PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Training Neural Networks to Perform Structured Prediction Task
Auteur: Author:	Maziar Sargordi
Date:	2024
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Sargordi, M. (2024). Training Neural Networks to Perform Structured Prediction Task [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/58007/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/58007/
Directeurs de recherche: Advisors:	Sarath Chandar Anbil Parthipan, & Amal Zouaq
Programme: Program:	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Training Neural Networks to Perform Structured Prediction Task

MAZIAR SARGORDI
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Avril 2024

© Maziar Sargordi, 2024.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Training Neural Networks to Perform Structured Prediction Task

présenté par **Maziar SARGORDI**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Foutse KHOMH, président
Sarath Chandar ANBIL PARTHIPAN, membre et directeur de recherche
Amal ZOUAQ, membre et codirectrice de recherche
Hervé LOMBAERT, membre

ACKNOWLEDGEMENTS

In the journey of academic research, the guidance and support of mentors and colleagues are invaluable. I am grateful to my primary supervisor, Prof. Sarath Chandar, and my co-supervisor, Prof. Amal Zouaq, for their guidance and support throughout this research. Their expertise and mentorship have been invaluable.

I am also deeply thankful to Dr. Quentin Fournier, a Postdoctoral Fellow in our group, whom I learned a lot from during our weekly meetings. His assistance and willingness to share his knowledge have been incredibly invaluable.

My gratitude extends to Prof. Gilles Pesant and Dr. Hélène Verhaeghe, whose contributions to my research were significant, specifically for conducting the constraint programming part of the experiments. Their advice, support, and insights have enriched my understanding and greatly enhanced the quality of this work.

I also want to thank my lab mates, especially Daphné Lafleur.

This thesis has indeed been a collaborative effort, and the contributions of these individuals have been essential to its completion. I am thankful for their support and am honored to have worked alongside them.

Last but not least, I am eternally grateful to my family – Zari, Abdi, Shima, and especially my partner Beheshteh, whose support and encouragement made my journey in computer engineering possible.

RÉSUMÉ

Malgré leurs nombreux succès sur diverses tâches complexes, les réseaux de neurones profonds ont encore du mal à apprendre la structure combinatoire, où de multiples sorties discrètes ont des relations interconnectées régies par des contraintes. Ceci est particulièrement vrai lorsqu'il n'y a pas suffisamment de données pour que le modèle apprenne la structure de sortie.

La programmation par contraintes, un type d'algorithme non-apprenant, se concentre sur la structure. Elle a un passé développé et couronné de succès dans la reconnaissance de structures combinatoires qui reviennent fréquemment, et dans le développement d'algorithmes avancés pour extraire des informations de ces structures. En particulier, nous nous intéressons à la fréquence relative d'une affectation donnée variable-valeur dans cette structure combinatoire.

Le cadre de programmation par contraintes avec propagation de croyance généralise ce modèle en propageant ces fréquences relatives issues d'un modèle de programmation par contraintes pour approximer les fonctions de masse de probabilité marginales de chaque variable. Ces probabilités marginales estimées sont utilisées comme pénalités au sein de la fonction de perte, améliorant l'apprentissage et l'efficacité du réseau neuronal à partir des échantillons.

Dans ce mémoire, nous proposons de former un réseau de neurones pour générer une sortie qui s'aligne sur une structure combinatoire exprimée comme un modèle de programmation par contraintes. Ceci est réalisé en calculant une fonction de perte qui inclut les marginales déterminées par la programmation par contraintes avec un solveur de propagation de croyance.

Nous soutenons que ce modèle offre une intégration plus naturelle de la programmation par contraintes et des réseaux neuronaux. Nous apportons des preuves pratiques que la formation du modèle en utilisant cette approche améliore considérablement ses performances, surtout lorsqu'il existe une quantité limitée de données disponibles. Nos résultats sur le problème du Carré Latin Partiel indiquent une amélioration constante de la précision du modèle par rapport aux méthodes existantes.

ABSTRACT

Despite their numerous successes on various challenging tasks, deep neural networks still struggle to learn combinatorial structure, where multiple discrete outputs have interconnected relationships governed by constraints, especially when there is not enough data for the model to learn the output structure.

Constraint programming, a type of non-learning algorithm, focuses on structure. It has a developed and successful past in recognizing combinatorial structures that frequently recur, and in developing advanced algorithms to extract information from these structures. In particular, we are interested in the relative frequency of a given variable-value assignment in that combinatorial structure.

The constraint programming with belief propagation framework generalizes this model by propagating these relative frequencies from a constraint programming model to approximate the marginal probability mass functions of each variable. These estimated marginal probabilities are used as penalties within the loss function, improving the neural network's learning and efficiency from samples.

In this thesis, we propose to train a neural network to generate output that aligns with a combinatorial structure expressed as a constraint programming model. This is achieved by calculating a loss function that includes marginals determined by constraint programming with a belief propagation solver.

We argue that this model offers a more natural integration of constraint programming and neural networks. We offer practical evidence that training the model using this approach significantly enhances its performance, especially when there is a limited amount of data available. Our results on the Partial Latin Square problem indicate consistent improvement in the accuracy of the model over the existing methods.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

RÉSUMÉ iv

ABSTRACT v

TABLE OF CONTENTS vi

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF SYMBOLS AND ACRONYMS xi

LIST OF APPENDICES xii

CHAPTER 1 INTRODUCTION 1
1.1 Research Questions 2
1.2 Objectives 2
1.3 Thesis Outline 3

CHAPTER 2 BACKGROUND 4
2.1 Fundamentals of Machine Learning 4
2.1.1 Neural Networks 4
2.1.2 Classification 6
2.1.3 Gradient Descent 7
2.1.4 Training Neural Networks 9
2.2 Constraint Programming 10
2.2.1 A simple CP model using a Partial Latin Square 11
2.2.2 Declaration of Constraints 12
2.2.3 Propagation of Constraints 12
2.2.4 Understanding the Filtering Algorithm with an Example 13
2.2.5 Belief Propagation 13
2.2.6 CP-BP with an Example 15

CHAPTER 3 LITERATURE REVIEW 17

3.1 Pure Machine Learning Approaches . . . 17
3.2 Hybrid Approaches . . . 18
3.2.1 Tightly Integrated Approaches . . . 19
3.2.2 Loss-Guided Approaches . . . 20

CHAPTER 4 METHODOLOGY . . . 23
4.1 Problem Definition and Motivation . . . 23
4.1.1 Partial Latin Square . . . 23
4.1.2 Motivation for Hybrid Approach . . . 24
4.2 Loss Functions and Feedback . . . 26
4.2.1 CP Based . . . 26
4.2.2 CP-BP Based . . . 27
4.3 Modeling Details . . . 28
4.3.1 Input, Output, and Labels . . . 29
4.3.2 Neural Network Architecture . . . 29
4.3.3 Masking Mechanism . . . 30
4.4 Training and Evaluation . . . 31
4.4.1 Training . . . 31
4.4.2 Testing . . . 31
4.4.3 Dataset . . . 32
4.5 Baselines . . . 32
4.5.1 Random . . . 32
4.5.2 Symbolic . . . 33
4.5.3 Multiplayer Perception . . . 33
4.5.4 Agnostic . . . 33
4.5.5 Silvestri with CP . . . 34
4.5.6 Silvestri with CP-BP . . . 34

CHAPTER 5 EXPERIMENTS . . . 35
5.1 Training MLP with Labels . . . 36
5.2 Problem Difficulty . . . 40
5.3 Dataset Size . . . 42
5.4 Training without Label . . . 45
5.5 Discussion . . . 48

CHAPTER 6 CONCLUSION . . . 50
6.1 Summary of Works . . . 50

6.2 Limitations 50
6.3 Future Research 51

REFERENCES 52

APPENDICES 57

LIST OF TABLES

Table 2.1	Marginals: (a) True marginals, (b) Initial marginals, (c) Marginals after 1 iteration, (d) Marginals after 10 iterations.	16
Table 5.1	Minimum and average feasibility ratios of MPL for different data sizes + Random model	38
Table 5.2	Minimum and average feasibility ratios for different changes in the MLP models	39
Table 5.3	Minimum feasibility ratios for different models	42
Table 5.4	Minimum feasibility ratios for different models and different data sizes	44
Table 5.5	Minimum feasibility ratios for different models trained without (and with) labels	47
Table 5.6	Minimum feasibility ratios for our best models trained without labels with 100% (and 10%) of data	48
Table A.1	General Hyperparameters	57
Table A.2	Hyperparameters for PLS of Size 7	57
Table A.3	Hyperparameters for PLS of Size 10	58
Table A.4	Hyperparameters for PLS of Size 12	58
Table B.1	CP-BP Marginal Calculation Times	59

LIST OF FIGURES

Figure 2.1	Schematic of a simple neural network with one hidden layer for house price prediction.	5
Figure 2.2	Gradient Descent on a Parabolic Loss Function	8
Figure 2.3	Training Process as a Series of Steps	10
Figure 4.1	Illustration of PLS assignments with local constraints and feasibility considerations	24
Figure 5.1	Average feasibility ratio of the MLP.	36
Figure 5.2	Feasibility ratio of the trained MLP with respect to the proportion of filled cells.	37
Figure 5.3	Feasibility ratio of uniform random assignments.	38
Figure 5.4	Feasibility ratio of the trained neural network.	39
Figure 5.5	Feasibility ratio of Agnostic, Symbolic, Silvestri, and Silvestri with CP-BP models	40
Figure 5.6	Feasibility ratio of our trained models with respect to the proportion of filled cells	41
Figure 5.7	Feasibility ratio of Agnostic, Silvestri, and Silvestri with CP-BP models with 10% and 50% of data	43
Figure 5.8	Feasibility ratio of our models by training on 10% or 100% of the dataset	44
Figure 5.9	Training without the labels with no change in softmax and with no masking.	46
Figure 5.10	Training without the labels with cell-wise softmax and masking. Here we trained	46
Figure 5.11	Training on 10% of data without the labels with cell-wise softmax and masking.	48

LIST OF SYMBOLS AND ACRONYMS

DNN	Deep Neural Network
NN	Neural Network
ML	Machine Learning
MLP	Multilayer Perceptron
FCNN	Fully Connected Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory Network
GAN	Generative Adversarial Network
GNN	Graph Neural Network
RL	Reinforcement Learning
AI	Artificial Intelligence
CP	Constraint Programming
CP-BP	Constraint Programming with Belief Propagation
CE	Cross Entropy
BCE	Binary Cross Entropy
CCE	Categorical Cross Entropy
KL	Kullback–Leibler
SGD	Stochastic Gradient Descent
ReLU	Rectified Linear Unit
CSP	Constraint Satisfaction Problem
PLS	Partial Latin Square
MAXSAT	Maximum Satisfiability
ASP	Answer Set Programming
SPEN	Structured Prediction Energy Network
STE	Straight-Through Estimator
SPL	Semantic Probabilistic Layer

LIST OF APPENDICES

Appendix A PLS Hyper-parameters 57
Appendix B CP-BP Marginal Computation Times 59

CHAPTER 1 INTRODUCTION

Deep Neural Networks (DNNs) have transformed the realm of Artificial Intelligence (AI), demonstrating effectiveness in various fields such as speech recognition [1], natural language processing [2], and computer vision [3]. However, when it comes to structured prediction problems that involve multiple interdependent discrete outputs where certain constraints are expected to be met, DNNs often encounter challenges. This is particularly true when these constraints are deterministic and there is insufficient data to learn an input-output mapping. Yet situations involving these types of structured problems are numerous, as shown in the research by Lee et al. (2019) [4].

Constraint Programming (CP) focuses on identifying structures, with a rich history of recognizing frequently occurring combinatorial substructures and creating sophisticated inference algorithms for these patterns. These algorithms can do more than just check for satisfiability; they can measure the extent of a constraint's violation, checking if a particular variable assignment is part of at least one solution, or in the context of CP with Belief Propagation (CP-BP), they count the number of solutions that include a specific variable assignment. This leads to effectively computing marginals over individual variables using CP-BP.

To this end, we explore incorporating CP in the training phase of DNNs by adding an extra term in the loss function to incorporate the CP-BP probabilities in the neural networks. Our focus is on understanding how the CP-BP framework [5], with its provision of marginal probability mass functions, may present more intuitive and informative possibilities for integration than traditional CP methods.

Before moving into the research question, let's consider a familiar example to better understand the challenges in structured prediction: Sudoku. Sudoku is a puzzle involving a 9×9 grid, where the objective is to fill the grid so that each row, column, and 3×3 subgrid contains all digits from 1 to 9. The constraint here is that no digit can repeat in any row, column, or subgrid.

Sudoku is solvable with neural networks, particularly convolutional neural networks (CNNs). The model can achieve decent accuracy in solving Sudoku [6]. However, it might not reach 100% accuracy and can also be slower than traditional logic-based algorithms. That is because the model usually requires more data points to learn when the output is structured. This becomes particularly challenging when there is limited data, i.e., a small number of solved puzzles for the network to learn from. The neural network might predict numbers that fill the grid but often might violate the Sudoku rules.

This is where CP, with its focus on structure and constraints, can significantly aid DNNs. By integrating CP, especially the CP-BP framework, into the training of DNNs, the network can be guided to not only predict the numbers but also to respect the inherent constraints of Sudoku. This integration could potentially lead to more accurate and rule-compliant solutions, even with limited training data.

With this Sudoku example in mind, our research questions are as follows:

1.1 Research Questions

1. **Evaluating CP-BP's Role in Complex Structured Prediction Tasks:** How does the integration of Constraint Programming with Belief Propagation influence neural networks in handling increasing complexity within structured prediction tasks? Specifically, what changes occur in DNN performance as the complexity of these tasks escalates?

2. **CP-BP's Role in Structured Prediction with Less Data:** How does CP-BP help neural networks in structured prediction tasks under conditions of limited data availability? This question explores the effectiveness of neural networks integrated with CP-BP in scenarios where the size of the training dataset is constrained.

3. **Leveraging CP-BP for Label-Free Training in Structured Prediction:** In the context of structured prediction tasks with high complexity and no labeled data, how can the CP-BP framework enhance the training and overall performance of neural networks? This question investigates the potential of CP-BP methodologies to compensate for the absence of labeled data in training neural networks.

These questions examine the role and impact of CP-BP in enhancing neural network performance across different challenges in structured prediction, including increasing task complexity, data availability, and the absence of labeled data. We employed the Partial Latin Square (PLS) problem to conduct all of our experiments.

1.2 Objectives

The long-term goal of this project is to extend the applicability of neural networks, enabling efficient implementation for complex structured tasks. We want to see how combining CP-BP with neural networks changes their performance. We also plan to look at how the size of the data affects how well these neural networks perform. Lastly, we want to develop and

test methods for integrating CP methodologies into neural network training, particularly in scenarios with limited or no labeled data in the PLS problem.

1.3 Thesis Outline

This thesis is organized in the following manner:

In Chapter 2, we jump into the essential background knowledge necessary for understanding the research. This chapter covers the fundamentals of neural networks, including training in a neural network, and an introduction to the key concepts and techniques in Constraint Programming. This background serves as a foundation for the next chapters.

Chapter 3 is dedicated to reviewing existing literature on structured prediction tasks. This chapter highlights the current state of research in this field and also identifies the limitations. By examining previous studies and their methodologies, we position our research within the existing literature.

Chapter 4 outlines the specific tasks addressed in this work. We describe the methodology employed, including the integration of CP-BP with Neural Networks. This section highlights the novelty of our model and approach. Additionally, it details the datasets we utilize in our research.

Chapter 5, presents a series of experiments designed to evaluate the performance of our proposed model for solving the PLS completion problem. We investigate how factors such as problem complexity, dataset size, and the availability of labeled data influence the model's effectiveness. Additionally, we compare our approach against the baselines and explore the potential of CP and CP-BP to guide the learning process, particularly in scenarios where labeled data is limited.

Finally, Chapter 6 presents a summary of the work, discussing the key findings and contributions of the research. This chapter also addresses the limitations of the current study and proposes directions for future research.

CHAPTER 2 BACKGROUND

This chapter is dedicated to presenting the fundamental background that is essential to comprehend the scope of our research.

2.1 Fundamentals of Machine Learning

Machine Learning (ML), a subset of Artificial Intelligence (AI), focuses on learning complex relationships between input and the corresponding output in data. The primary objective of ML is to predict outputs from given inputs, using previously established input-output correlations. Consider the function $F(\mathbf{x}) = y$ as a representative input-output mapping. For straightforward tasks, F can be relatively simple. For example, in linear regression with vector $\mathbf{x}$ and scalar y, the goal is to find the parameters of a linear function that best maps $\mathbf{x}$ to y. This task is simple and has a closed-form solution. However, it becomes very expensive to calculate this close-form solution when the dataset is extensive. In these cases, instead of finding a closed-form solution, an alternative method is required. Here, the gradient descent algorithm becomes essential. This iterative technique incrementally approximates the optimal parameter values and is more practical for more complex tasks and large datasets.

2.1.1 Neural Networks

Neural Networks (NNs), inspired by the biological neural networks in our brains, are an important family of functions in the field of ML [7–10]. They are particularly great at learning complex patterns from datasets and making regression or classifications based on learned parameters. NNs are used in a variety of architectures, some of which include: Convolutional Neural Networks (CNNs) [11,12], sequence models such as Recurrent Neural Networks (RNNs) [13] and Long Short-Term Memory Networks (LSTMs) [14], Autoencoders [15], Generative Adversarial Networks (GANs) [16], Transformer Networks [17] and Graph Neural Networks (GNNs) [18].

In this thesis, we go over the foundational concepts of Fully Connected Neural Networks (FCNNs), also known as Multilayer Perceptrons (MLP). These networks are key parts of many different deep learning systems and are also used in our experiments (see chapter 4). A neural network typically consists of layers of interconnected neurons. These layers are categorized into three types:

- **Input Layer:** This is where the network receives data as input.

- **Hidden Layers:** These consist of one or more layers between the input and output layers. The *hidden* aspect refers to their inaccessibility for direct observation from the input and output. In these layers, complex computations are done through a network of interconnected neurons. Each neuron in these layers applies a set of non-linear functions to the data that it receives. These transformations allow the network to learn complex patterns and relationships within the input data and make the network capable of predictions or decisions.
- **Output Layer:** Produces the final output of the network.

To illustrate, let's consider a simple neural network for predicting house prices. The input layer might include neurons representing features such as the size of the house, number of rooms, and location. The output is the predicted price. Figure 2.1 shows the semantics of the neural network for this example, which is a regression task.

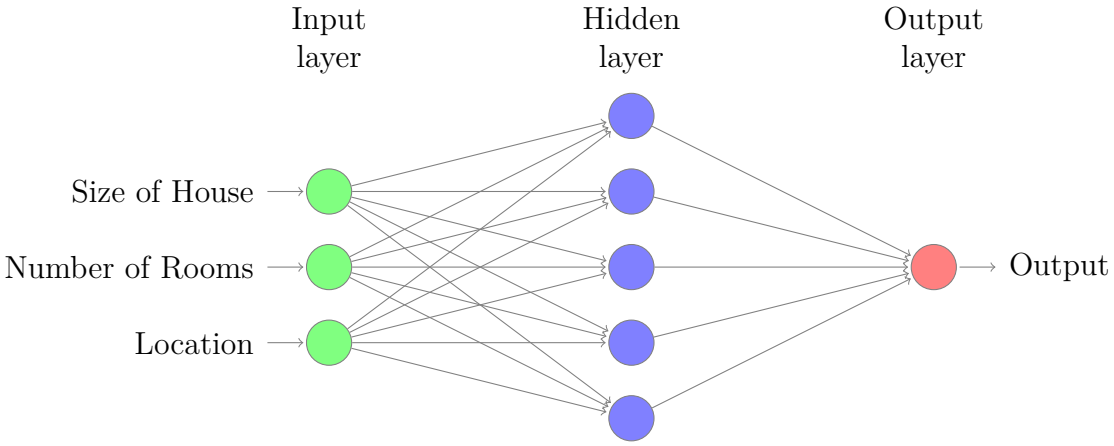


Figure 2.1 Schematic of a simple neural network with one hidden layer for house price prediction.

The core operation in a neural network involves weighted summation and activation functions.

Weighted Summation: The value in each neuron is a weighted sum of its inputs, represented as:

$$z = b + \sum_{i=1}^{n} w_i x_i \tag{2.1}$$

where x_i are input values, w_i are weights, and b is a bias term.

Activation Function: To introduce some non-linearity, the weighted sum is usually passed through an activation function σ, giving the output:

$$y = \sigma(z) \tag{2.2}$$

Common activation functions include sigmoid, ReLU, and tanh.

Learning: The network learns by adjusting weights and biases to minimize the difference between predictions and actual values (the labels). Generally, we minimize a loss function which may be the difference between the prediction and expected output but may also include penalties. This is usually done using backpropagation and a gradient descent optimization algorithm.

Neural networks are powerful learning algorithms in machine learning, capable of modeling complex patterns in input data. By adjusting weights and biases during the training process, they can predict accurately, making them invaluable in various applications from simple regression tasks like house price prediction to complex image and speech recognition.

2.1.2 Classification

After understanding the basics of NNs, let's jump into their application in classification tasks. Classification involves categorizing data into predefined classes (labels). The difference between classification and regression is that classification's outputs are discrete while regressions are continuous.

Consider a NN designed to classify handwritten digits (0 to 9). Each input image is converted into a list of pixel intensity values. The goal is to map each input image to a correct class (digit).

In this scenario, the network has an input layer corresponding to the number of pixels, one or more hidden layers, and an output layer with 10 neurons, each representing a digit from 0 to 9.

Layer Dynamics: As with any neural network, each neuron in a layer can be connected to all neurons in the previous layer, with these connections having adjustable weights.

Feature Learning: Hidden layers are for learning various features from the input data. In the context of digit recognition, early layers might learn to identify edges and curves, whereas deeper layers could recognize more complex shapes like parts of digits.

The critical component in a classification neural network is the **softmax function** that the output passes through. Softmax function converts the output of the network into a probability distribution over predicted classes. Given an output vector z from the last hidden

layer, the softmax function for the j-th output is given by:

$$\text{Softmax}(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{2.3}$$

where K is the total number of classes (10 for digit recognition) and e is the base of the natural logarithm. This function ensures that the sum of the probabilities of all output classes equals 1, making the output layer a probability distribution. To train the network, we can use the **cross-entropy loss function**. This function compares the predicted output probability distribution with the actual labels. The loss is calculated as follows:

$$L = -\sum_{i=1}^{K} y_i \log(\hat{y}_i) \tag{2.4}$$

where K is the number of classes, y_i is the true label (1 for the correct class and 0 for others), and $\hat{y}_i$ ($\text{Softmax}(z)_j$) is the predicted probability of the class.

The network is trained using gradient descent to minimize the cross-entropy loss function. In each iteration, the weights are adjusted in the direction that reduces the loss. This involves computing the gradient of the loss function with respect to each weight by the chain rule, propagating this gradient backward through the network which is called backpropagation and will be explained later on.

Here, the classification algorithm is explained through a digit recognition example, which represents a more advanced application of NNs. The use of the softmax function and cross-entropy loss, combined with algorithms like backpropagation, enables these networks to categorize data with high accuracy. NNs' flexibility makes them invaluable for a wide range of applications beyond simple digit recognition, from facial recognition systems to medical diagnosis tools.

2.1.3 Gradient Descent

Gradient Descent is a fundamental algorithm in ML and NNs, crucial for optimizing the model's performance by iteratively updating the parameters of the model. Gradient Descent is an optimization algorithm used to usually minimize some function by iteratively moving towards the minimum, which is often the negative of the gradient. In the context of neural networks, this function is typically a loss function that measures the difference between the neural networks prediction and the actual label.

Consider a loss function $L(\theta)$, where θ is the parameters of the neural network including weights and biases. The goal of gradient descent is to find the value of θ that minimizes

$L(\theta)$.

The Gradient Descent Algorithm

The algorithm updates the parameters in the opposite direction of the gradient of the loss function. The update rule for each parameter θ_i is given by:

$$\theta_i = \theta_i - \alpha \frac{\partial L}{\partial \theta_i} \tag{2.5}$$

where α is the learning rate, a hyperparameter that determines the magnitude of the steps taken towards the minimum.

A high learning rate might overshoot the minimum and lead to divergence. While, a low learning rate causes it to converge slowly, increasing computational cost.

In batch gradient descent, the gradient of the loss is calculated over the entire input dataset. This can be computationally expensive (but parallelizable) and slow for large datasets. On the other hand, Stochastic Gradient Descent (SGD) updates the parameters for each training example one by one. SGD is faster per update and can help escape local minima but might be more unstable. Mini-batch gradient descent combines the concepts of batch and stochastic gradient descent. It updates parameters for every mini-batch of training examples. It usually has the best features of both batch gradient descent and the SGD.

We can visualize the gradient descent process using a simple $2D$ graph in Figure 2.2.

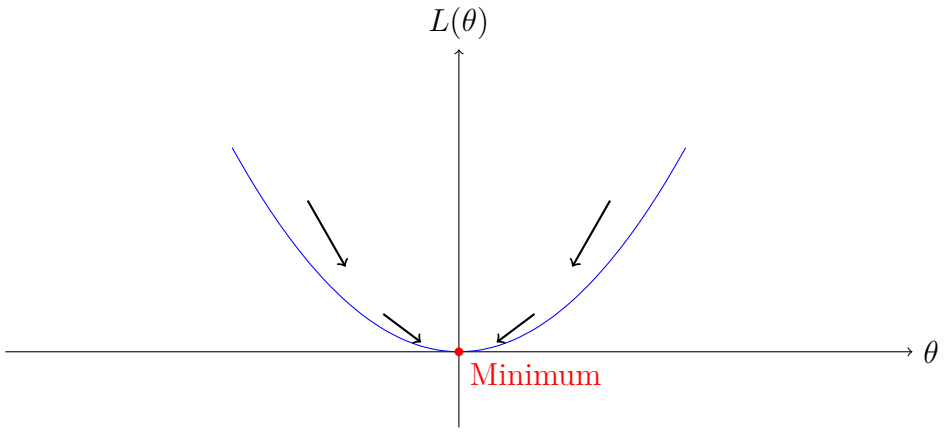


Figure 2.2 Gradient Descent on a Parabolic Loss Function

2.1.4 Training Neural Networks

Training a neural network involves adjusting its weights and biases to minimize the error in its predictions. The most common method for training is backpropagation. This section outlines the backpropagation process and its implementation in neural network training.

The Essence of Backpropagation Backpropagation is a method for efficiently computing gradients of the loss function with respect to the parameters of the network. It applies the chain rule of calculus recursively from the output layer back to the input layer. The essence of backpropagation can be summarized by the following steps:

1. **Forward Pass:** Compute the output of the network and the loss. This involves passing the input data through each layer of the network and using the objective function to calculate the loss. For a network with L layers, the output of the l^{th} layer is given by:

$$a^{(l)} = \sigma(W^{(l)}a^{(l-1)} + b^{(l)}) \tag{2.6}$$

where $a^{(l)}$ is the activation of the l^{th} layer, $W^{(l)}$ and $b^{(l)}$ are the weights and biases of the l^{th} layer and σ is the activation function.

2. **Backward Pass:** Compute the gradient of the loss function with respect to each parameter (weights and biases) in the network. This is achieved by applying the chain rule to propagate the error backward from the loss to all the parameters (θ) of the model.

The essence of backpropagation is the computation of the gradient of the loss function. For a loss function L, the gradient with respect to the parameters $\theta^{(l)}$ (which include both weights $W^{(l)}$ and biases $b^{(l)}$) in layer l is given by:

$$\frac{\partial L}{\partial \theta^{(l)}} = \frac{\partial L}{\partial a^{(l)}} \cdot \frac{\partial a^{(l)}}{\partial \theta^{(l)}} \tag{2.7}$$

Here, $\frac{\partial L}{\partial a^{(l)}}$ represents the error at layer l and is computed in a recursive manner from the subsequent layer. The recursive relation for this error term, $\delta^{(l)}$, is given by:

$$\delta^{(l)} = \left((\theta^{(l+1)})^T \delta^{(l+1)}\right) \odot \sigma'(z^{(l)}) \tag{2.8}$$

In this relation, $\delta^{(l)}$ is the error at layer l, σ' is the derivative of the activation function and $z^{(l)}$ is the input to the activation function at layer l. The symbol $\odot$ is element-wise multiplication.

Backpropagation allows the efficient computation of gradients required for parameter up-

dates. Through the iterative application of backpropagation and weight updates, the neural network learns to reduce the error in its predictions and improve its performance on the downstream task. Figure 2.3 shows steps in the training process.

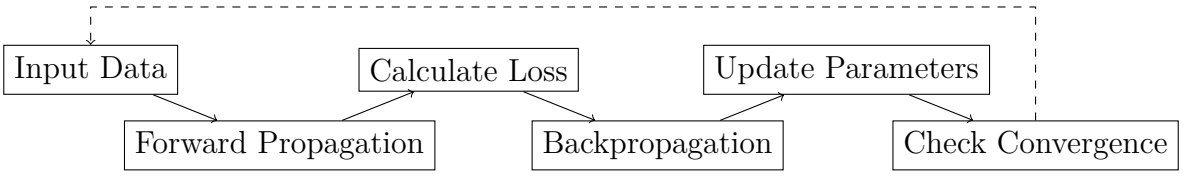


Figure 2.3 Training Process as a Series of Steps

By repeatedly applying steps in Figure 2.3 over multiple epochs, the network gradually improves its predictions, learning from the input data it is trained on.

2.2 Constraint Programming

Unkile neural networks that learn by finding a set of optimal parameters for a complex function, no learning is involved in Constraint Programming (CP). Like many other computational methods used for combinatorial problems, CP represents the problem through formal mathematical modeling. However, it distinguishes itself by using a language that includes high-level elements (constraints), which explicitly expose much of the combinatorial structure of the problem.

A Constraint Satisfaction Problem (CSP) is represented as $\langle X, D, C \rangle$. This structure comprises three key elements:

1. $X = \{x_1, x_2, \ldots, x_n\}$, a finite set of variables,
2. $D = \{D(x_1), D(x_2), \ldots, D(x_n)\}$, representing finite domains for each variable, which are the possible value assignments, and
3. $C = \{c_1, c_2, \ldots, c_m\}$, a finite set of constraints. These constraints are relations applied to subsets of X.

The objective of a CSP is to find a complete instantiation of the variables:

$$(x_1, x_2, \ldots, x_n) = (v_1, v_2, \ldots, v_n),$$

where each v_i is an element of $D(x_i)$, and this instantiation must satisfy all constraints in C.

Constraints in a CSP can be diverse. For example, they may be expressed as:

$$\textsc{Sum}(x_1, x_2, \ldots, x_k, s),$$

meaning the sum of variables x_1 to x_k equals s. Another example is:

$$\textsc{AllDifferent}(x_1, x_2, \ldots, x_k),$$

which indicates that all these variables must have different values. These forms allow for a more concise and efficient representation of constraints.

In the context of constraint satisfaction problems, each constraint type represents a unique combinatorial structure and contains specialized inference algorithms. These algorithms focus on the variable domains, filtering out values that either conflict with the constraint or would violate the constraint beyond an acceptable level [19].

In short, the goal is to set variables to values that follow certain rules. This approach uses clear rules known ahead of time instead of learning from data. For each variable, we define what values it can take and what rules (constraints) it must follow. The CP solver then finds one or more solutions that fit these constraints.

2.2.1 A simple CP model using a Partial Latin Square

The Partial Latin Square problem is a type of combinatorial problem that extends the concept of a Latin square. A Latin square of order n is an $n \times n$ grid filled with n different numbers in such a way that each number occurs exactly once in each row and exactly once in each column. In a Partial Latin Square, some of the cells in the grid are pre-filled with numbers, and the objective is to complete the grid following the Latin square rules.

Now we explore a Constraint Programming model using a Partial Latin Square example. Consider a 10×10 grid, partially filled, where we aim to fill in the empty cells.

Variables and Domains In CP, variables are unchangeable once assigned, and their domains can only be reduced and not expanded. For our 10×10 PLS, each cell in the grid is a variable with a domain of integers from 1 to 10. We represent this grid as a two-dimensional array:

$$cell[i][j] \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \quad 1 \leq i, j \leq 10$$

Pre-filled cells in the PLS are assigned fixed values from their domains.

Constraints The primary constraint in a Latin Square is that each number appears exactly once in each row and column. In CP, this is implemented using the `alldifferent` constraint.

For rows and columns, the constraints are:

$$\texttt{alldifferent}(c[i][:]) \quad 1 \leq i \leq 10$$

$$\texttt{alldifferent}(c[:][j]) \quad 1 \leq j \leq 10$$

Now, our model is complete. Unlike Sudoku, a Latin Square does not have the subgrid uniqueness constraint. So, our model is simpler but still ensures each row and column contains unique values from 1 to 10.

2.2.2 Declaration of Constraints

Each constraint type is associated with a unique declaration algorithm. Upon the creation of the constraint, the algorithm examines the domain of the variables involved in the constraint with the goal of narrowing down the domains of other variables and removing certain options. Once all relevant variables are determined or the constraint is unable to infer further, regardless of subsequent assignments, the constraint becomes deactivated. An example is shown in Section 2.2.4.

2.2.3 Propagation of Constraints

In Constraint Programming, each constraint is subscribed to a specific event on some variables.

1. A variable getting a specific value assigned.
2. Any change in the variable's domain.
3. Change in the domain's bound.

This is designed based on what can actually be useful in this specific constraint. For instance, consider the `lessOrEqual` constraint. Its job is to make sure a variable x is less than or equal to another variable y. Here, if just one option in x or y's domain is removed, it does not really affect the other, unless that option is at the bounds of the domain. That is why `lessOrEqual` only subscribes to events with changing bounds of x or y.

When this event is triggered, the constraint joins a propagation queue for further action. During the propagation, constraints in the queue will be called in turn with information

about the event that was triggered. This information might lead to more domain changes, which then might trigger other constraints to act and alter other domains, and so on.

2.2.4 Understanding the Filtering Algorithm with an Example

Now we explore how a filtering algorithm works using the `lessOrEqual` constraint. This constraint is the relationship between x and y as follow:

$$x \leq y$$

Here, we are mainly interested in when the highest or lowest values of x or y change. Here is the filtering algorithm for the `lessOrEqual` constraint:

Algorithm 1 Filtering for lessOrEqual Constraint

```
function FILTER(x, y)
    Remove any value from x that is greater than y's maximum.
    Remove any value from y that is less than x's minimum.
    if x's maximum is less than or equal to y's minimum then
        Deactivate the constraint.
    end if
end function
```

In General, when dealing with multiple constraints, each constraint is considered both individually and as part of a larger system of constraints. The process is iterative, which means the solver repeatedly evaluates each constraint to refine the possible values in the domains of the involved variables.

While the CP framework (which is not a learning method) is beneficial in a variety of problems, it does not converge to the true marginal distributions. CP model provides us with 0 and 1 values indicating if a variable can take a value or not. Therefore, we do not have any information on the distribution of the variables.

2.2.5 Belief Propagation

When the solver has applied all constraints, it reaches a point where no further deductions can be made. This stopping point occurs either because the solver has found the only solution or because there are multiple possible solutions. In situations with multiple solutions, identifying the most likely values becomes useful. Solution counting is a method to achieve this [5]. It involves listing all solutions for each constraint and calculating the frequency of each value

in these solutions to estimate its marginal probability. However, finding all solutions can be a challenging task [5].

Integrating Belief Propagation (BP) into constraint programming allows for an approximation of these marginal probabilities through a message-passing algorithm [5]. This approach simplifies solution counting by focusing on individual constraints and determining the frequency of solutions related to specific variable-value combinations.

Consider the `lessOrEqual` constraint as an example. The following algorithm (2) demonstrates how to calculate initial messages for a variable x. A similar method can be applied for variable y.

Algorithm 2 Solution counting algorithm for `lessOrEqual` constraint, initializing *beliefsX* to zero.

```
function FILTER(x, y)
    for x_i in x.DOMAIN do
        for y_i in y.DOMAIN do
            if x_i ≤ y_i then
                beliefsX[i] ← beliefsX[i] + beliefsY[i]
            end if
        end for
    end for
end function
```

In this algorithm, we start by setting the belief values for all possible values of variable x to zero. For each value of x, we then sum up the beliefs of all the values of y that are equal to or less than the current value of x. This process aligns with satisfying the given constraint.

The total number of solutions derived is conveyed through messages, and the message-passing algorithm estimates the marginal probabilities for each variable-value pair by combining these messages. The following equations, derived from reference [5], are used in Belief Propagation. Here, $N(x)$ denotes the set of constraints involving variable x, and $N(c)$ indicates the set of variables linked to constraint c. Let the notation $\mu_{x\to c}$ represent the message from variable x to constraint c and $\mu_{c\to x}$ signify the message from constraint c to variable x. $D(x)$ is the domain of all possible values for variable x:

$$\mu_{x\to c}(v) = \prod_{c'\in N(x)\setminus c} \mu_{c'\to x}(v) \qquad \forall v \in D(x) \tag{2.9}$$

$$\mu_{c\to x}(v) = \sum_{V:V[x]=v} f_c(V) \prod_{x'\in N(c)\setminus x} \mu_{x'\to c}(V[x']) \qquad \forall v \in D(x) \tag{2.10}$$

The algorithm involves two types of messages: from variables to constraints, and from con-

straints to variables. The message from variable x to constraint c about value v is calculated by multiplying the messages received from all other constraints associated with x for that value. The message from constraint c to variable x for value v involves considering all potential assignments where x equals v. For each assignment, we calculate the sum of the product of messages from each variable in the assignment, and then a constraint-specific function f_c checks if the assignment satisfies the constraint. This function helps filter and sum only those assignments that meet the constraint.

After the initial constraint propagation, we perform a set number of BP iterations to approximate the marginal probabilities. We use the following formula:

$$\theta_x(v) = \prod_{c \in N(x)} \mu_{c \to x}(v) \quad \forall v \in D(x) \tag{2.11}$$

This calculates the marginal probabilities of variable x for value v by multiplying the messages from all constraints related to v.

The aim here is not to find a specific set of solutions but to determine the probabilities of different values leading to a valid solution.

2.2.6 CP-BP with an Example

Constraint Programming with Belief Propagation (CP-BP) framework introduced by Pesant [5]. As discussed, this framework offers a more sophisticated approach than traditional constraint propagation, allowing it to capture the complex relationships between variables and ultimately aims to estimate true marginal distributions.

We illustrate CP-BP in the following example taken from that reference [5].

Example: Let a, b, c, and d be variables with identical domains $\{1, 2, 3, 4\}$. Suppose they are subject to the following constraints:

$$\texttt{AllDifferent}(a, b, c),$$
$$\texttt{Sum}(a, b, c, d, 7),$$
$$c \leq d.$$

This CSP has two solutions: $(2, 3, 1, 1)$ and $(3, 2, 1, 1)$. Examining variable a, we see that $a = 2$ and $a = 3$ each appear only in one solution, while $a = 1$ and $a = 4$ are never valid assignments. Table 2.1a shows the true marginal probabilities for each variable. CP-BP, starting from uniform distributions, iteratively refines its estimates. As seen in Table 2.1,

these estimates converge towards the true marginals after a few iterations.

	1	2	3	4
θ_a	0	.5	.5	0
θ_b	0	.5	.5	0
θ_c	1	0	0	0
θ_d	1	0	0	0

(a) true marginals

	1	2	3	4
$\hat{\theta}_a$	.25	.25	.25	.25
$\hat{\theta}_b$	.25	.25	.25	.25
$\hat{\theta}_c$	.25	.25	.25	.25
$\hat{\theta}_d$	.25	.25	.25	.25

(b) initial marginals

	1	2	3	4
$\hat{\theta}_a$	.50	.30	.15	.05
$\hat{\theta}_b$	.50	.30	.15	.05
$\hat{\theta}_c$	.62	.28	.09	.01
$\hat{\theta}_d$	.29	.34	.26	.11

(c) 1st iteration

	1	2	3	4
$\hat{\theta}_a$	.01	.52	.46	.01
$\hat{\theta}_b$	.01	.52	.46	.01
$\hat{\theta}_c$	.98	.02	.00	.00
$\hat{\theta}_d$	.90	.10	.00	.00

(d) 10th iteration

Table 2.1 Marginals: (a) True marginals, (b) Initial marginals, (c) Marginals after 1 iteration, (d) Marginals after 10 iterations. This figure is depicted from the reference [5].

The approximate marginals computed by CP-BP have primarily been used to inform branching heuristics when solving combinatorial problems [20]. Next, in Chapter 3, we will provide an overview of related works in tackling structured prediction problems.

CHAPTER 3 LITERATURE REVIEW

In Chapter 2 we covered the basics of training a Multilayer Perceptron and how CP and CP-BP work. Then, we went over a hybrid approach to combine Constraint Programming with an MLP. Now, this section delves into different approaches within the field to solve structured prediction tasks. Structured prediction represents complex challenges, requiring strategies that can effectively model and solve problems that involve interdependent variables. We explore and outline the key contributions of various methods, organizing them into two main categories. The first category, *Pure Machine Learning Approaches*, focuses on methods solely based on machine learning techniques. The second category, *Hybrid Approaches*, examines methods that blend machine learning with additional strategies such as rule-based systems. Our proposed methodology, which we detail in Chapter 4, falls within this category. Here, our goal is to present these methodologies clearly and systematically, highlighting their contributions to the field.

3.1 Pure Machine Learning Approaches

In this section, we go over some of the methods that use machine learning approaches for structure prediction tasks.

Structured Prediction Energy Networks (SPENs) [21], introduced by Belanger and McCallum (2016), offer a flexible Machine Learning framework for handling tasks where the predicted output has internal dependencies. SPENs use a deep neural network to learn an energy function that assigns scores to potential output structures. The optimal output configuration is found by minimizing this energy function. While this approach allows SPENs to model complex relationships without strict modeling assumptions, it has limitations. Finding the optimal output structure can be computationally expensive, and the original training method might limit accuracy compared to later end-to-end techniques [22]. Partial Latin Square problems can have multiple feasible solutions. However, our datasets reflect only a single solution per instance which makes it difficult to train an energy function effectively.

Nandwani et al. [23] (2021) tackle a fundamental issue within combinatorial problem solving using neural networks: the presence of multiple equally valid solutions. When the goal is to find any feasible solution (like in Sudoku, where many correct completions exist), traditional training methods can struggle. Their paper, *Neural Learning of One-of-Many Solutions for Combinatorial Problems in Structured Output Spaces*, introduces a framework to enhance

training in this scenario. This framework augments existing prediction networks with a selection module that intelligently chooses which solutions to focus on during each training iteration to improve the efficiency of learning. They employ Reinforcement Learning (RL) to train this selection module. A potential limitation is the increased complexity introduced by this additional selection module. Additionally, while the concepts around finding one of many valid solutions might be adaptable to Partial Latin Square problems, the method does not inherently guarantee the satisfaction of hard constraints. While it seems possible to adopt this method to solve PLS, it would be expensive since they use RL. Therefore we do not use it as a baseline.

Nandwani et al. (2019) [24] present a method for embedding constraints directly into the training process of deep neural networks. Their approach introduces a primal-dual formulation where constraints are transformed into soft penalties that guide the network's learning. This allows models to produce outputs that are more likely to satisfy the desired constraints. However, a potential limitation is that this method requires the constraints to be expressed in a differentiable form. Applicability to the PLS problem would likely be challenging, as PLS constraints (unique values in rows/columns) are inherently combinatorial and not easily translated into smooth, differentiable functions. In their experiments, they demonstrate the technique on tasks such as semantic role labeling (where grammatical structure imposes constraints) and image segmentation (where spatial relationships can be expressed as constraints).

The predict-and-optimize framework, explored by Mandi et al. (2020) [25] and Mulamba et al. (2020) [26], tackles complex combinatorial optimization problems often found in real-world decision-making. This framework trains a predictive model to estimate uncertain problem parameters (such as task durations) from historical data. These predictions then guide an optimization process to generate solutions. One key challenge lies in the potential disconnect between the predictive model's accuracy and the quality of the final optimization-derived solutions. Additionally, this would be difficult to apply to the Partial Latin Square problem, since PLS relies on hard constraints, training a predictive model to anticipate these while ensuring overall solution feasibility would be complex.

3.2 Hybrid Approaches

In this section, we go over methods that combine machine learning approaches with other methods for structure prediction tasks. A growing body of research explores the integration of model-based combinatorial solvers with machine learning techniques [27]. One notable approach is the work of Hu et al. (2021) [28], who propose an iterative distillation method

to embed logic rules directly into the weights of neural networks. This method can enhance the performance of various network architectures, including Convolutional Neural Networks and Recurrent Neural Networks. Icarte et al. (2019) [29], on the other hand, focus on training binarized Deep Neural Networks, where weights and activations are limited to ± 1. They formulate these networks as Constraint Programming and Mixed Integer Programming models, optimizing them for improved generalization capabilities and achieving promising results in few-shot learning scenarios. Here we divide these approaches into *Tightly Integrated* and *Loss-Guided* approaches.

3.2.1 Tightly Integrated Approaches

Neuro-symbolic methods, as discussed by De Raedt et al. (2020) [30], aim to solve issues by incorporating knowledge about the specific domain into the training phase. For example, they use the fact that in a Sudoku puzzle, each row, column, and 3×3 square must contain unique elements. The goal is to increase the probability that the neural network's outputs will satisfy the domain's rules. However, this likelihood is usually difficult to calculate, limiting its practical application. Despite this complexity, Xu et al. (2018) [31] and Ahmed et al. (2022) [32] show that it is feasible to manage for several practical applications. However, it still remains challenging for many areas of interest.

DeepProbLog [33], introduced by Manhaeve et al. (2021), represents an advancement in the integration of neural networks with probabilistic logic programming. This framework combines the computational power of neural networks with the structured reasoning capabilities of logic programming. This gives it the capability to support a wide range of functionalities. This method is a programming language with well-defined semantics that supports machine learning. Also, it unifies symbolic and subsymbolic reasoning with logical and probabilistic modeling. DeepProbLog extends the ProbLog language [34] by introducing neural predicates, therefore allowing for the optimization of parameters within both probabilistic logic programs and neural networks through standard gradient-based methods. This integration not only enhances model performance in tasks requiring both symbolic and subsymbolic reasoning but also significantly improves accuracy, data efficiency, and generalization compared to traditional neural network models. Despite its innovative approach, DeepProbLog has certain limitations, notably in its reliance on exact probabilistic inference. This reliance could lead to increased computational costs as the complexity of the programs escalates. Therefore, it would be costly to apply it to our case study.

NeurASP by Yang et al. (2020) [35] is an extension of ASP (answer set programming) with neural networks by considering the neural network output as a probability distribution

over the fundamental units of information (atomic facts) that the ASP program (a set of logical rules and facts that describe a problem and its potential solutions.) uses to represent knowledge and solve problems. The output of the neural network goes through the ASP program. They are able to backpropagate from the output ASP program and solve the problem using gradient descent. Similar to semantic loss [31], the model needs all the possible combinations of the solutions to be able to calculate the ASP coefficient required to be multiplied by the gradients of the neural networks. Therefore, similar to many other methods, its applicability remains limited to small problems. While this method is limited, they show that their method is much faster than DeepProbLog [33] in some problems like shortest path. Again this method is not applicable to the PLS problem due to computational cost.

In *Semantic Probabilistic Layers for Neuro-Symbolic Learning*, Ahmed et al. (2022) [36] present a Semantic Probabilistic Layer (SPL) that enhances neural networks for structured-output prediction. This layer ensures neural network predictions align with certain rules or symbolic constraints. The SPL is adaptable, allowing for detailed modeling and logical reasoning, while still being learnable through standard training methods. The researchers show that their method outperforms others in complex tasks like multi-label classification and preference learning. To adopt this method to PLS, the core challenge is translating the strict combinatorial rules of PLS into constraints usable by the SPL inference mechanism which is non-trivial. That means when the neural network proposes potential values for the empty cell in PLS, the SPL's inference mechanism needs to repeatedly check if inserting each proposed value violates the existing PLS constraints and if there is a potential path to a full solution. Therefore, this method does not work for PLS problem and we do not consider it.

SATNet (2019) [37] also presents a hybrid approach, directly integrating a differentiable maximum satisfiability solver as a layer within a deep neural network. This allows the network to learn within a framework of logical constraints. Potential limitations include the expressiveness of constraints that this model can naturally represent and the challenges associated with ensuring the differentiability of the embedded solver.

3.2.2 Loss-Guided Approaches

Xu et al. (2018) [31], in their paper *A Semantic Loss Function for Deep Learning with Symbolic Knowledge*, propose a hybrid approach to bridge the gap between deep learning models and symbolic knowledge representations. At the core of their method is a novel loss function designed to penalize neural network outputs that violate logical constraints expressed as symbolic rules. This integration of explicit rules differentiates the method from pure machine learning approaches. While the technique shows promise for structured

output problems like Partial Latin Squares, where satisfying strict constraints is essential, a significant limitation lies in its computational cost. The process of repeatedly checking the logical consistency of the network's outputs against the symbolic knowledge base during training can become prohibitively expensive. A key challenge is that the loss function must implicitly consider all possible answers – both valid and invalid – to guide the network's learning. In cases like an empty PLS, where many possibilities exist, the task of checking every output against the rules becomes much more computationally demanding. This can make the method less practical for large-scale problems with large solution spaces.

In the study *Semantic Strengthening of Neuro-Symbolic Learning* by Ahmed et al. (2023) [38], a new approach to neuro-symbolic integration is presented, addressing the computational challenges of combining neural network predictions with symbolic knowledge. The researchers introduce an iterative method that improves the approximation of symbolic constraints by calculating the mutual information between constraint pairs, conditioned on the network's features. This technique is efficient and leads to superior performance over traditional methods. The effectiveness of this approach is demonstrated through tasks like Warcraft pathfinding, perfect matching computations, and Sudoku solving, outperforming baselines without sacrificing the computation.

Lee et al. (2019) [4] propose a hybrid method to enforce constraints on the predictions of a pre-trained deep neural network. Their approach involves an iterative process where a loss function incorporates the degree of constraint violation, and weights are adjusted using gradient descent. Importantly, their method accommodates non-differentiable constraints, enabling the use of complex combinatorial rules. However, the success may depend on the pre-trained network's quality, and the iterative process could introduce computational cost.

Yang et al. (2022) [39] propose a method for integrating logical constraints into neural networks using Straight-Through Estimators (STEs). This approach addresses the challenge in neuro-symbolic AI, where combining neural learning with symbolic reasoning has proven difficult. By systematically encoding constraints as a loss function and leveraging STEs for efficient gradient computation, their method effectively steers the network's weights toward satisfying the constraints. They show that their experiments run faster than DeepProbLog [33] and NeurASP [35] while there is a slight performance drop.

Chen et al (2020). [40] design an end-to-end framework called Deep reasoning networks, DRNets. combining deep learning and CP to solve pattern de-mixing problems such as the overlapping Visual Sudoku problem. The framework was improved by Bai et al. (2021) [41] with the use of curriculum learning and restarts. These approaches are expensive and do not translate directly to our problem.

Lafleur et al. (2022) [42] introduce a reinforcement learning technique for music generation that enables them to modify a pre-trained RNN to satisfy arbitrary constraints. Their key innovation lies in using the same CP-BP framework to express constraints and design a reward function. This reward combines the RNN's output probabilities, how many constraints are violated, and marginals. Their results demonstrate that this method successfully guides the model to produce music that adheres to the constraints while retaining the characteristics learned by the original RNN.

Silvestri et al. (2021) [43] investigate the integration of domain knowledge within neural networks for constrained problems. To provide a controlled and measurable experimental environment, they utilize the Partial Latin Square problem as their primary benchmark. With its well-defined rules, PLS offers a clear context to assess the impact of knowledge injection. Their core methodology begins with a basic neural network trained for the task of predicting cell values in a partially filled PLS instance. They then systematically explore multiple knowledge injection techniques. These include modifying the loss function to penalize constraint violations. Crucially, they evaluate the performance of models both with and without knowledge injection, carefully measuring solution accuracy, feasibility ratio, and the computational cost associated with training. However, their integration of a CP solver into the loss function necessitates pre-calculating the solver's outputs. Our work builds upon these concepts, leveraging CP-BP information (instead of CP) within the loss functions themselves and redefining the loss calculation methodology. Further details on our approach will be provided in Chapter 4. A key limitation of the previous work is its bias towards specific solutions. We address this limitation by introducing a label-free training methodology.

This literature review has surveyed diverse strategies for solving structured prediction tasks. We have categorized them into pure Machine Learning approaches and hybrid methods that integrate Machine Learning with other techniques. The key takeaway is the importance of adapting the approach to the specific problem structure. Hybrid methods show great promise for combinatorial problems, but the challenge lies in balancing the computational cost of their reasoning components with the benefits they provide in guiding the learning process. Our work builds upon the strengths of a hybrid approach by integrating it with CP-BP which has more information for training the network. Next, in Chapter 4 we delve into the details of our methodology.

CHAPTER 4 METHODOLOGY

In this chapter, we explore our hybrid approach to solving challenging combinatorial problems that leverage the strengths of both neural networks and constraint programming. Specifically, we focus on the task of Partial Latin Square Completion. Our methodology aims to train neural networks to internalize the combinatorial structure of PLS. We use traditional neural network training by integrating feedback from a CP framework, specifically using Constraint Programming with Belief Propagation. CP-BP provides richer information than traditional CP solvers by calculating marginal probabilities, indicating the likelihood of each potential assignment being consistent with the combinatorial constraints. Our approach incorporates this probabilistic information directly into the network's loss function, guiding its learning process toward predictions that respect the structural constraints of the PLS problem. We will see the details of our model, masking, and the unique way we calculate the loss (applying cell-wise softmax) later on in this chapter. Here we start with the problem definition and our motivations.

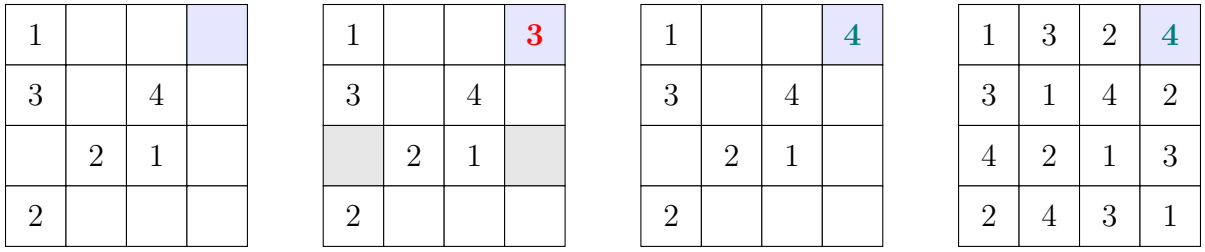
4.1 Problem Definition and Motivation

4.1.1 Partial Latin Square

As explained before, a Partial Latin Square is a mathematical concept where a square grid of $n \times n$ cells is partially filled with n numbers, with each number appearing at most once in each row and column. The objective is to fill one more square such that the PLS remains feasible, that is we should be able to fill the entire PLS according to constraints.

We use PLS to evaluate our methodology. Though simple in formulation, PLS is $\mathcal{NP}$-hard (unless starting from an empty square). The problem has applications in areas like optical fiber routing and serves as a foundation for more complex problems such as timetabling [43]. Using a well-understood constrained problem like PLS provides a solid foundation and reliable domain knowledge for our study.

Figure 4.1 illustrates such an assignment. Figure 4.1a shows an instance of PLS and we want to add one more value to the PLS in the upper right corner. As we can see in Figure 4.1b, adding a value of 3 respect the local constraints as we do not have this value in the first row and last column. However, this assignment is unfeasible because we can not fill the third row (gray cells) with the value of 3. That is because we have 3 in the first and last column. Figure 4.1c shows a feasible assignment because from this assignment we can continue and



(a) PLS of size 4 (b) Unfeasible assign. (c) Feasible assignment (d) Filled PLS

Figure 4.1 Illustration of PLS assignments with local constraints and feasibility considerations. Subfigure (a) displays an initial PLS instance, indicating the intention to assign an additional value in the upper right corner. In subfigure (b), the assignment of the value 3 meets local constraints as it does not duplicate in the first row and last column; however, the assignment is unfeasible due to the inability to allocate the value 3 to the third row (marked in gray) without violating columns of one and three constraints. Subfigure (c) shows a feasible assignment, setting the stage for a complete solution by allowing the continuation of the assignment process. Finally, subfigure (d) presents the complete cell configuration following the feasible assignment outlined in subfigure (c).

fill all the other cells. The complete cells are shown in Figure 4.1d.

As we will see in Chapter 5, the inherent combinatorial structure of this task poses a significant challenge for traditional MLP approaches.

4.1.2 Motivation for Hybrid Approach

Constraint programming excels at declaratively expressing structural relationships and constraints that define valid solutions within a problem domain. However, CP is not inherently a learning method; it relies on search or inference techniques to discover solutions conforming to these constraints. This limits its adaptability as it does not learn to output a valid solution.

Multilayer Perceptrons, on the other hand, are powerful function approximators with a demonstrated capacity to learn complex mappings from data. They extract patterns and regularities through iterative updates driven by a loss function. However, when faced with highly structured tasks where outputs must follow strict combinatorial rules, standard MLPs often struggle to capture these underlying constraints within their learned representations.

The motivation for combining CP and MLP comes from a desire to leverage their complementary abilities: Using CP models to guide the learning process of the MLP. Then, MLPs internalize combinatorial constraints, improving their ability to generate predictions (solu-

tions) that are more accurate by considering the constraints.

In this work, we try to explore a neuro-symbolic combination where neural networks are trained to capture combinatorial structures for generating predictions consistent with that structure. Our constrained combinatorial search space is defined over variables with finite domains, restricting feasible combinations of variable values. Following Silvestri et al. [43], we consider the PLS problem and training instances of $\langle x, y \rangle$ consisting of x, a partial instantiation (variable-value assignments), and, y, an individual assignment extending x while maintaining compatibility with a full valid instantiation.

The learning goal is for the neural network to predict such consistent assignments. Iterative applications of this model could generate complete instantiations. Here, we focus on its performance with single assignment additions.

A fully connected neural network serves as the predictive model. During training, it receives feedback through its loss function, which integrates information from both the true label and a constraint programming model. This allows the network to learn within the context of a downstream combinatorial task.

The CP model provides feedback based on the provided partial instantiation x. The use of CP offers key advantages:

- Declarative: CP focuses on expressing structure, not solution methodology.
- Modular: Complex structures are built by composing basic constraint primitives.

CP traditionally provides two types of information regarding constraint satisfaction under a given partial instantiation:

- Constraint Violations: Binary indications of whether constraints are violated.
- Unsupported Domain Values: Binary indications of variable values that cannot be part of any complete solution.

In both cases, the feedback is binary. Some CP solvers might quantify by how much some constraints are violated [44] but this is not always the case.

The CP-BP framework, on the other hand, offers some advantages over traditional CP:

- Finer-grained feedback: Marginal distributions over domains enable discrimination between values, indicating the probability of assignment consistency.

- Natural Loss Integration: As a probability mass function, this information can be naturally incorporated into the network's loss function.

Silvestri et al. [43] incorporates traditional CP feedback into neural network loss functions. We extend their work in three key ways:

1. Enhanced CP Integration: We explore better incorporation of the CP model within our MLP network.
2. Harnessing CP-BP: We train models leveraging the richer probabilistic information provided by CP-BP.
3. Label-Free Training: We investigate training the network without explicit labels, guided solely by feedback from the CP and CP-BP models.

In the next section, we delve into how to directly utilize CP-BP marginal probabilities within the network's loss function.

4.2 Loss Functions and Feedback

4.2.1 CP Based

We model the MLP as a function f, whose output represents the learned probability distribution over potential remaining assignments. For a given partial instantiation x, $f_k(x)$ denotes the probability of selecting the k^{th} assignment.

Drawing inspiration from semantic-based regularization [31] and Silvestri et al [43], we convert combinatorial constraints into additional components within the loss function.

Let $C_k(x)$, the output of CP solver, equal 0 if the k^{th} assignment is unsupported (filtered out) by the solver given partial instantiation x, and 1 otherwise. To guide the learning process, there are several candidate terms to serve as a penalty within the loss function as in Silvestri [43]:

Unsupported Assignment:

$$neg(x) = \sum_k ((1 - C_k(x)) \cdot f_k(x))$$

Binary Cross Entropy (BCE):

$$BCE(x) = \sum_k (C_k(x) \cdot \log(f_k(x)) + (1 - C_k(x)) \cdot \log(1 - f_k(x)))$$

L2 Loss (Mean Squared Error):

$$L_2(x) = \sum_k (C_k(x) - f_k(x))^2 \tag{4.1}$$

For our CP experiments, we primarily utilize the L_2 loss that was working better in Silvestri [43], while also exploring the L_1 loss:

$$L_1(x) = \sum_k |C_k(x) - f_k(x)| \tag{4.2}$$

The final loss function combines one of the above terms, $t(x)$, with standard categorical cross-entropy:

$$L(x, y) = -\sum_k y_k \log(f_k(x)) + \lambda \cdot t(x) \tag{4.3}$$

Here, λ weights the influence of the constraint-based loss component, and y_k is the label.

We also do not use the second penalty, BCE, as the problem is not binary and there are multiple classes per square of PLS. We believe BCE is not relevant here as BCE loss is meant to compare two sets of probabilities and $C(x)$ is not a probability distribution. Next, we explain penalty terms for our CP-BP integrated models.

4.2.2 CP-BP Based

The indicator function $C_k(x)$ does not differentiate between assignments that remain feasible after filtering. In contrast, CP-BP marginals provide this useful information. Therefore, by replacing $C_k(x)$ with $\hat{\theta}_k(x)$, representing the marginal probability of the k^{th} assignment as calculated by the CP-BP framework, we can fill this gap. Note that we only need to find C_k and $\hat{\theta}_k$ once to use them in the loss.

Among the previous candidate terms, the first (unsupported assignment penalty) becomes less meaningful where marginals as CP-BP output are not binary. Also, as explained BCE is not a good implementation and we do not use it. Therefore, we retain the L_2 distance and explore the L_1 distance as well. Overall, for our CP-BP experiments, we consider:

L_1 Distance:

$$L_1(x, \hat{\theta}) = \sum_k |\hat{\theta}_k(x) - f_k(x)| \tag{4.4}$$

L_2 Distance:

$$L_2(x, \hat{\theta}) \quad = \quad \sum_k (\hat{\theta}_k(x) - f_k(x))^2 \tag{4.5}$$

Additionally, for our CP-BP experiments, we also consider losses that better compare probability distributions:

Categorical Cross-Entropy (CCE):

$$\text{CCE}(x, \hat{\theta}) = -\sum_k \hat{\theta}_k(x) \log(f_k(x)) \tag{4.6}$$

KL-Divergence:

$$\text{KL}(x, \hat{\theta}) = -\sum_k \hat{\theta}_k(x) \log\left(\frac{f_k(x)}{\hat{\theta}_k(x)}\right) \tag{4.7}$$

The last two losses that use marginals align more naturally with our goal. The CP-BP marginals serve as a reference probability distribution compared to the network's learned probabilities. Both $f(x)$ and $\hat{\theta}(x)$ contain distributions—one for each unassigned variable. This contrasts with $C(x)$, representing a single variable with filtered domain values that do not sum to 1.

Computational Efficiency: Training time for our approach sits between methods wholly integrated with the neural network (benefiting from GPU acceleration) and those requiring NP-hard symbolic solver calls during backpropagation. We rely on a few BP iterations, using specialized weighted counting algorithms with polynomial time complexity.

4.3 Modeling Details

Our methodology aims to train fully connected neural networks to learn and respect combinatorial structures within their predictions. We define a constrained combinatorial space using variables with finite domains and constraints that limit valid combinations of values.

We train on samples $\langle x, y \rangle$, where x is a partial instantiation and y is an extending assignment that maintains the potential for a complete, valid solution. The neural network's task is to assign a new value such that we can continue and fill the grid afterward.

Next, we explain the differences between our model from Silvestri [43].

4.3.1 Input, Output, and Labels

Our method modifies the input format to a two-dimensional binary matrix of size $n^2 \times n$, where n^2 is the number of squares. An example for $n = 2$ (PLS of size 2) would be:

$$\begin{array}{|c|c|}\hline 2 & 1 \\ \hline & \\ \hline \end{array} \quad \Rightarrow \quad \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

This matrix on the left represents the 2×2 grid. In the matrix on the right:

- The first row represents the value 2. This is indicated by the 1 in the second column.
- The second row represents the number 1, as shown by the 1 in the first column.
- The last two rows represent an empty cell, indicated by all elements being 0.

The output in our approach is also a two-dimensional binary matrix of the same size as the input. However, before determining the next assignment (through argmax), a softmax operation is applied over each cell in this output matrix to provide a probability distribution for each square. We find the model prediction by finding the argmax over this output.

The label matches the input in shape containing all zeros except for a single 1 value. A label for the previous example could be:

$$\begin{array}{|c|c|}\hline & \\ \hline & 2 \\ \hline \end{array} \quad \Rightarrow \quad \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$$

This approach is in contrast with Silvestri [43] where the inputs, outputs, and labels are n^3 and the softmax is applied globally over n^3.

4.3.2 Neural Network Architecture

Our neural network architecture builds upon the work of Galassi et al. [45] and closely follows the design employed by Silvestri et al. [43]. We utilize a Multilayer Perceptron composed of two hidden layers, each containing 512 neurons activated by the ReLU function.

For optimization, we employ the Adam optimizer [46] with its default hyperparameter settings (see Appendix A). To integrate the CP-BP model and its feedback, we use the Mini CP-BP solver[1], a CP-BP framework implementation, also using its default configuration (5 iterations of Belief Propagation). Appendix B provides additional information on the calculation time of CP-BP marginals.

As explained in Equation 4.3, our loss function comprises two distinct components:

1. Categorical Cross-Entropy
2. Penalty Term

Categorical Cross-Entropy: Calculated between the label and the network's output. The network's outputs are logits that go through the softmax. We implement a unique approach where, instead of a single softmax over the entire PLS grid, we apply the softmax operation individually to each square within the instance as explained in Section 4.3.1.

Penalty Term: A function of both the MLP's output and the information provided by either the CP or CP-BP framework. When we have CP as input we use L_1 (Equation 4.2) and L_2(Equation 4.1), while we use CCE (Equation 4.6) and KL divergence (Equation 4.7) when we have CP-BP as input.

Our approach deviates from the work of Silvestri et al. [43], who apply a single softmax over the entire grid. We believe applying softmax to distinct variables within each cell aligns more closely with the combinatorial structure of the problem, where constraints operate at the level of individual squares. Additionally, this modification ensures that the CP-BP solver's output, which provides probabilities for each value within each cell, directly matches the format of our network's output.

4.3.3 Masking Mechanism

We introduce a masking mechanism in both the loss calculation and the prediction process. Masking selectively focuses the loss on empty cells within the PLS instance. Cells that are already filled have their corresponding loss contributions masked out. This approach aims to concentrate the learning process on the areas of the problem where predictions are actively needed, rather than spreading its focus over already-determined values. This represents a deviation from the approach used by Silvestri [43]. Additionally, during the prediction phase, we applied a similar masking strategy. The cells that were already filled in the input PLS

[1] `https://github.com/PesantGilles/MiniCPBP`

instances were masked. Doing this during prediction ensures that the model's output is from empty cells.

Therefore we have made the following changes:

1. Applied a cell-wise softmax to the output of the network.
2. masked the loss and output predictions.
3. Used CP-BP in the penalty term

Next, we delve into the training and evaluation procedure.

4.4 Training and Evaluation

4.4.1 Training

We train the model using the Adam optimizer. Training continues until convergence is reached. To monitor progress, we evaluate the model's performance using the feasibility ratio used by Silvestri et al. [43]. This ratio is defined as follows:

$$\text{feasibility ratio} = \frac{\text{feasible assignments}}{\text{total assignments}}, \tag{4.8}$$

where the total assignments is the sum of feasible and unfeasible assignments. This feasibility ratio represents an average value across all instances.

For PLS sizes 10 and 12, this evaluation occurs every 10 epochs, while for PLS size 7, evaluation occurs after each epoch. This is because PLS of size 7 usually converges faster.

The evaluation utilizes Google OR-Tools' CP-SAT solver [47] to determine if a solution satisfies all PLS constraints. This approach ensures feasibility but is computationally expensive.

To optimize training efficiency, we employ an early stopping mechanism. If the feasibility ratio does not improve for 10 consecutive evaluations, training terminates. The model with the highest feasibility ratio achieved during training is selected for the testing phase. Note that all the experiments are done with five seeds and we illustrate the mean and standard deviation in all the figures.

4.4.2 Testing

During testing, we report the feasibility ratio per number of filled squares. When testing our models, we calculate a slightly modified version of the feasibility ratio for specific filled

instances. This is expressed as:

$$\text{feasibility ratio}(i) = \frac{\text{feasible assignments (when } i \text{ squares are filled)}}{\text{total assignments (when } i \text{ squares are filled)}}, \tag{4.9}$$

where i is the number of filled squares in PLS. This provides a performance metric based on the number of filled squares. Due to the computational expense of calculating the feasibility ratio (which requires the solver to assess PLS feasibility after each assignment), we evaluate the model on a subset of 5k examples, following the approach of Silvestri et al. [43].

4.4.3 Dataset

We employ the same datasets used in Silvestri et al. [43] for each PLS size. These datasets were generated from sets of 10,000 complete, correct PLS solutions.

Dataset Generation: Each dataset originates from decomposing the full PLS solutions. Each dataset originates from decomposing the full PLS solutions. This decomposition creates sequences of $\langle x, y \rangle$ pairs, where x is the partial state and y is the subsequent cell assignment. Every partial state in the sequence, along with the subsequent cell assignment, forms a single data entry.

Dataset Sizes: The datasets contain 490K, 1M, and 1.44M entries for PLS sizes of n=7, 10, and 12 respectively (where n represents both the number of rows and columns). We allocate 350K, 700K, and 1M entries for training, with an additional 5K entries from each training set reserved for validation purposes.

4.5 Baselines

To evaluate our methodology, we compare our models (with CP and CP-BP input) to a range of baseline methods. In the following subsections, we detail each baseline.

4.5.1 Random

The random baseline establishes a point of comparison. It involves assigning a uniform random value to a randomly selected empty square within the PLS instance. This baseline serves to illustrate the minimum expected feasibility ratio when making assignments without any consideration of the underlying constraints or combinatorial structure of the problem.

4.5.2 Symbolic

The symbolic baseline represents a traditional Constraint Programming approach, enhanced with Belief Propagation. It does not involve a learning process. Instead, the assignment of the next value is determined by utilizing the output of the CP-BP solver. For each empty square within the PLS instance, CP-BP provides a probability distribution over the potential values (e.g., $\{1...7\}$ for a 7×7 PLS). The symbolic baseline selects the value with the highest probability within these distributions. That is taking the argmax over the output matrix (49×7 for a 7×7 PLS).

4.5.3 Multiplayer Perception

The MLP baseline establishes the performance of a neural network without any integration of CP/CP-BP elements. It shares the same training procedure as our model, including the masking of loss and output, along with the modified softmax calculation within the loss function.

The differences between this model with our main model are:

1. No CP/CP-BP: The core distinction lies in the absence of any CP or CP-BP solver during training. The MLP relies solely on the provided labels for feedback.
2. Loss Function: The loss function is a straightforward Cross-Entropy between the MLP's output and the corresponding labels.

4.5.4 Agnostic

The Agnostic baseline closely resembles the MLP baseline but with two key differences:

1. No Masking: Unlike the MLP baseline, loss calculation and prediction in the Agnostic model do not utilize masking.
2. Global Softmax: The Agnostic baseline applies a single softmax operation over the entire PLS grid, rather than individual softmax operations per cell.

This model also was a baseline in Silvesri's work [43]. Adding a penalty term to the penalty of this model results in our next baseline, Silvestri with CP.

4.5.5 Silvestri with CP

This baseline is presented by Silvestri [43]. Key differences compared to our method include:

1. No Masking: Our approach introduces masking for both the loss calculation and the network's output, while the Silvestri [43] method does not employ this technique.
2. Global Softmax: Silvestri et al. [43] apply a single softmax over the entire output of the network. In contrast, we apply individual softmax operations to each square within the PLS instance. Therefore, they employ a one-dimensional binary vector of length n^3 to represent the PLS instance. For example, with a PLS of dimension 2, an input might look like $[1, 0, 0, 1, 0, 1, 1, 0]$.

We additionally explore the use of an L_1 penalty term in the loss function alongside the Silvestri method [43].

4.5.6 Silvestri with CP-BP

This baseline follows the previous baseline, except for incorporating a key modification: the utilization of CP-BP feedback in the penalty term of the loss function. Therefore, instead of relying on the binary (supported/unsupported) feedback from a traditional CP solver, this baseline leverages the richer, probabilistic information provided by CP-BP in the form of marginal distributions.

We also experiment with two additional penalties in the loss function well-suited for comparing probability distributions: Categorical Cross-Entropy and KL-divergence.

Similar to the original Silvestri baseline [43], this variant does not employ masking and applies a single softmax operation over the entire grid within the loss function. Note that this is one of our methods based on Silvestri [43].

CHAPTER 5 EXPERIMENTS

In this section, we empirically evaluate the effectiveness of integrating CP-BP into our neural network training process. Our primary metric, the feasibility ratio (as defined in Equation 4.8), measures the model's ability to generate predictions that consistently adhere to the constraints of the problem domain.

Creating a dataset with labels is challenging. Also we would like our model to be sample-efficient. Therefore, we explore training the model with smaller dataset sizes. Additionally, we explore how the model performs when trained on data with no labels.

We replicate the experiments of Silvestri et al. using their publicly available code[1].

In the following sections, we jump into a series of targeted experiments designed to address the research questions of this thesis which are the impact of data size, problem complexity, and the availability of labeled data.

Section 5.1 focuses on establishing the performance of a Multilayer Perceptron trained without any CP or CP-BP integration. This serves as our MLP baseline and we compare its performance to the random baseline. Additionally, we investigate several factors: the effect of reducing the training dataset to 10% of its original size, the impact of masking both the loss and the MLP's output, and the effect of applying softmax individually to cells within the PLS instance.

Section 5.2 investigates the inherent difficulty of the PLS completion problem as the size of the PLS instance changes. We perform a comprehensive comparison across all baseline methods, including Symbolic, Random, MLP, Agnostic, Silvestri with CP, Silvestri with CP-BP, and our proposed method. This analysis aims to reveal how the performance of different approaches is affected by the increasing complexity of the problem.

Section 5.3 investigates the influence of dataset size on the performance of our proposed method and the established baselines. We analyze how the amount of training data available affects the ability of each method to learn the underlying combinatorial structure of the PLS problems. Generating datasets for problems with complex constraints can be challenging, and labeled data might not always be readily available. This analysis provides insights into the data requirements for the effectiveness of each model.

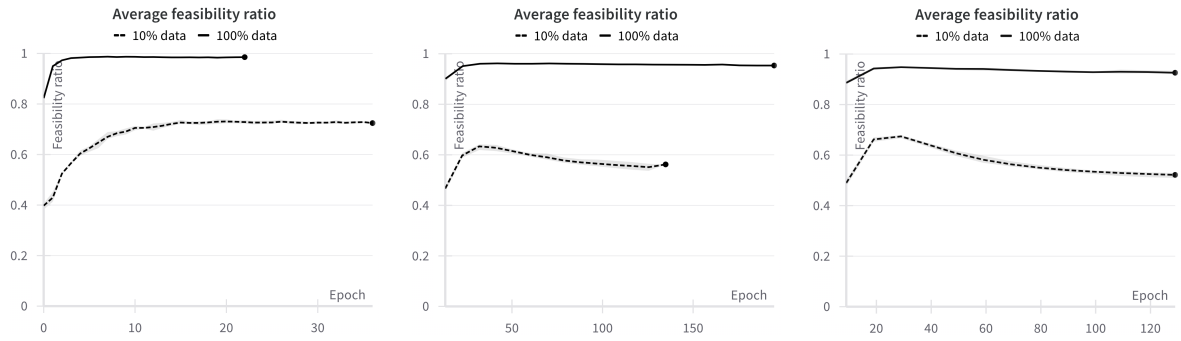
Finally, Section 5.4 explores strategies for training models in the absence of labeled data. We place particular emphasis on the potential benefits of integrating Constraint Programming

[1] `https://github.com/matsilv/knowledge-injection-dnn`

methodologies, especially the CP-BP framework, to guide the learning process even when explicit labels are not available.

5.1 Training MLP with Labels

In this section, we train the MLP model and compare it with the Random model. Figure 5.1 shows this procedure for various PLS sizes. We have also included training the model with 10% of data to investigate the importance of dataset size for training the neural network. Unlike testing where we report the feasibility ratio per filled cell (Equation 4.9), we report the average feasibility ratio for the validation set as described in Equation 4.8.



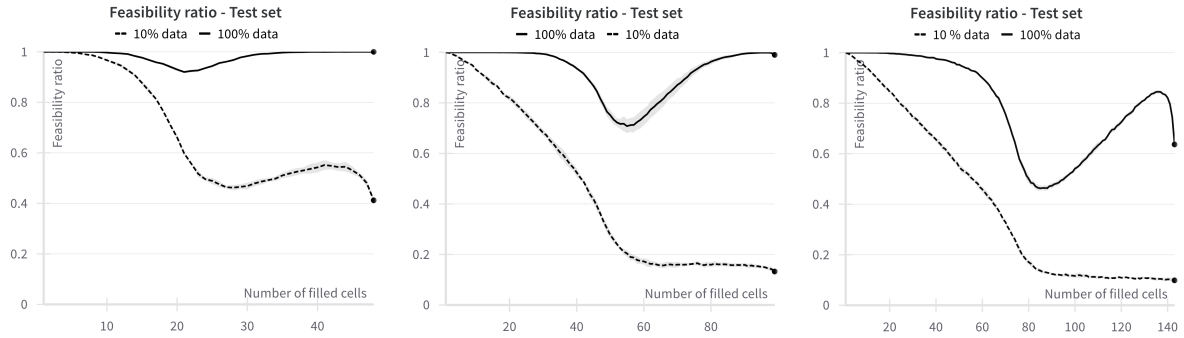
(a) $n = 7$ (b) $n = 10$ (c) $n = 12$

Figure 5.1 Average feasibility ratio of the MLP after every 10 epochs (1 epoch for PLS of size 7). Each curve is the mean of five training sessions, with the surrounding shaded area (which has a very small standard deviation) showing the standard deviation. The best model occurs 100 epochs (or 10 epochs for a PLS of size 7) before training ends, not at the last point.

We see that most models are generally trained for about 20 epochs and after that, the performance deteriorates in the validation set. Since evaluating the models is expensive, we use 5000 examples for each evaluating step. When the model is trained, we use the best-trained one and start testing the model. The performance of three different sizes of PLS is visualized in Figure 5.2.

In the evaluation of the model's performance, the y-axis of the plots represents the feasibility ratio. The x-axis indicates the number of filled cells within the PLS, providing information about the model's behavior under various levels of completion.

Initially, when the grid is empty, the task for the neural network is easy. In this state, any assignment the model makes is going to be correct, resulting in a high feasibility ratio. However, as the the number of filled cells reaches approximately half of the total, the problem



(a) $n = 7$ (b) $n = 10$ (c) $n = 12$

Figure 5.2 Feasibility ratio of the trained MLP with respect to the proportion of filled cells. Each curve is the mean of five training sessions, with the surrounding shaded area showing the standard deviation.

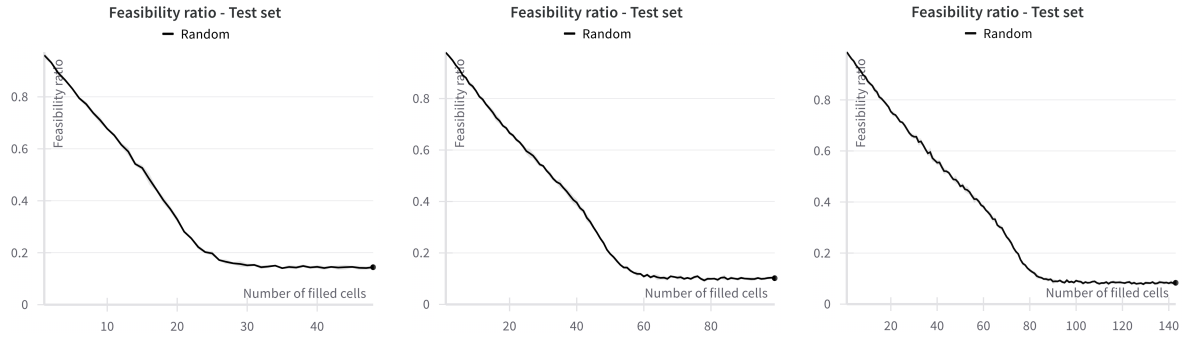
becomes more difficult and the model performance decreases as evidenced by the notable drop in the feasibility ratio.

When the grid is almost full, the problem becomes deterministic. Again this would be easy for neural networks provided enough data. As we can see, in Figure 5.2 parts (c), the feasibility ratio drops at the end, showcasing a lack of data for training. Overall, when we use 10% of data the feasibility ratio decreases especially where we have almost a full grid.

The detailed plots in Figure 5.2—parts (a), (b), and (c)—provide a visual representation of these phenomena for PLS configurations of sizes 7, 10, and 12, respectively. It is observed that the neural network's performance deteriorates as the complexity of the PLS increases, with the most significant drop occurring in the scenarios where the grid is half-filled. Notably, in part (c) of the figure, which illustrates the PLS12, we witness a reduced feasibility ratio even when the grid is nearly full. This indicates a critical aspect of the model's learning requirements which is the necessity for a substantial amount of data to train effectively, especially in more complex configurations.

Overall we observe a decline in performance across all data sizes when a limited subset of data—10% of the total—is used for training. This effect is particularly obvious as the grid is almost full. This highlights the importance of dataset size in training our neural network.

Figure 5.3 provides a visual representation of the feasibility ratios for the Random baseline. The process involves filling an empty cell at random, based on a uniform distribution, rather than selecting the cell with the highest predicted probability by the neural network. The sub-figures illustrate the scenarios for $n = 7$, $n = 10$, and $n = 12$, respectively. The average



(a) $n = 7$ (b) $n = 10$ (c) $n = 12$

Figure 5.3 The feasibility ratio is determined by considering uniform random assignments. Rather than choosing the cell with the highest predicted probability from the neural network output (argmax), we select an empty cell based on a uniform random distribution. The average feasibility ratio from left to right is 0.37, 0.36 and 0.35.

feasibility ratios, observed from left to right, are 0.37, 0.36, and 0.35, which are relatively about the same. We use this baseline as the worst-case scenario.

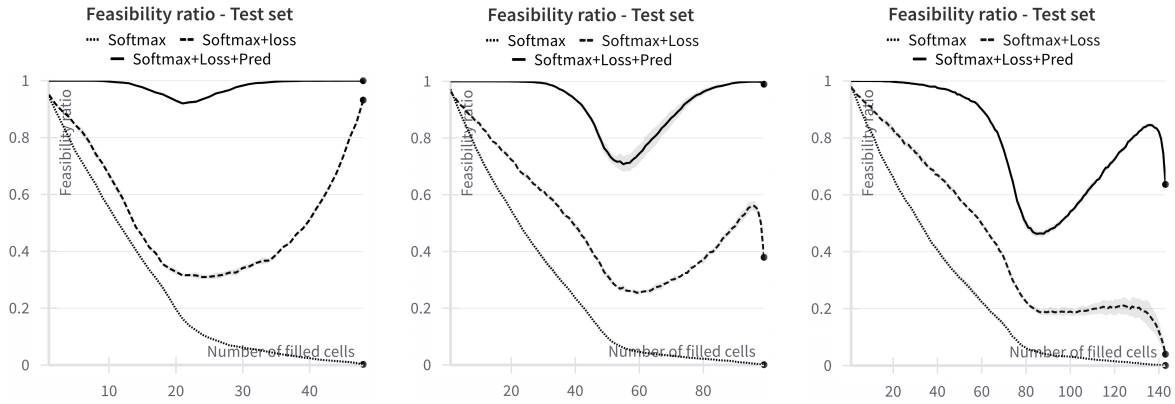
Dataset/Model	**Minimum (Average) Feasibility Ratio**		
	PLS7	**PLS10**	**PLS12**
MLP 10%	0.41 (0.68)	0.13 (0.44)	0.10 (0.41)
MLP 100%	0.92 (0.98)	0.70 (0.92)	0.46 (0.88)
Random Uniform	0.14 (0.37)	0.10 (0.36)	0.08 (0.35)

Table 5.1 This table illustrates the improvements in minimum (and average) feasibility ratios after training the MLP with 10% and 100% of the dataset. The last row indicates the Random uniform model as a reference.

Table 5.1 shows the minimum and average of the feasibility ratio when 100% and 10% data is used during training. It also includes the results for the random uniform model. As we see, all the models work better than the random model.

Next, we explore the effect of each individual component in our MLP baseline. Note that this components are the same in our proposed model. These components are:

- Masking the loss function
- Masking predictions
- Applying the softmax cell-wise in the loss function



(a) $n = 7$ (b) $n = 10$ (c) $n = 12$

Figure 5.4 Feasibility ratio of the trained neural network. The softmax label indicates only softmax is applied cell-wise and there is no masking. Softmax+Loss adds the masking to the loss function and Softmax+Loss+Pred adds the prediction mask. Each curve is the mean of five training sessions, with the surrounding shaded area showing the standard deviation.

Figure 5.4 illustrates these steps in detail. Initially, we apply only the cell-wise softmax (dotted line). Subsequently, we incorporate masking into the loss function (dashed line), and finally, we combine all three components (solid line) to the MLP baseline. As depicted in Figure 5.4, applying solely the cell-wise softmax to the loss function yields suboptimal results. This occurs because the model is trained to predict the highest value for each cell, complicating accurate predictions as the model chooses argmax over the output values. The next step is to add masking the loss function. This is a critical adjustment, as it prevents the calculation of loss for cells that have already been filled. We observe that this masking notably enhances the feasibility ratio. Ultimately, we apply add masking to the output to avoid predictions for cells that are already filled. We show in Table 5.2, that the combination of these three modifications significantly boosts the performance.

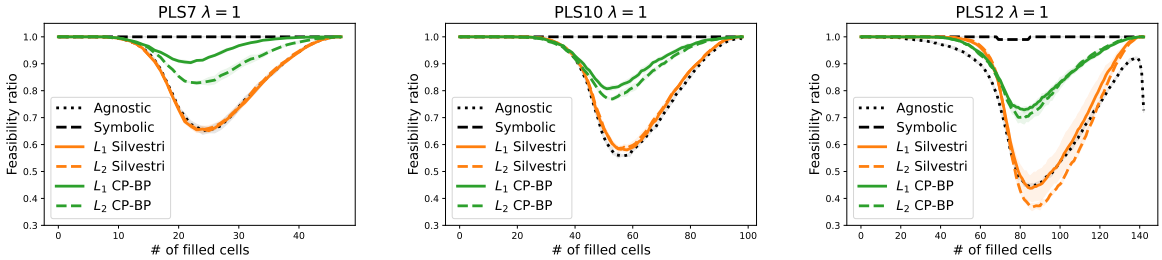
Changes in MLP	**Minimum (Average) Feasibility Ratio**		
	PLS7	**PLS10**	**PLS12**
Softmax	0.00 (0.26)	0.00 (0.26)	0.00 (0.26)
Softmax+Loss	0.33 (0.54)	0.27 (0.51)	0.03 (0.44)
Softmax+Loss+Pred	0.92 (0.98)	0.70 (0.92)	0.46 (0.88)

Table 5.2 This table illustrates the improvements in minimum (and average) feasibility ratios after applying cell-wise softmax and masking the loss function and predictions. The data indicates that modifications have led to a consistent increase in feasibility ratios across different problem size configurations.

Table 5.2 shows the minimum (and average) feasibility ratio in Figure 5.4. Next, we compare our model with baselines.

5.2 Problem Difficulty

To manage the numerous baselines, we will begin by comparing those methods that share these characteristics: no masking in the loss calculation or prediction, and with a single softmax over the entire model output. Figure 5.5 compares four different baselines. The first two are the Agnostic and Symbolic models. The third baseline, Silvestri with CP (orange lines), incorporates either an L_1 or L_2 penalty term in its loss function. The fourth, Silvestri with CP-BP (green lines), augments the Silvestri approach with feedback from the CP-BP framework, also using either an L_1 or L_2 penalty term. We experimented on three instance sizes and three values of the balancing parameter $\lambda = 0.1, 1, 10$ and reported the best value of λ for each instance size.



(a) $n = 7$ (b) $n = 10$ (c) $n = 12$

Figure 5.5 Feasibility ratio of Agnostic, Symbolic, Silvestri, and Silvestri with CP-BP models with respect to the proportion of filled cells. Each curve is the mean of five training sessions, with the surrounding shaded area showing the standard deviation.

Learning methods typically experience the following in terms of instance hardness:

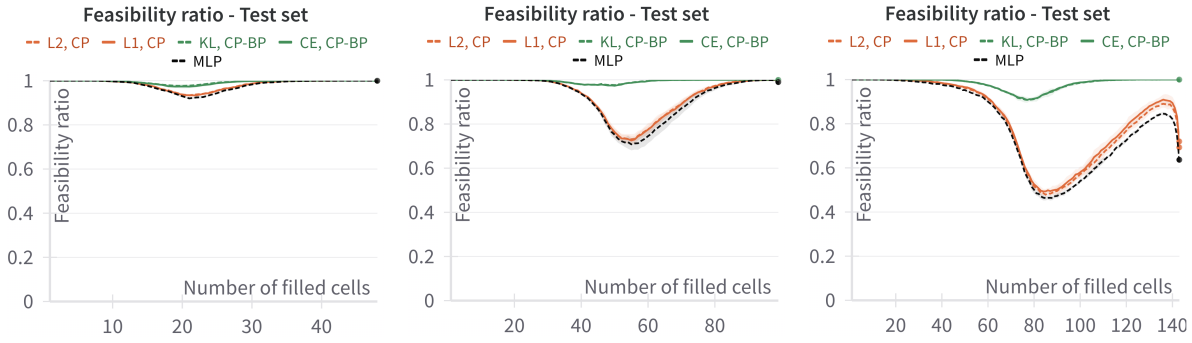
The difficulty of PLS completion varies with the number of filled squares. When the grid is mostly empty, there are many potential solutions, and most assignments are likely to be valid. Conversely, a nearly full grid often has only a single solution path, with most other assignments ruled out by the constraints. The most challenging instances tend to occur when the grid is approximately half-full [43]. The Symbolic baseline, which employs a highly effective tree-search heuristic, achieves near-perfect feasibility ratios ($\geq$ .9999) on the PLS instances we consider. This demonstrates the high quality of the CP-BP marginals we utilize to guide the training of our neural network later on.

Among learning approaches, we observe a very significant improvement of the feasibility ratio

with CP-BP, particularly with the L_1 metric, on all three instance sizes. A small $\lambda = 0.1$, corresponding to less reliance on our marginal term, does not perform as well as the larger values, $\lambda = 1$ and $\lambda = 10$ performing about the same. Since the $\lambda = 1, 10$ were about the same, we have reported $\lambda = 1$ in Figure 5.5. Note that there is a significant drop in the feasibility ratio at the end of the Agnostic curve for the $n = 12$ instances similar to the MLP baseline. Neural networks require enough data to function effectively. As the number of filled cells increases, we hypothesize that the neural network may need proportionally more data points to learn since there are more constraints to learn.

Now we compare the models that share these characteristics: masking in the loss calculation and prediction, and with a cell-wise softmax over the model output.

Figure 5.6 compares three different baselines. The First baseline, the black dashed line, is MLP. The second model, orange lines, is our method with CP input that incorporates either an L1 or L2 penalty term in its loss function. The third, Green lines, is our method with CP-BP, which uses feedback from the CP-BP framework, using either a CE or KL-divergence penalty term. We experimented with various values of $\lambda = 0.1, 1, 10$ and identified $\lambda = 1, 10$ as yielding the optimal result. We can see a significant enhancement of the feasibility ratio across all curves in comparison with Figure 5.6. We can see the comparison of all the models in Table 5.3.



(a) $n = 7$ (b) $n = 10$ (c) $n = 12$

Figure 5.6 Feasibility ratio of our trained models with respect to the proportion of filled cells. Each curve is the mean of five training sessions, with the surrounding shaded area showing the standard deviation. Here the softmax is applied over each cell in the grid. Loss and predictions are masked.

One of the most remarkable outcomes of these modifications (Figure 5.6) was the consistent improvement in the performance of the CP-BP configuration. Unlike other configurations, Our model with CP-BP shows no decline in performance when most of the cells are filled.

Model	Minimum Feasibility Ratio		
	PLS7	**PLS10**	**PLS12**
Ours, CP-BP	**0.98**	**0.97**	**0.91**
Ours, CP	0.93	0.72	0.49
MLP (ours)	0.92	0.70	0.46
Agnostic	0.64	0.55	0.43
Silvestri, CP-BP (ours)	0.90	0.82	0.71
Silvestri, CP	0.64	0.57	0.47
Random	0.14	0.10	0.08
Symbolic	1.00	1.00	0.99

Table 5.3 This table illustrates the minimum feasibility ratios of all the methods. The values indicate that our modifications have led to a consistent increase in feasibility ratios across different problem sizes.

This stability is crucial, as it indicates a higher level of reliability and consistency in the model, particularly in more complex scenarios.

Since we explore multiple loss variations (L1/L2 and CE/KL) for our models, Table 5.3 presents the results of the best-performing configuration. We highlight this performance by comparing the minimum feasibility ratios across the different baselines. A key trend emerges from Table 5.3: as the problem size (PLS) increases, so does the relative improvement in performance. More specifically, we see relative improvements of 34% in the minimum feasibility ratio for PLS size 7, 40% for PLS size 10, and a substantial 44% improvement for PLS size 12 when comparing our CP-BP model to Silvestri with CP's approach. This trend suggests that our modifications to the softmax function and the incorporation of masking techniques are particularly beneficial for larger-scale problems. This scalability is an important aspect of our approach that makes the model remain effective and efficient even as the complexity of the tasks increases.

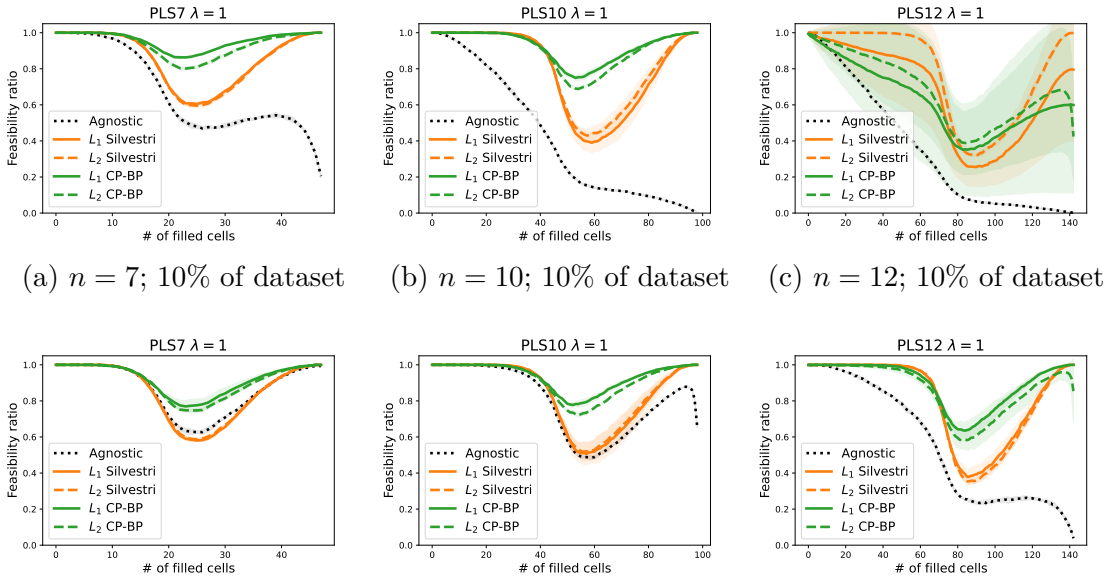
Overall, the integration of a cell-wise softmax function and masking techniques has proven to be highly effective in enhancing the performance of our models. The improvements are consistent across various configurations and are particularly effective in larger problem sizes.

Next, we delve into the effect of dataset size on our models.

5.3 Dataset Size

Here, we investigate the effect of training set size by using only 10% or 50% of the initial size. Similar to before, we begin by comparing those methods that have no masking in the loss calculation or prediction, and with a single softmax over the entire model output. We

fixed λ to 1, which worked well previously. Fig. 5.7 reports our findings. The first two lines in the figure are the Agnostic and Symbolic models. The third baseline is Silvestri with CP(orange lines) and the last one is Silvestri with CP-BP (green lines). The agnostic approach, without any information from the CP model, is very sensitive to the size of the training set and its performance significantly deteriorates as we reduce it. This observation is in line with our hypothesis that fully connected neural networks require more data points to learn when the grid is almost full. The two approaches using CP (Silvestri) and CP-BP (Silvestri with CP-BP input) are not as sensitive to dataset size, with our approach (green lines) still dominating except in some areas for the largest instances ($n = 12$) with only 10% of the original training set, where we see a high standard deviation. This confirms the results of [35] as we also show that introducing feedback from the constraints (either CP or CP-BP) helps compensate for a smaller dataset. The high standard deviation in part C of Figure 5.7 indicates that the models are more sensitive to the size of the dataset when they are about to diverge.



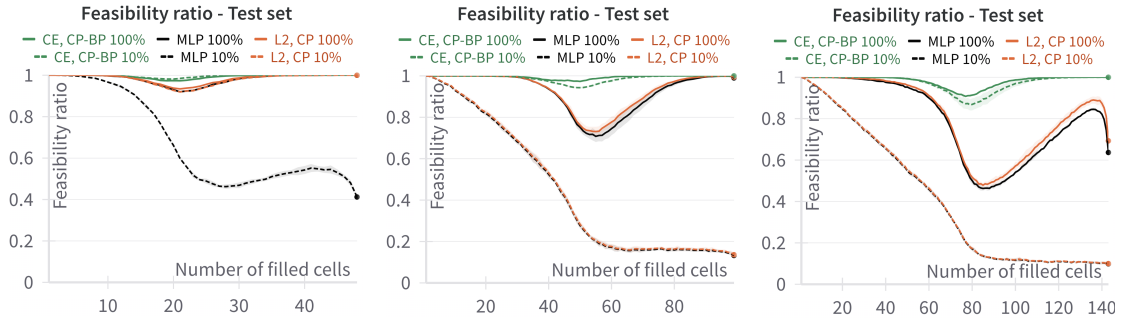
(a) $n = 7$; 10% of dataset (b) $n = 10$; 10% of dataset (c) $n = 12$; 10% of dataset

(d) $n = 7$; 50% of dataset (e) $n = 10$; 50% of dataset (f) $n = 12$; 50% of dataset

Figure 5.7 Feasibility ratio of Agnostic, Silvestri, and Silvestri with CP-BP models with 10% and 50% of data. In these models, there is no masking in the loss function and predictions, and the softmax is applied globally.

Next, we compare models that use masking in loss and predictions and apply cell-wise softmax to their output. We report the best value for λ in our results. That is MLP, our models with CP and with CP-BP. Due to the similar performance achieved by our model when using different penalty terms (L_1/L_2 with CP, and CE/KL with CP-BP), we focus on reporting

the L_2 and CE variants. Again we see that there is a great improvement in the feasibility ratio with increased data. The results can be found in Figure 5.8.



(a) $n = 7$; $\lambda = 1$ (b) $n = 10$; $\lambda = 1$ (c) $n = 12$; $\lambda = 1$

Figure 5.8 Feasibility ratio of our models by training on 10% or 100% of the dataset. The black dashed line is the MLP, the Green lines are our models with CP-BP input and the orange lines are our models with CP output.

Figure 5.8 shows a slight performance decrease in PLS of size 7 when the loss is combined with CP or CP-BP. However, as the problem becomes more difficult, models with CP in their loss deteriorate significantly when the model is trained with 10% od dataset. We see that the models with CP-BP in their penalty perform close to 100% data when fewer data is used.

The minimum feasibility ratio of each curve is shown in Table 5.4. Since we explore multiple loss variations (L_1/L_2) in some models, we report the best-performing configuration.

Model	**Minimum Feasibility Ratio - 100% (10%) Data**		
	PLS7	**PLS10**	**PLS12**
Ours, CP-BP	**0.98 (0.97)**	**0.97 (0.93)**	**0.91 (0.86)**
Ours, CP	0.93 (0.91)	0.72 (0.13)	0.49 (0.10)
MLP (ours)	0.92 (0.41)	0.70 (0.13)	0.46 (0.10)
Agnostic	0.64 (0.19)	0.55 (0.00)	0.43 (0.00)
Silvestri, CP-BP (ours)	0.90 (0.84)	0.82 (0.75)	0.71 (0.38)
Silvestri, CP	0.64 (0.59)	0.57 (0.42)	0.47 (0.30)
Random	0.14	0.10	0.08
Symbolic	1.00	1.00	0.99

Table 5.4 Minimum feasibility ratios demonstrating the impact of dataset size (100% vs. 10%, in parentheses) on various models. Our modifications enhance feasibility ratios across different problem sizes. Since we explore multiple loss variations (L1/L2) in some models, we report the best-performing configuration.

Table 5.4 indicates striking differences in the minimum feasibility ratios achieved by various

models when trained on full (100%) versus reduced (10%) datasets. Importantly, our modifications, particularly the integration of CP-BP feedback, consistently lead to improvements in feasibility ratios across all PLS sizes. This demonstrates the robustness of our approach, even when training data is limited.

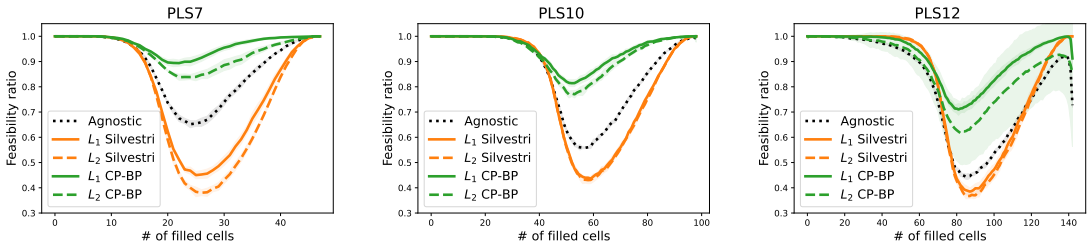
In most cases, models utilizing CP-BP outperform their counterparts that rely solely on CP or lack constraint-based guidance. This highlights the value of the richer information provided by CP-BP. However, there is an exception for PLS7, where our model with CP input achieves a slightly higher feasibility ratio than the Silvestri with CP-BP baseline. Therefore, the result shows that CP-BP is particularly helpful in larger instances. Even with a reduced dataset, our CP-BP based models still maintain high feasibility ratios.

Table 5.4 shows the significant performance gap between our best model and the baselines (MLP, Agnostic, and Silvestri variants). The impact of dataset size is particularly evident in the baselines, where a reduction to 10% of the data dramatically decreases their feasibility ratios. In contrast, our modifications limit this effect which suggests that they help the model learn the combinatorial structure more effectively.

5.4 Training without Label

The ability to train with unlabeled data offers a significant advantage in the context of combinatorial problems such as the PLS, where multiple valid solutions often exist. This approach eliminates the need for specialized techniques like the one in [23] and avoids the potential bias introduced by datasets where the label represents only one feasible completion, despite the possibility of multiple correct solutions which is noted in [43]. We address this potential bias by removing reliance on the label during training. Instead, we leverage either binary feedback from a traditional CP model or marginals from a CP-BP model. Both of these approaches reflect the potential for multiple feasible completions rather than a single arbitrary solution. This means we train the model only with the penalty term in the loss function. Mirroring our earlier approach, we first compare methods with no masking in loss calculation or prediction, and those using a single softmax over the entire model output as presented in Figure 5.9.

Figure 5.9 compares three baselines. The first, the Agnostic model, serves as the primary point of reference. This baseline is trained with labels. The second baseline, Silvestri with CP (orange lines), incorporates a loss function of either the L_1 or L_2. Finally, the Silvestri with CP-BP (green lines) uses CP-BP information in the loss function which is again either an L_1 or L_2 distance.

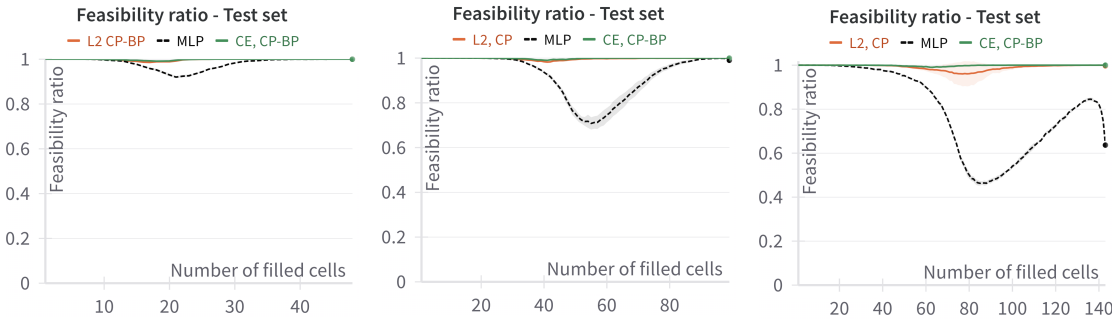


(a) $n = 7$ (b) $n = 10$ (c) $n = 12$

Figure 5.9 Training without the labels with no change in softmax and with no masking. The agnostic cure serves as a reference and is trained with labels. Our models (Silvestri with CP-BP input) are named L_1 and L_2 CP-BP. The orange lines are Silvetri baselines.

The results demonstrate that Silvestri with CP-BP models exhibits successful learning. It almost consistently outperforms the Agnostic model, indicating that the integration of only CP-BP feedback (and not labels) provides valuable guidance during the training process. While Silvestri with CP does show some signs of learning, its performance stays behind both the Agnostic model and Silvestri with CP-BP models. This suggests that while the binary feedback from a traditional CP solver can provide some guidance, it is ultimately less effective than the richer probabilistic information offered by CP-BP or the purely data-driven approach of the Agnostic model.

Note that in these three models, we do not apply cell-wise softmax and masking. We see the effect of these modifications in Figure 5.10.



(a) $n = 7$ (b) $n = 10$ (c) $n = 12$

Figure 5.10 Training without the labels with cell-wise softmax and masking. The MLP cure serves as a reference and is trained with labels. Other lines in green and orange are our models with either only CP or CP-BP input in the loss (with no labels).

Figure 5.10 demonstrates the impressive performance of our models (with either CP or CP-

BP) when trained without labels, comparing them to the traditionally trained MLP model (with labels). Despite the lack of labels, our models achieve near-perfect performance across different PLS problem sizes. For a more comprehensive comparison of training with and without labels, please refer to Table 5.5.

Model	**Minimum Feasibility Ratio - without (with) Label**		
	PLS7	**PLS10**	**PLS12**
Ours, CP-BP	**0.99** (0.98)	**0.99** (0.97)	**0.99** (0.91)
Ours, CP	0.98 (0.93)	0.98 (0.72)	0.96 (0.49)
Silvestri, CP-BP (ours)	0.90 (0.90)	0.81 (0.82)	0.71 (0.71)
Silvestri, CP	0.44 (0.64)	0.41 (0.57)	0.37 (0.47)
MLP (ours)	0.92	0.70	0.46
Agnostic	0.64	0.55	0.43
Random	0.14	0.10	0.08
Symbolic	1.00	1.00	0.99

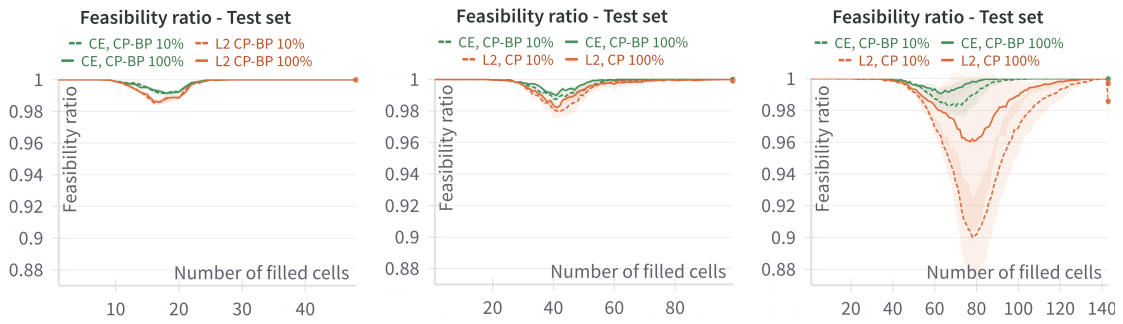
Table 5.5 Minimum feasibility ratios for different models trained without (and with) labels. Since we explore multiple loss variations (L_1/L_2) in some models, we report the best-performing configuration. Note that MLP and Agnostic are label-based; Random and Symbolic are non-learning references.

Table 5.5 demonstrates the effectiveness of our modifications (cell-wise softmax and masking) when training models without labels. Our models in the top two rows show that models trained without labels outperform their counterparts trained with labels. We see in the first row that our model with CP-BP input achieves near-perfect feasibility ratios of 0.99 across different PLS sizes. These results closely follow our Symbolic baseline. As discussed before, the dataset is biased and we can avoid this bias by following just the feedback from our CP/CP-BP models.

Table 5.5 reveals a key difference between the Silvestri models: Silvestri with CP-BP achieves comparable performance with or without labels, whereas Silvestri with CP shows a significant decrease of up to 20% in performance when trained without labels.

Finally, Our next step is to explore the potential of our best models to learn effectively from fewer data points. Figure 5.11 illustrates these results.

Figure 5.11 Shows these models when trained with 100% and 10% of data. Both models are performing very well and the difference is only in the more difficult problem, PLS = 12. Here, the CP-BP model maintains near-perfect performance, while the CP model experiences some performance decline. Table 5.6 provides a more detailed comparison of their minimum feasibility ratios.



(a) $n = 7$ (b) $n = 10$ (c) $n = 12$

Figure 5.11 Training on 10% od data without the labels with cell-wise softmax and masking. Results compare training the model with all of the dataset and with its 10% of the dataset.

Model	**Minimum Feasibility Ratio - 100% (and 10%) Data**		
	PLS7	**PLS10**	**PLS12**
Ours, CP-BP	0.99 (0.99)	0.99 (0.98)	0.99 (0.98)
Ours, CP	0.98 (0.98)	0.98 (0.98)	0.96 (0.90)
MLP (ours)	0.92	0.70	0.46
Agnostic	0.64	0.55	0.43
Random	0.14	0.10	0.08
Symbolic	1.00	1.00	0.99

Table 5.6 Minimum feasibility ratios for our best models trained without labels with 100% (and 10%) of data. Note that MLP and Agnostic are label-based; Random and Symbolic are non-learning references.

Table 5.6 shows that even with only 10% of the data, our best-performing model (with CP-BP input, trained without labels) maintains its effectiveness across various PLS sizes.

5.5 Discussion

Our experiments reveal the significant advantages of integrating constraint programming, specifically CP-BP, into MLP-based models for solving PLS completion problems. The CP-BP feedback, combined with cell-wise softmax and masking modifications, consistently enhances model performance. Models incorporating CP-BP demonstrate better results compared to those relying on traditional CP guidance or purely data-driven approaches like the Agnostic model. This indicates the value of CP-BP's richer information in guiding learning. Furthermore, our cell-wise softmax and masking techniques proved to boost feasibility ratios, particularly as problem instances grow in size.

The experiments show the effectiveness of our CP-BP based models when faced with limited datasets. They maintain high feasibility ratios even when trained with only 10% of the original data. In contrast, traditional neural networks like the Agnostic model experience notable performance degradation under reduced datasets, particularly when the PLS grid approaches a full state.

Finally, our models incorporating CP or CP-BP feedback demonstrate the ability to train effectively without explicit labels, when utilizing cell-wise softmax and masking. The performance of our CP-BP model closely matches the symbolic model, despite the Symbolic model not being a learning-based method. This label-free approach offers an advantage by avoiding potential biases introduced by datasets that represent only single-solution completions.

These findings suggest that the integration of CP-BP feedback, cell-wise softmax, and masking facilitates effective scaling to larger and more complex PLS instances. CP-BP serves as a source of domain knowledge, guiding the neural network toward solutions that satisfy the constraints even with limited or unlabeled data. Finally, the ability to train effectively without labels and with reduced datasets presents exciting possibilities for real-world applications where obtaining labeled data may be challenging or costly.

CHAPTER 6 CONCLUSION

6.1 Summary of Works

In this thesis, we presented an approach to combine neural networks with constraint programming to address the challenge of learning combinatorial structures. The primary limitation of neural networks in this context is their difficulty in capturing the interdependencies in the outputs. Our proposed method leverages the CP-BP framework to refine the learning process of networks.

The CP-BP framework provides marginal probabilities that are naturally a better match in the loss function. We trained neural networks to generate outputs that are consistent with the combinatorial structure defined by marginals computed by the CP-BP solver. This method represents a significant improvement over our baselines. More specifically, by applying the cell-wise softmax and masking, we could even obtain very good results with our model with CP input.

Our empirical studies confirm the effectiveness of our approach. We investigated the influence of training set size and found that our method helps compensate for smaller datasets. Furthermore, using our approach, the models can be trained without labeled data, which is particularly advantageous in settings where labeled data are scarce or expensive to obtain.

6.2 Limitations

The introduction of marginals, while beneficial for accuracy, creates a dependency on the quality of the CP-BP model. Therefore, when the CP-BP solver is not accurate we have misleading feedback that might hurt the accuracy.

Another key limitation is that we did not use a very diverse set of data to train our networks. The dataset employed for PLS, as outlined in the work of Silvestri et al. [43], presents certain inherent limitations that could influence the training and subsequent performance of machine learning models, particularly neural networks. One primary limitation stems from the dataset's construction methodology, where each dataset entry is derived from a single solution path of a fully completed PLS. This approach results in each partial grid state being linked to a unique, singular cell assignment leading to the complete grid. Such a dataset design inherently introduces a bias, as it represents a very limited perspective of the diverse range of possible solutions for each partial state of a PLS. This bias can have a consequential

impact on the learning process of neural networks trained on this dataset. Therefore, the networks might not generalize effectively to other valid solutions or configurations of PLS that are not represented in the training data. However, this bias does not affect our models when trained without labels.

Lastly, we only looked at one specific problem in our study. Different problems might have their own unique challenges, and our approach might not work as expected. In the future, it would be good to test our method with different kinds of problems to investigate if it works just as well or if it requires additional adoptations.

6.3 Future Research

The flexibility of our approach allows it to be extended beyond our experimental setting. The key requirement is that the combinatorial structure of the problem can be modeled using the CP-BP solver. Our framework can then be adapted to guide the learning process.

Furthermore, during inference, marginals from the CP-BP framework could guide the neural network toward predictions that align with the problem's structure. This represents a potential research direction with computational advantages.

Finally, exploring RNN or transformer-based architectures could lead to a single, adaptive model capable of handling various PLS sizes which would be more practical. These architectures can handle sequences of varying lengths (PLS in our experiments). Such a model would have the potential to generalize to unseen PLS sizes, for example, PLS of size 15.

REFERENCES

[1] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[2] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *IEEE Computational intelligence magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[4] J. Y. Lee, S. V. Mehta, M. Wick, J. Tristan, and J. G. Carbonell, "Gradient-based inference for networks with output constraints," in *AAAI*, 2019, pp. 4147–4154.

[5] G. Pesant, "From support propagation to belief propagation in constraint programming," *Journal of artificial intelligence research.*, vol. 66, pp. 123–150, 2019. [Online]. Available: https://doi.org/10.1613/jair.1.11487

[6] X. Wei *et al.*, "Difficulty level classification of sudoku puzzles based on convolutional neural network," *Academic journal of computing & information science*, vol. 6, no. 11, pp. 35–39, 2023.

[7] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[8] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*. Springer, 2006, vol. 4, no. 4.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[10] C. M. Bishop and H. Bishop, "Deep learning: foundations and concepts." Springer, 2023, pp. 1–22.

[11] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[12] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[13] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[14] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[15] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[16] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[18] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[19] W. van Hoeve and I. Katriel, "Global constraints," in *Handbook of constraint programming*. Elsevier, 2006, vol. 2, pp. 169–208. [Online]. Available: https://doi.org/10.1016/S1574-6526(06)80010-6

[20] B. Babaki, B. Omrani, and G. Pesant, "Combinatorial search in cp-based iterated belief propagation," vol. 12333. Springer, 2020, pp. 21–36. [Online]. Available: https://doi.org/10.1007/978-3-030-58475-7_2

[21] D. Belanger and A. McCallum, "Structured prediction energy networks," in *International conference on machine learning*, vol. 48. PMLR, 20–22 Jun 2016, pp. 983–992. [Online]. Available: https://proceedings.mlr.press/v48/belanger16.html

[22] D. Belanger, B. Yang, and A. McCallum, "End-to-end learning for structured prediction energy networks," in *International conference on machine learning*. PMLR, 2017, pp. 429–439.

[23] Y. Nandwani, D. Jindal, Mausam, and P. Singla, "Neural learning of one-of-many solutions for combinatorial problems in structured output spaces," in *International conference on learning representations, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=ATp1nW2FuZL

[24] Y. Nandwani, A. Pathak, and P. Singla, "A primal dual formulation for deep learning with constraints," *Advances in neural information processing systems*, vol. 32, 2019.

[25] J. Mandi, P. J. Stuckey, T. Guns *et al.*, "Smart predict-and-optimize for hard combinatorial optimization problems," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 02, 2020, pp. 1603–1610.

[26] M. Mulamba, J. Mandi, M. Diligenti, M. Lombardi, V. Bucarey, and T. Guns, "Contrastive losses and solution caching for predict-and-optimize," *arXiv*, 2020.

[27] C. Gambella, B. Ghaddar, and J. Naoum-Sawaya, "Optimization problems for machine learning: A survey," *European journal of operational research*, vol. 290, no. 3, pp. 807–828, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S037722172030758X

[28] Z. Hu, X. Ma, Z. Liu, E. Hovy, and E. Xing, "Harnessing deep neural networks with logic rules," *arXiv*, 2016.

[29] R. T. Icarte, L. Illanes, M. P. Castro, A. A. Ciré, S. A. McIlraith, and J. C. Beck, "Training binarized neural networks using MIP and CP," in *Principles and practice of constraint programming*, vol. 11802. Springer, 2019, pp. 401–417. [Online]. Available: https://doi.org/10.1007/978-3-030-30048-7_24

[30] L. De Raedt, S. Dumančić, R. Manhaeve, and G. Marra, "From statistical relational to neuro-symbolic artificial intelligence," *arXiv*, 2020.

[31] J. Xu, Z. Zhang, T. Friedman, Y. Liang, and G. V. den Broeck, "A semantic loss function for deep learning with symbolic knowledge," in *International conference on machine learning*, J. G. Dy and A. Krause, Eds., vol. 80. PMLR, 2018, pp. 5498–5507. [Online]. Available: http://proceedings.mlr.press/v80/xu18h.html

[32] K. Ahmed, E. Wang, K. Chang, and G. V. den Broeck, "Neuro-symbolic entropy regularization," in *Uncertainty in artificial intelligence*, vol. 180. PMLR, 2022, pp. 43–53. [Online]. Available: https://proceedings.mlr.press/v180/ahmed22a.html

[33] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, and L. D. Raedt, "Neural probabilistic logic programming in deepproblog," *Artificial intelligence*, vol. 298, p. 103504, 2021. [Online]. Available: https://doi.org/10.1016/j.artint.2021.103504

[34] L. De Raedt, A. Kimmig, and H. Toivonen, "Problog: a probabilistic prolog and its application in link discovery," in *International joint conference on artificial intelligence*, 2007, pp. 2462–2467.

[35] Z. Yang, A. Ishay, and J. Lee, "Neurasp: embracing neural networks into answer set programming," in *International joint conference on artificial intelligence*, 2020, pp. 1755–1762. [Online]. Available: https://doi.org/10.24963/ijcai.2020/243

[36] K. Ahmed, S. Teso, K. Chang, G. V. den Broeck, and A. Vergari, "Semantic probabilistic layers for neuro-symbolic learning," in *NeurIPS*, 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/c182ec594f38926b7fcb827635b9a8f4-Abstract-Conference.html

[37] P. Wang, P. L. Donti, B. Wilder, and J. Z. Kolter, "Satnet: bridging deep learning and logical reasoning using a differentiable satisfiability solver," in *International conference on machine learning*, vol. 97. PMLR, 2019, pp. 6545–6554. [Online]. Available: http://proceedings.mlr.press/v97/wang19e.html

[38] K. Ahmed, K. Chang, and G. V. den Broeck, "Semantic strengthening of neuro-symbolic learning," in *International conference on artificial intelligence and statistics*, vol. 206. PMLR, 2023, pp. 10 252–10 261. [Online]. Available: https://proceedings.mlr.press/v206/ahmed23a.html

[39] Z. Yang, J. Lee, and C. Park, "Injecting logical constraints into neural networks via straight-through estimators," in *International conference on machine learning*, vol. 162. PMLR, 2022, pp. 25 096–25 122. [Online]. Available: https://proceedings.mlr.press/v162/yang22h.html

[40] D. Chen, Y. Bai, W. Zhao, S. Ament, J. Gregoire, and C. Gomes, "Deep reasoning networks for unsupervised pattern de-mixing with constraint reasoning," in *International conference on machine learning*. PMLR, 2020, pp. 1500–1509.

[41] Y. Bai, D. Chen, and C. P. Gomes, "CLR-DRNets: curriculum learning with restarts to solve visual combinatorial games," in *International conference on principles and practice of constraint programming*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[42] D. Lafleur, S. Chandar, and G. Pesant, "Combining reinforcement learning and constraint programming for sequence-generation tasks with hard constraints," in *International conference on principles and practice of constraint programming*, vol. 235. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 30:1–30:16. [Online]. Available: https://doi.org/10.4230/LIPIcs.CP.2022.30

[43] M. Silvestri, M. Lombardi, and M. Milano, "Injecting domain knowledge in neural networks: a controlled experiment on a constrained problem," in *Integration of constraint programming, artificial intelligence, and operations research*, P. J. Stuckey, Ed., vol. 12735. Springer, 2021, pp. 266–282. [Online]. Available: https://doi.org/10.1007/978-3-030-78230-6_17

[44] W. J. van Hoeve, G. Pesant, and L. Rousseau, "On global warming: flow-based soft global constraints," *J. Heuristics*, vol. 12, no. 4-5, pp. 347–373, 2006. [Online]. Available: https://doi.org/10.1007/s10732-006-6550-4

[45] A. Galassi, M. Lombardi, P. Mello, and M. Milano, "Model agnostic solution of csps via deep learning: a preliminary study," in *International conference on the integration of constraint programming, artificial intelligence, and operations research*. Springer, 2018, pp. 254–262.

[46] D. P. Kingma and J. Ba, "Adam: a method for stochastic optimization," *arXiv*, 2014.

[47] L. Perron and F. Didier, "CP-SAT," Google. [Online]. Available: https://developers.google.com/optimization/cp/cp_solver/

APPENDIX A PLS HYPER-PARAMETERS

This appendix provides detailed hyperparameters used in our models for problem sizes of 7, 10, and 12. These configurations are used for optimizing the model's performance for specific problem dimensions.

General Hyperparameters

First, we list hyperparameters that are common across all problem sizes.

Parameter	Value	Description
`validation set`	5000	The size of validation set.
`save model`	1	Saving the model after each evaluation step.
`patience`	10	If the model performance does not increase after 10 evaluation steps the model stops and the best model will be selected.
`number of layers`	2	Number of hidden layers in the fully connected network.
`neuron per layer`	512	Number of neurons in each hidden layer.
`learning rate`	0.001	The learning rate for the Adam optimizer.
`max epochs`	500	Maximum number of training epochs.

Table A.1 General Hyperparameters

Hyperparameters by Problem Size

Next, we detail the hyperparameters that vary depending on the problem size (PLS).

Problem Size 7

Parameter	Value for PLS 7	Description
`input dimansion`	7^3	Input dimension for neural network.
`batch size`	2024	Mini-batch size for training the network.
`train size`	350,000	Training set size.
`test size`	190,000	Test set size.
`eval steps`	1	Evaluating every epoch.

Table A.2 Hyperparameters for PLS of Size 7

Problem Size 10

Parameter	Value for PLS 10	Description
`input dimansion`	10^3	Input dimension for neural network.
`batch size`	50,000	Mini-batch size for training the network.
`train size`	700,000	Training set size.
`test size`	300,000	Test set size.
`eval steps`	10	Evaluating every 10 epochs.

Table A.3 Hyperparameters for PLS of Size 10

Problem Size 12

Parameter	Value for PLS 12	Description
`input dimansion`	12^3	Input dimension for neural network.
`batch size`	50,000	Mini-batch size for training the network.
`train size`	1,000,000	Training set size.
`test size`	440,000	Test set size.
`eval steps`	10	Evaluating every 10 epochs.

Table A.4 Hyperparameters for PLS of Size 12

Additional Notes

- Since the output of the CB-BP solver is a probability distribution for each square in PLS, we have used KL-divergence and cross-entropy for the penalty.
- We tried penalty coefficient, $\lambda = 0.1, 1, 10$.

APPENDIX B CP-BP MARGINAL COMPUTATION TIMES

CP-BP Marginal Computation Times

The following table summarizes the average time required to calculate CP-BP marginals for different PLS grid sizes on a Dell Mobile Precision Workstation 3560 BTX with processor Intel Core Processor i7-1165G7 (4 Core, 12MB Cache, 2.8 GHz up to 4.70 GHz Turbo, 17.5W) and 16 GB, DDR4, 3200Mhz ram.:

Grid Size	Training Dataset	Time per Grid (seconds)	Total Time per Dataset (hours)
7 x 7	350,000	0.034	3.31
10 x 10	700,000	0.077	14.97
12 x 12	1,000,000	0.147	40.83

Table B.1 CP-BP Marginal Calculation Times

Note that we need to find these marginal only once.