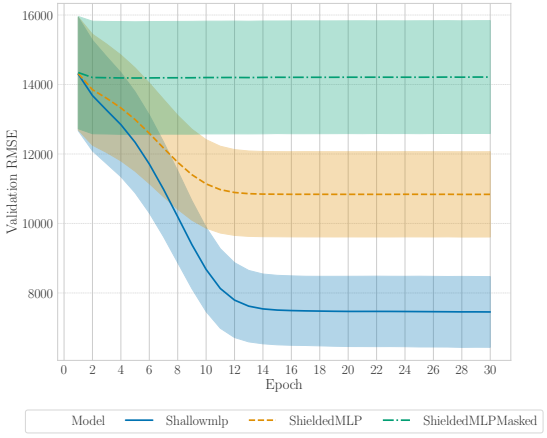


(b) Masked Shallow Shielded MLP

Figure C.23: Performance comparison between the Masked Shallow MLP and Masked Shallow Shielded MLP models on the NEWS dataset.



(a) Masked Shallow Shielded MLP with Projected Gradient



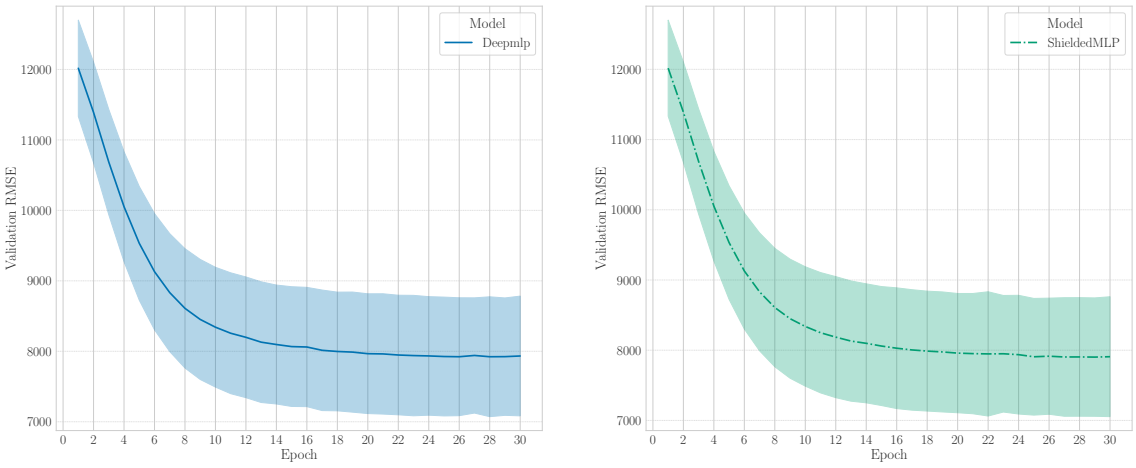
(b) Combined Masked Shallow Models

Figure C.24: Performance of the Masked Shallow Shielded MLP with Projected Gradient and the combined view of all Masked Shallow Models on the NEWS dataset.

C.4 LCLD

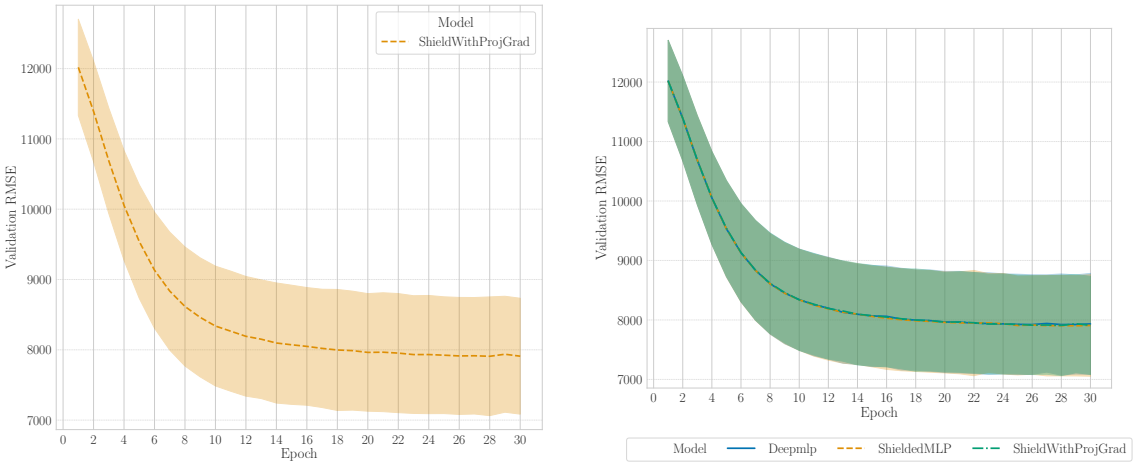
C.4.1 Projected Gradient Plots

Deep MLP



(a) Deep MLP (b) Deep Shielded MLP

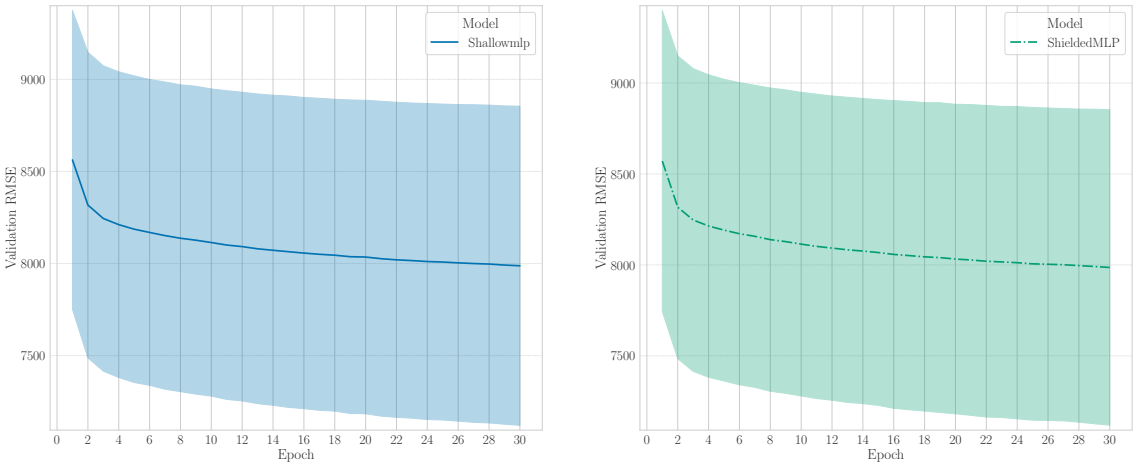
Figure C.25: Performance comparison between the Deep MLP and Deep Shielded MLP models on the LCLD dataset.



(a) Deep Shielded MLP with Projected Gradient (b) Combined Deep Unmasked Models

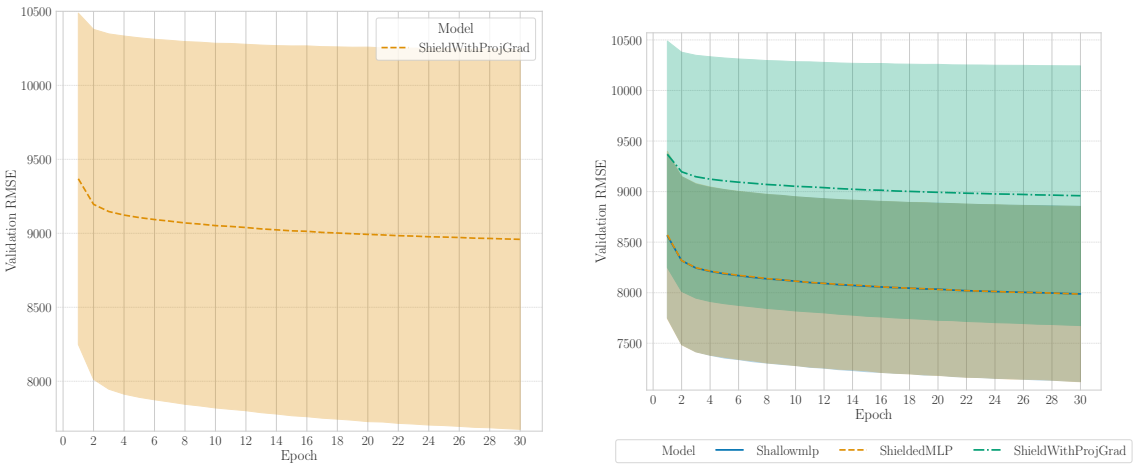
Figure C.26: Performance of the Deep Shielded MLP with Projected Gradient and the combined view of all Deep Unmasked Models on the LCLD dataset.

Shallow MLP



(a) Shallow MLP (b) Shallow Shielded MLP

Figure C.27: Performance comparison between the Shallow MLP and shallow Shielded MLP models on the LCLD dataset.



(a) Shallow Shielded MLP with Projected Gradient (b) Combined shallow Unmasked Models

Figure C.28: Performance of the shallow Shielded MLP with Projected Gradient and the combined view of all Shallow Unmasked Models on the LCLD dataset.

Bibliography

[1] M. C. Stoian, A. Tatomir, T. Lukasiewicz, and E. Giunchiglia, "Pishield: A pytorch package for learning with requirements," in *International Joint Conference on Artificial Intelligence*, 2024. [Online]. Available: https://arxiv.org/abs/2402.18285.

[2] E. Giunchiglia and T. Lukasiewicz, "Coherent hierarchical multi-label classification networks," in *Neural Information Processing Systems (NeurIPS)*, 2020. [Online]. Available: https://arxiv.org/abs/2010.10151v1.

[3] M. C. Stoian, S. Dyrmishi, M. Cordy, T. Lukasiewicz, and E. Giunchiglia, "How realistic is your synthetic data? constraining deep generative models for tabular data," in *ICLR*, 2024. [Online]. Available: https://arxiv.org/abs/2402.04823.

[4] E. Giunchiglia, "Beyond the convexity assumption: Realistic tabular data generation under quantifier-free real linear constraints," in *ICLR*, 2025. [Online]. Available: https://arxiv.org/abs/2502.18237.

[5] P. L. Donti, D. Rolnick, and J. Z. Kolter, "Dc3: A learning method for optimization with hard constraints," in *ICLR*, 2021. [Online]. Available: https://arxiv.org/abs/2104.12225.

[6] E. Giunchiglia and Anonymous, "Hard-constrained neural networks with universal approximation guarantees," in *Under Review for ICLR*, 2025.

[7] T. Frerix, M. Nießner, and D. Cremers, "Homogeneous linear inequality constraints for neural network activations," in *CVPR Deep Vision Workshop*, 2020. [Online]. Available: https://arxiv.org/abs/1902.01785v4.

[8] R. Wang, Y. Zhang, Z. Guo, T. Chen, X. Yang, and J. Yan, "Linsatnet: The positive linear satisfiability neural networks," in *ICML*, 2023. [Online]. Available: https://arxiv.org/abs/2407.13917.

[9] R. Sinkhorn and P. Knopp, "Concerning non-negative matrices and doubly stochastic matrices," in *Pacific Journal of Mathematics*, 1967. [Online]. Available: https://projecteuclid.org/journals/pacific-journal-of-mathematics/volume-21/issue-2/Concerning-nonnegative-matrices-and-doubly-stochastic-matrices/pjm/1102992505.full?tab=ArticleLink.

[10] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, 2015. [Online]. Available: https://arxiv.org/abs/1412.6980.

[11] S. Reddi, S. Kale, and S. Kumar, *Openreview discussion for "on the convergence of adam and beyond"*, https://openreview.net/forum?id=ryQu7f-RZ, Accessed: 2025-06-10, 2018.

[12] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, PMLR, 2020. [Online]. Available: https://arxiv.org/abs/2002.05709.

[13] P. Yin, S. Zhang, J. Lyu, S. Osher, Y. Qi, and J. Xin, "Understanding straight-through estimator in training activation quantized neural nets," in *International Conference on Learning Representations (ICLR)*, 2019. [Online]. Available: https://arxiv.org/abs/1903.05662.

[14] K. G. Murty, *Linear Complementarity, Linear and Nonlinear Programming*. Heldermann Verlag, 1988.

[15] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed. Springer, 2006.

[16] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, 2010.

[17] J. Pennington, S. S. Schoenholz, and S. Ganguli, "Resurrecting the sigmoid in deep learning through dynamical isometry: Theory and practice," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. [Online]. Available: https://arxiv.org/abs/1711.04735.

[18] S. S. Schoenholz, J. Gilmer, S. Ganguli, and J. Sohl-Dickstein, "Deep information propagation," in *International Conference on Learning Representations (ICLR)*, 2017. [Online]. Available: https://arxiv.org/abs/1611.01232.

[19] A. M. Saxe, J. L. McClelland, and S. Ganguli, "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks," in *International Conference on Learning Representations (ICLR)*, 2014. [Online]. Available: https://arxiv.org/abs/1312.6120.

[20] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*. SIAM, 1997.

[21] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd. SIAM, 2002.

[22] L. Lu, Y. Shin, Y. Su, and G. E. Karniadakis, "Dying relu and initialization: Theory and numerical examples," *Communications in Computational Physics*, 2020. [Online]. Available: https://arxiv.org/abs/1903.06733.

[23] W. Tarnowski, P. Warchoł, Y. Cho, P. Kurpios, K. Zyczkowski, and P. Robakiewicz, "Dynamical isometry is achieved in residual networks in a universal way for any activation function," in *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2019. [Online]. Available: `https://arxiv.org/pdf/1809.08848`.

[24] M. H. Kutner, C. J. Nachtsheim, J. Neter, and W. Li, *Applied linear statistical models*. McGraw-Hill/Irwin, 2005.

[25] J. Cohen, *Statistical power analysis for the behavioral sciences*. Routledge, 1988.

[26] T. Simonetto, S. Dyrmishi, S. Ghamizi, M. Cordy, and Y. Le Traon, "A unified framework for adversarial attack and defense in constrained feature space," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22)*, International Joint Conferences on Artificial Intelligence Organization, 2022. [Online]. Available: `https://arxiv.org/abs/2112.01156`.