الجمهورية الشعبية الديمقراطية الجزائرية
**République Algérienne Démocratique et Populaire**
وزارة التعليم العالي و البحث العلمي
**Ministère de l'Enseignement supérieur et de la Recherche scientifique**

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

**Ecole Nationale Polytechnique**
**Département Génie Industriel**

# End of Study Project Dissertation

for **Obtaining State Engineer's Degree**
in **Industrial Engineering**
Option : **Data Science & Artificial Intelligence**

# Exploring a new method for the formal verification of neural networks through Coloured Petri Net modeling

Presented by : **LAFIFI Bochra**

Defended on **29 June, 2025**, before a jury composed of :

| | |
|---|---|
| **Mr. Zouaghi Iskander** | President, ENP |
| **Mrs. Beldjoudi Samia** | Examiner, ENP |
| **Mr. Arki Oussama** | Supervisor, ENP |
| **Mrs. Gabis Asma** | Supervisor, EFREI |
| **Mr. Klai Kais** | Supervisor, LIPN, USPN |
| **Mrs. Chakchouk Faten** | Supervisor, EFREI |

ENP 2025

الجمهورية الشعبية الديمقراطية الجزائرية
**République Algérienne Démocratique et Populaire**
وزارة التعليم العالي و البحث العلمي
**Ministère de l'Enseignement supérieur et de la Recherche scientifique**

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

**Ecole Nationale Polytechnique**
**Département Génie Industriel**

# End of Study Project Dissertation

for **Obtaining State Engineer's Degree**
in **Industrial Engineering**
Option : **Data Science & Artificial Intelligence**

# Exploring a new method for the formal verification of neural networks through Coloured Petri Net modeling

Presented by : **LAFIFI Bochra**

Defended on **29 June, 2025**, before a jury composed of :

| | |
|---|---|
| **Mr. Zouaghi Iskander** | President, ENP |
| **Mrs. Beldjoudi Samia** | Examiner, ENP |
| **Mr. Arki Oussama** | Supervisor, ENP |
| **Mrs. Gabis Asma** | Supervisor, EFREI |
| **Mr. Klai Kais** | Supervisor, LIPN, USPN |
| **Mrs. Chakchouk Faten** | Supervisor, EFREI |

ENP 2025

الجمهورية الشعبية الديمقراطية الجزائرية
**République Algérienne Démocratique et Populaire**
وزارة التعليم العالي و البحث العلمي
**Ministère de l'Enseignement Supérieur et de la Recherche Scientifique**

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

**Ecole Nationale Polytechnique**
**Département Génie Industriel**

# Mémoire de Projet de Fin d'Études

en vue de l'obtention du **Diplôme d'Ingénieur d'État**
en **Génie Industriel**
Option : **Science des Données & Intelligence Artificielle**

# Vérification formelle des réseaux de neurones via les Réseaux de Petri Colorés

Présenté par : **LAFIFI Bochra**

Soutenu le **29 Juin 2025**, devant le jury composé de :

| | |
|---|---|
| **M. Zouaghi Iskander** | Président, ENP |
| **Mme. Beldjoudi Samia** | Examinatrice, ENP |
| **M. Arki Oussama** | Superviseur, ENP |
| **Mme. Gabis Asma** | Superviseuse, EFREI |
| **M. Klai Kais** | Superviseur, LIPN, USPN |
| **Mme. Chakchouk Faten** | Superviseuse, EFREI |

ENP 2025

# الملخص

تقدم هذه الأطروحة نهجاً مبتكراً لقابلية تفسير الشبكات العصبية من خلال دمج شبكات بتري الملونة مع بنى الإدراك متعدد الطبقات، مما ينتج عنه نموذج الشبكة العصبية لبتري الملونة. يتناول هذا النموذج تحدي القابلية للتفسير في أنظمة التعلم العميق من خلال توفير رؤى مؤسسة رياضياً حول أهمية الخصائص وعمليات اتخاذ القرار.

يستفيد إطار عمل الشبكة العصبية لبتري الملونة من قدرات التحقق الرسمي لشبكات بتري الملونة لالتقاط الحالات الحاسوبية المتوسطة أثناء الانتشار الأمامي، مما يتيح تحليلاً دقيقاً لتدفق المعلومات وفحصاً شفافاً لمساهمات الخصائص. من خلال الجمع بين الطرق الرسمية والشبكات العصبية، يوضح هذا البحث كيف يمكن للشبكات العصبية لبتري الملونة تحسين قابلية تفسير أنظمة الذكاء الاصطناعي دون التضحية بالأداء، خاصة في تطبيقات الرعاية الصحية.

علاوة على ذلك، يوفر التحليل الرياضي لتأثيرات المعاملات الفائقة على تعقيد مساحة الحالة نموذجاً تنبؤياً للمتطلبات الحاسوبية، مما يسلط الضوء على التأثير الكبير لعمق الطبقات وحجم الدفعات الصغيرة على التحقق من النموذج وتحسينه. يضع هذا العمل الأسس لتطوير أنظمة تعلم عميق قابلة للتفسير وفعالة وقابلة للتحقق في التطبيقات الحرجة.

---

**الكلمات المفتاحية:** شبكات بتري الملونة، الشبكات العصبية، التحقق الرسمي، الذكاء الاصطناعي القابل للتفسير، فحص النماذج

---

# Résumé

Cette thèse présente une approche novatrice pour l'interprétabilité des réseaux de neurones en intégrant les Réseaux de Petri Colorés (RPC) avec les architectures de perceptron multicouche (PMC), donnant naissance au modèle de Réseau de Neurones de Petri Coloré. Ce modèle répond aux enjeux d'explicabilité dans l'apprentissage profond en permettant un suivi formel et détaillé du flux d'informations lors de la propagation. Cette approche offre une lecture transparente des contributions des caractéristiques et du processus de décision.

En tirant parti des capacités de vérification formelle des RPC, le modèle permet une analyse rigoureuse sans compromettre les performances prédictives — en particulier dans des domaines critiques comme la santé. De plus, une analyse mathématique de l'impact des hyperparamètres du réseau de neurones sur la complexité de l'espace d'états met en évidence l'influence de facteurs tels que la profondeur du réseau ou la taille des mini-lots sur les besoins en calcul, orientant ainsi vers une conception et une vérification plus efficaces.

Ce travail pose les fondations pour développer des systèmes d'apprentissage profond interprétables, efficaces et vérifiables dans les applications critiques.

---

**Mots-clés :** Réseaux de Petri Colorés, Réseaux de Neurones, Vérification Formelle, Intelligence Artificielle Explicable (IAX), Vérification de Modèles

---

# Abstract

This thesis introduces the **Colored Petri Neural Network (CPNN)**, a novel framework that integrates Colored Petri Nets (CPNs) with multi-layer perceptrons (MLPs) to enhance the interpretability of neural networks. The CPNN model addresses the challenge of explainability in deep learning by enabling formal, fine-grained tracking of information flow during forward propagation. This approach provides transparent insights into feature contributions and decision-making processes.

By leveraging the formal verification strengths of CPNs, the model supports rigorous analysis without compromising predictive performance—particularly in critical domains such as healthcare. Additionally, a mathematical investigation of the neural network hyperparameters effects on state space complexity reveals the influence of factors like layer depth and mini-batch size on computational requirements, guiding more efficient design and verification.

This work lays the foundation for developing interpretable, efficient, and verifiable deep learning systems in critical applications.

---

**Keywords:** Colored Petri Nets, Neural Networks, Formal Verification, Explainable Artificial Intelligence (XAI), Model Checking

---

# Acknowledgement

I could never have accomplished this work without, first and foremost, the will of *Allah*, and then the unwavering support of many individuals—starting with my supervising team. I am deeply grateful for the freedom they gave me, always encouraging my explorative approach while patiently guiding me toward completing this thesis.

Mrs. ***Asma Gabis***, more than anyone, played a pivotal role in helping me reach the end of this journey. From the very beginning, she offered invaluable mentorship and consistent support. Her insightful teaching, kind guidance, and steadfast encouragement were instrumental to my progress, delivered with a rare combination of pedagogical clarity and human warmth. Mrs. ***Faten Chakchouk*** and Mr. ***Kais Klai*** were also exceptional mentors. They ensured that I communicated my results clearly and meaningfully, encouraging me to consider the broader impact of my work. Mr. ***Kais***, in particular, opened new doors that allowed me to explore fields beyond my initial academic background. I consider myself truly fortunate to have worked with such inspiring individuals. Even before my arrival at Efrei, They facilitated my integration into the research lab.

I also extend my sincere thanks to the entire Efrei Research Lab team—doctoral students, associate professors, and fellow interns—for welcoming me with such generosity and for offering me a genuine glimpse into the collaborative and stimulating environment of real research.

This thesis has been made possible thanks to the invaluable support of my academic supervisor, Mr. ***Oussama Arki***, who provided insightful guidance throughout the course of this thesis. I am deeply grateful to the École Nationale Polytechnique, *Department of Industrial Engineering*, for the exceptional education and mentorship I have received during my academic journey. In particular, I would like to extend my sincere thanks to Mrs. ***Samia Beldjoudi***, for being a constant support to us, Mrs. ***Bahia Bouchafaa*** and Mr. ***Ali Boukabous***.

Mr. ***Iskander Zouaghi***, whose assistance in facilitating my internship from the very beginning has been crucial to the success of this work.

Finally, my heartfelt thanks to all my colleagues—those who offered advice, encouragement, or even just a kind word. Your support, in all its forms, has meant more than you know.

# Contents

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| **AI** | Artificial Intelligence |
| **ANN** | Artificial Neural Network |
| **APQ3** | Amplitude Perturbation Quotient (3 periods) |
| **APQ5** | Amplitude Perturbation Quotient (5 periods) |
| **APQ11** | Amplitude Perturbation Quotient (11 periods) |
| **AUC** | Area Under the Curve |
| **AUROC** | Area Under the Receiver Operating Characteristic Curve |
| **CPN** | Colored Petri Net |
| **CNN** | Convolutional Neural Network |
| **CPN Tools** | Colored Petri Net Tools |
| **CPNN** | Colored Petri Neural Network |
| **CT** | Control Transition |
| **CTL** | Computation Tree Logic |
| **DARPA** | Defense Advanced Research Projects Agency |
| **DDA** | Difference of Differences of Amplitudes |
| **DL** | Deep Learning |
| **ELU** | Exponential Linear Unit |
| **F1** | F1 Score (Precision and Recall balance) |
| **FP** | False Positive |
| **GDPR** | General Data Protection Regulation |
| **GRU** | Gated Recurrent Unit |
| **HCPNs** | Hierarchical Colored Petri Nets |
| **HPNs** | Hierarchical Petri Nets |
| **HTN** | Harmonics to Noise Ratio |
| **LIME** | Local Interpretable Model-agnostic Explanations |
| **LR** | Logistic Regression |
| **LRP** | Layer-wise Relevance Propagation |
| **LSTM** | Long Short-Term Memory |
| **LTL** | Linear Temporal Logic |
| **MAE** | Mean Absolute Error |
| **MAPE** | Mean Absolute Percentage Error |
| **ML** | Machine Learning |
| **MLP** | Multi-Layer Perceptron |
| **MSE** | Mean Squared Error |
| **NTH** | Noise to Harmonics Ratio |

| | |
|---|---|
| **PPQ5** | Pitch Perturbation Quotient (5 periods) |
| **PVS** | Prototype Verification System |
| **RAM** | Random Access Memory |
| **ReLU** | Rectified Linear Unit |
| **RF** | Random Forest |
| **RNN** | Recurrent Neural Network |
| **SCCs** | Strongly Connected Components |
| **SML** | Standard Meta Language |
| **SS** | State Space |
| **Tanh** | Hyperbolic Tangent |
| **TN** | True Negative |
| **TP** | True Positive |
| **XAI** | Explainable Artificial Intelligence |
| **XOR** | Exclusive OR |

# General Introduction

## Context and Background

In recent decades, Artificial Intelligence (AI) has emerged as a revolutionary field aiming to endow machines with capabilities that mimic human intelligence. These capabilities include learning, reasoning, decision-making, perception, and natural language understanding [1]. AI's goal is not just automation but the creation of intelligent systems that can autonomously adapt to new circumstances.

One of the most transformative subdomains of AI is Machine Learning (ML). Instead of relying on explicit rules programmed by humans, ML systems learn from data. This paradigm shift is clearly illustrated in Figure 1, which compares classical programming to machine learning. In classical programming, a human provides both the input and the rules (program/algorithm), and the machine produces a result. In contrast, machine learning systems are fed data and a desired outcome (goal); from these, the machine learns the rules (model) that map inputs to outputs.

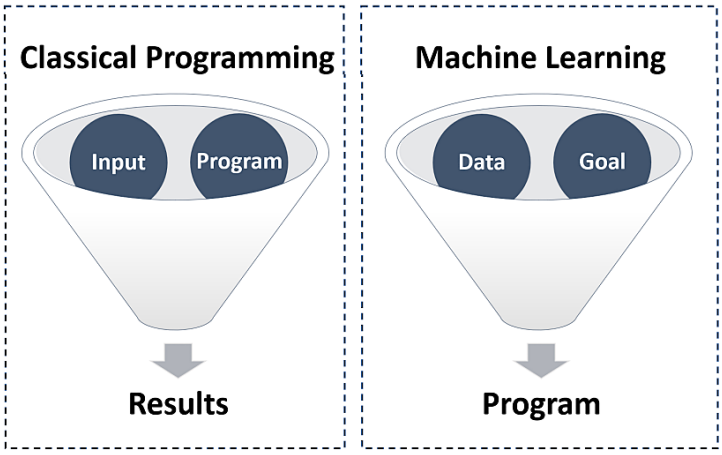


Figure 1: The difference between Classical Programming and Machine Learning [2].

This data-driven approach has proven particularly powerful in environments characterized by uncertainty, noise, or complexity [3] —situations where writing precise rules is infeasible. However, many ML models, especially traditional ones like decision trees or logistic regression, still rely on feature engineering and domain knowledge.

To address problems of higher complexity, Deep Learning (DL) has emerged as a powerful subclass of machine learning [4]. Deep learning algorithms use Artificial Neural Networks

(ANNs), which are computational models inspired by the human brain's neural structure. These networks are composed of layers of nodes (neurons), where each node processes a piece of the input and passes it forward. In networks with many such hidden layers—so-called deep neural networks—the model can automatically extract and hierarchically represent features from raw input data, such as images or audio.

Despite their powerful performance, deep neural networks are often criticized for their lack of transparency. These models are typically considered "black boxes," meaning that even their creators may not understand precisely how specific decisions are made [5].

This issue has led to two important areas of research:

- **Explainable Artificial Intelligence (XAI)** – A field aiming to make AI systems more interpretable and trustworthy. XAI seeks to provide human-understandable justifications for model predictions [6].
- **Formal Verification** – A set of mathematical techniques used to prove or disprove the correctness of a system's behavior with respect to a formal specification [7].

While traditionally used in hardware and safety-critical software systems, formal verification methods are increasingly being adapted for AI systems [7], particularly in high-stakes applications such as autonomous vehicles and medical diagnostics. The integration of formal verification with XAI represents a powerful approach, as it combines intuitive explanations with mathematical certainty—formal methods can verify that explanations accurately represent model behavior, while explanations can be grounded in mathematically proven properties rather than heuristic interpretations.

As AI systems continue to evolve and are deployed in mission-critical contexts, the ability to rigorously verify their correctness and explain their behavior becomes essential. This research work is positioned at the intersection of these concerns: exploring a new method for the formal verification of neural networks through Coloured Petri Net modeling.

# Problem Statement

This thesis aims to address the following research questions: How can we overcome the fundamental "black box" problem of deep neural networks by developing a unified framework that simultaneously provides mathematical transparency, visualization and formal verification capabilities ?

# Our Approach and Contributions

The contributions of this thesis can be summarized as follows:

- Development of the CPNN Framework: This work introduces the Colored Petri Neural Network (CPNN) as a novel methodology to enhance the interpretability and transparency of neural networks. The integration of Colored Petri Nets with MLP architectures has provided a formal, visual, and verifiable representation of the training and evaluation processes of neural networks.

- Formal Verification of Neural Networks: The application of CPNs' formal verification capabilities allows for the capture and analysis of intermediate computational states during forward propagation. This deepens our understanding of information flow and decision-making within the network.
- Feature Importance Analysis: the novel approach that we use to calculate feature importance each iteration is proposed through the use of our CPNN Model's state space report. This enables a transparent examination of how individual features influence the output predictions, improving the explainability of complex deep learning models.
- Mathematical Modeling of Hyperparameter Impact: The impact of hyperparameters on state space complexity is analyzed mathematically. A predictive model is developed to help practitioners understand the computational implications of various configurations, aiding in the optimization of deep learning models for better efficiency and performance.
- Applications in Healthcare: The CPNN framework, with its focus on both interpretability and performance, holds significant promise in healthcare applications, where understanding AI-driven decisions is crucial. This work paves the way for further research and implementation of interpretable and verifiable AI models in critical domains like healthcare.

# Structure of the Thesis

Our thesis will be structured around **four main parts**, each dedicated to a fundamental aspect of our research and providing an in-depth synthesis of the work carried out. Each part aims to address our research question and achieve the defined objectives.

**First: State of the Art (Chapters 1 & 2)**
This part establishes the theoretical foundation and reviews existing work. Chapter 1 examines machine learning fundamentals, neural network architectures, Explainable AI (XAI), formal verification techniques, and Petri Nets theory. Chapter 2 provides a critical analysis of existing approaches to modeling neural networks with Petri Nets, reviewing key contributions and identifying gaps that position our research within the broader context of interpretable AI.

**Second: Methodology and Model Development (Chapter 3)**
This core section presents our novel Colored Petri Neural Network (CPNN) model, detailing the architecture formalization, color sets, learning mechanisms, and backpropagation design that enable interpretability while maintaining performance.

**Third: Mathematical Analysis and Complexity Study (Chapter 4)**
We provide a rigorous mathematical examination of neural network hyperparameters' effects on state space complexity, introducing batch-based complexity concepts and deriving predictive models for computational requirements.

**Fourth: Experimental Validation and Explainability Analysis (Chapter 5)**

This section validates our CPNN model through comprehensive experiments, exploring explainability aspects and providing comparative validation against baselines to demonstrate interpretability advantages.

# Chapter 1

# Foundational Concepts

## Introduction

This chapter introduces the key theoretical concepts that underpin the work presented in this thesis. It begins with an overview of machine learning, covering its main types—supervised and unsupervised learning—along with essential evaluation metrics.

Next, we present the fundamentals of deep learning and neural networks, including their architecture, training process, and activation functions.

We then explore the emerging field of explainable AI (XAI), highlighting the need for model transparency and the trade-offs between accuracy and interpretability.

Finally, we introduce formal verification and Petri nets, which provide rigorous tools for modeling, analyzing, and verifying complex systems.

## 1.1 Machine Learning

Machine learning is a subset of artificial intelligence that involves building computer models capable of learning and making independent predictions or decisions based on provided data [8]. These models continually improve their accuracy through learned data. Arthur Samuel, who coined the term "machine learning" in 1959, described it as "the ability of computers to learn without programming new skills directly" [9]. Using available datasets, machine learning algorithms supported by mathematical models generate predictions or specific decisions. The main categories of machine learning are supervised and unsupervised learning [10].

### 1.1.1 Supervised Learning

Supervised machine learning operates on datasets where both input data and corresponding output data are provided. Once the model learns the relationship between input and output, it can classify new unknown datasets and make predictions or decisions based on them [11]. This type of learning is divided into two primary methods: classification and regression [12].

In classification, the solutions are typically binary or categorical. For example, an algorithm might classify a photograph as either a cat or a dog, representing a two-class classification problem. Another example is handwriting recognition, where software matches handwritten characters (output data) to their corresponding printed counterparts (classes).

Regression, a fundamental type of supervised learning, predicts continuous values using input data. In healthcare applications, regression can forecast medical costs based on input data such as drug prices, required medical equipment, and staff expenses. Through training these models with input and output data, predictions for total treatment costs can be made for new inputs.

### Metrics of evaluation

Machine learning models require robust evaluation frameworks to assess their performance and guide optimization. The specific metrics used depend largely on the problem type, with classification and regression tasks employing distinct evaluation approaches. Selecting appropriate performance metrics enables researchers and practitioners to quantify model efficacy, compare algorithms, and determine fitness for specific applications [13].

**Regression Metrics:** Regression metrics evaluate how well a model predicts continuous numerical values:

- **Mean Squared Error (MSE)** calculates the average of squared differences between predicted and actual values. By squaring errors, it penalizes larger mistakes more severely.
- **Root Mean Squared Error (RMSE)** provides the square root of MSE, expressing the error in the same units as the target variable, making interpretation more intuitive.
- **Mean Absolute Error (MAE)** measures the average magnitude of errors without considering their direction. Unlike MSE, it weights all errors linearly rather than quadratically.
- **Mean Absolute Percentage Error (MAPE)** expresses accuracy as a percentage of error, making it scale-independent and useful for comparing predictions across different scales.
- **R-squared (Coefficient of Determination)** quantifies the proportion of variance in the dependent variable explained by the independent variables. Values range from 0 to 1, with higher values indicating better fit.
- **Adjusted R-squared** modifies standard R-squared by penalizing excessive use of variables, providing a more accurate measure when comparing models with different numbers of predictors.

**Classification Metrics:** Classification metrics evaluate how well a model categorizes data into discrete classes:

- **Accuracy** measures the proportion of correct predictions among all predictions. While intuitive, it can be misleading with imbalanced datasets where the majority class dominates the metric.
- **Precision** (positive predictive value) quantifies the proportion of positive identifications that were actually correct. It answers: "Of all instances predicted as positive, how many were truly positive?"
- **Recall** (sensitivity) measures the proportion of actual positives correctly identified. It addresses: "Of all actual positive instances, how many did the model detect?"
- **F1 Score** represents the harmonic mean of precision and recall, providing a balance between the two metrics, particularly useful when class distribution is uneven.
- **Area Under the Receiver Operating Characteristic curve (AUROC)** evaluates classification performance across various threshold settings by plotting the true positive rate against the false positive rate. A higher AUROC indicates better discrimination capability.
- **Confusion Matrix** provides a comprehensive view of classification performance by tabulating predicted classes against actual classes, enabling the calculation of various metrics and visualization of error patterns.

| | Actually Positive (1) | Actually Negative (0) |
|---|---|---|
| Predicted Positive (1) | True Positives (TPs) | False Positives (FPs) |
| Predicted Negative (0) | False Negatives (FNs) | True Negatives (TNs) |

Figure 1.1: Confusion Matrix for Classification Performance Evaluation

### 1.1.2 Unsupervised Learning

Unsupervised machine learning differs fundamentally from supervised learning in its use of unannotated data that has not been previously labeled by humans or algorithms. The model learns from input data without expected values, and the available dataset does not provide answers to the given task. Instead of labeling or predicting outputs, this algorithm focuses on grouping data based on their inherent characteristics. The goal is to enable the machine to detect patterns and group data without a single correct answer [9].

There are two primary types of unsupervised learning approaches:

- **Clustering:** This involves grouping data based on their similarities and differences. For example, animals can be divided into groups based on their visual features determined using the model [14].
- **Association:** This method analyzes relationships between data in a dataset. For instance, the algorithm can pair people buying mattresses for pressure sores with those ordering products to aid in the healing of pressure sores. This method is commonly used in marketing strategies [15].

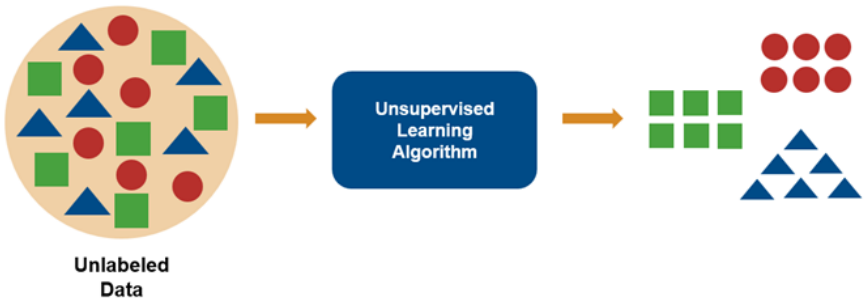


Figure 1.2: Unsupervised Learning @MATLAB

## 1.2 Deep Learning

Deep learning [4] is an advanced form of machine learning that enables computers to learn from experience and understand the world in terms of a hierarchy of concepts. Because the computer gathers knowledge from experience, there is no need for a human operator to formally specify all of the knowledge needed by the computer. The power of deep learning stems from its fundamental architectural component—neural networks—which serve as the computational foundation for these sophisticated learning systems.

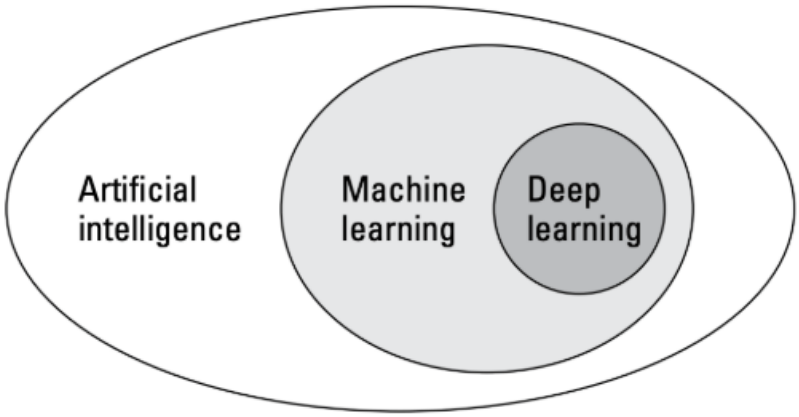


Figure 1.3: Deep Learning, Machine Learning and AI

## 1.3 Neural Networks

Artificial neural networks resemble the human brain in structure and function, comprising multiple perceptrons (or neurons) that process and transmit information. Understanding their functioning requires examining how data flows through the network. These networks form the backbone of deep learning systems, with their layered structure allowing for the hierarchical concept learning that defines deep learning approaches. While simple neural networks contain only a few layers of neurons, deep learning utilizes deep neural networks with many hidden layers, enabling the automatic extraction and representation of increasingly complex features from raw data.

Neural networks begin with inputting data, such as images, text, or sound. This data traverses the network, processed through successive layers of perceptrons until reaching the output. Each layer contains multiple neurons that process the input data.

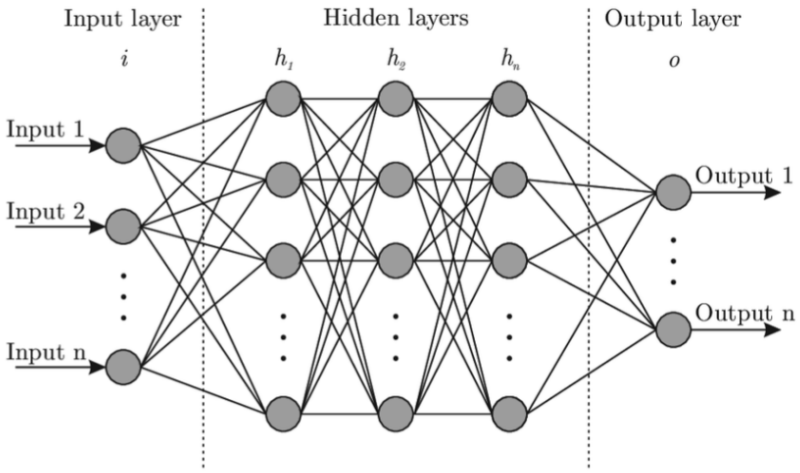


Figure 1.4: Basic Structure of a Neural Network [16]

### 1.3.1 Formal Definition of a Feedforward Neural Network

A feedforward neural network can be formally defined as a 6-tuple:

$$\text{NN} = (L, N, C, W, B, A) \tag{1.1}$$

Where:

1. $L = \{L_0, L_1, \ldots, L_n\}$ is an ordered set of layers, where:
   - $L_0$ is the input layer
   - $L_1, L_2, \ldots, L_{n-1}$ are hidden layers
   - $L_n$ is the output layer
2. $N = \{N(i) \mid i \in [0, n]\}$ where $N(i)$ represents the set of neurons in layer $L_i$:
   - $N(i) = \{n_1(i), n_2(i), \ldots, n_k(i)\}$ where $k$ is the number of neurons in layer $L_i$

- $|N(i)|$ denotes the number of neurons in layer $L_i$

3. $C \subseteq \{(n_j(i-1), n_k(i)) \mid i \in [1,n], j \in [1,|N(i-1)|], k \in [1,|N(i)|]\}$ is the set of connections between neurons:
   - Each connection is an ordered pair $(n_j(i-1), n_k(i))$ representing a directed link from neuron $j$ in layer $i-1$ to neuron $k$ in layer $i$
   - In a fully connected feedforward network, $C$ contains all possible connections between adjacent layers
4. $W : C \to \mathbb{R}$ is a weight function that maps each connection to a real-valued weight:
   - $W(n_j(i-1), n_k(i)) = w_{jk}(i) \in \mathbb{R}$
   - $W$ can alternatively be represented as a set of matrices $\{W(1), W(2), \ldots, W(n)\}$ where $W(i) \in \mathbb{R}^{|N(i-1)| \times |N(i)|}$
5. $B = \{B(i) \mid i \in [1,n]\}$ where $B(i)$ is a vector of bias values for layer $L_i$:
   - $B(i) = \{b_1(i), b_2(i), \ldots, b_k(i)\}$ where $k = |N(i)|$
   - $B(i) \in \mathbb{R}^{|N(i)|}$
6. $A = \{A(i) \mid i \in [1,n]\}$ where $A(i)$ represents the activation function applied to layer $L_i$:
   - $A(i) : \mathbb{R} \to \mathbb{R}$
   - Common examples: ReLU, Sigmoid, Tanh, etc.

The formal architecture defined as the 6-tuple 1.1 is engaged in two primary computational processes: **forward propagation** and **backward propagation**. These two phases leverage different elements of the tuple to respectively compute outputs and update the network's parameters during training.

**1. Forward Propagation.** The forward propagation phase computes the output of the neural network for a given input. It proceeds layer by layer, from the input layer $L_0$ to the output layer $L_n$, using the following components:

- $L$ (Layers) and $N$ (Neurons)
- $C$ (Connections)
- $W$ (Weights)
- $B$ (Biases)
- $A$ (Activation Functions)

The computation at each neuron $n_k^{(i)} \in N(i)$, for $i \in [1, n]$, is given by:

$$z_k^{(i)} = \sum_{j=1}^{|N(i-1)|} w_{jk}^{(i)} \cdot a_j^{(i-1)} + b_k^{(i)} \quad \text{and} \quad a_k^{(i)} = A^{(i)}(z_k^{(i)})$$

where:

- $w_{jk}^{(i)} = W(n_j^{(i-1)}, n_k^{(i)})$
- $a_j^{(i-1)}$ is the activation of neuron $n_j^{(i-1)}$
- $b_k^{(i)}$ is the bias of neuron $n_k^{(i)}$
- $A^{(i)}$ is the activation function applied at layer $L_i$

This process produces the final output vector at layer $L_n$, representing the prediction of the network.

**2. Backward Propagation.** The backward propagation phase computes gradients of the loss function with respect to each parameter in the network (weights and biases) and updates them accordingly. This phase uses the following elements:

- $W$ (Weights) and $B$ (Biases): updated based on the computed gradients.
- $A$ (Activation Functions): their derivatives are needed for gradient computation.
- $C$ (Connections): defines the dependencies between layers for propagating errors backward.
- $L$ (Layers) and $N$ (Neurons): the layers and neurons guide the reverse computation order.

For each layer $i$ from $n$ down to 1, the gradients are computed as:

$$\delta^{(i)} = (W^{(i+1)})^\top \delta^{(i+1)} \circ A'^{(i)}(z^{(i)}) \tag{1.2}$$

$$\nabla W^{(i)} = a^{(i-1)}(\delta^{(i)})^\top \tag{1.3}$$

$$\nabla B^{(i)} = \delta^{(i)} \tag{1.4}$$

where:

- $\delta^{(i)}$ is the error signal at layer $i$
- $A'^{(i)}$ is the derivative of the activation function
- $\circ$ denotes the Hadamard (element-wise) product

The parameters are then updated using gradient descent:

$$W^{(i)} \leftarrow W^{(i)} - \eta \nabla W^{(i)}, \quad B^{(i)} \leftarrow B^{(i)} - \eta \nabla B^{(i)}$$

where $\eta$ is the learning rate.

| Tuple Element | Used in Forward Propagation | Used in Backward Propagation |
|---|---|---|
| $L$ | Defines flow direction | Defines reverse pass |
| $N$ | Neuron activations | Gradient propagation |
| $C$ | Connection traversal | Dependency tracking |
| $W$ | Input weighting | Updated via gradients |
| $B$ | Threshold shifting | Updated via gradients |
| $A$ | Non-linear transformation | Derivative used in backprop |

Table 1.1: Tuple component usage in forward and backward propagation

### 1.3.2 Types of Neural Networks

Neural networks are widely used in image recognition, natural language processing, speech recognition, and stock price prediction. They come in various types:

- **Perceptron Networks:** [17] The simplest neural networks with an input and output layer composed of perceptrons. Perceptrons assign a value of one or zero based on the activation threshold, dividing the set into two.
- **Layered Networks (Feed Forward):** [18] These contain multiple layers of interconnected neurons where the outputs of the previous layer neurons serve as the inputs for the next layer. The neurons of each successive layer always have a +1 input from the previous layer. These networks enable the classification of non-binary sets and are used in image, text, and speech recognition.
- **Recurrent Networks:** [19] Neural networks with feedback loops where the output signals feed back into the input neurons. They can generate sequences of phenomena and signals until the output stabilizes and are used for sentiment analysis and text generation.
- **Convolutional Networks:** [20] Also known as braided networks, these are specialized for processing data with grid-like topology, such as images.
- **Gated Recurrent Unit (GRU) and Long Short-Term Memory (LSTM) Networks:** [21, 22] These perform recursive tasks with the output dependent on previous calculations. They possess network memory, allowing them to remember data states across different time steps. These networks have longer training times and are applied in time series analysis (e.g., stock prices), autonomous car trajectory prediction, text-to-speech conversion, and language translation.

### 1.3.3 Training Process

To train perceptrons, weights are adjusted to minimize the difference between the output and the expected signal. The network also learns through the gradient descent method [23], adjusting the step lengths in the opposite direction. If the target value at a new point surpasses the starting point, the steps are reduced until the desired value has been achieved.

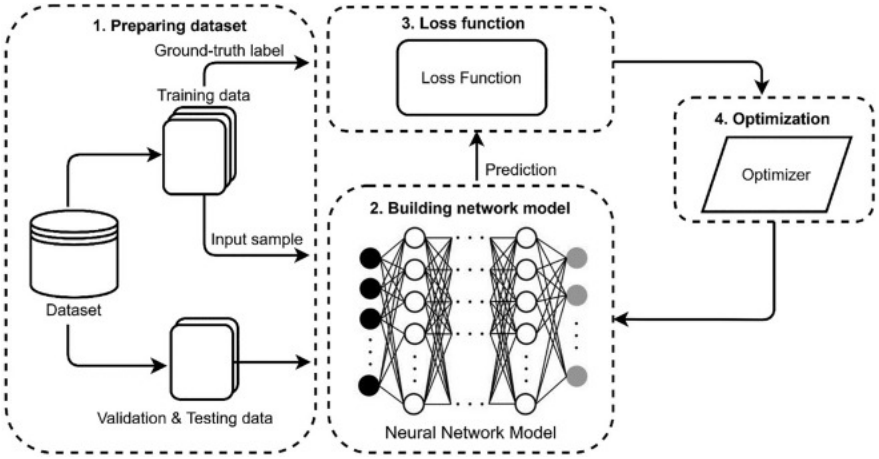


Figure 1.5: Gradient Descent Optimization Process [24]

Backpropagation [23] is another type of machine learning method that calculates the error for neurons in the last layer and propagates it backwards to the earlier layers.

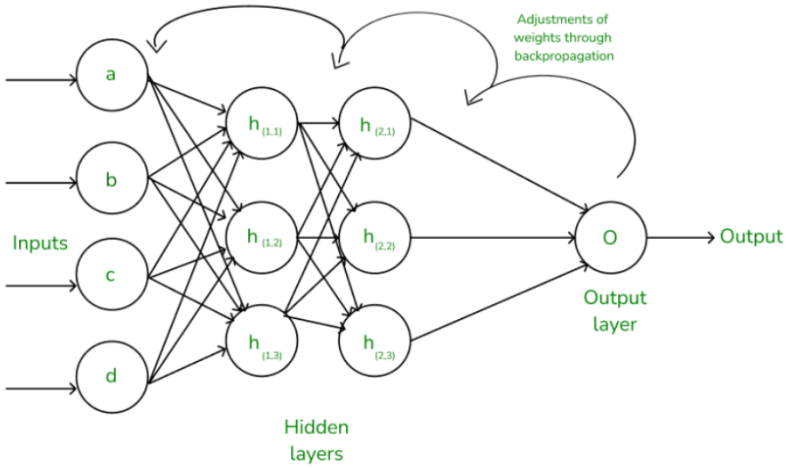


Figure 1.6: Backpropagation Process in Neural Networks. Source: GeeksforGeeks [25]

This efficient algorithm has been widely used in research. The trained network can be tested with new data to assess its performance in recognizing previously unseen information.

### 1.3.4 Activation Functions

Activation functions are crucial components of neural networks that introduce non-linearity into the system, enabling the network to learn complex patterns. They determine whether a neuron should be activated based on the weighted sum of its inputs. As shown in Figure 1.7, the most commonly used activation functions each have their own specific characteristics and applications.

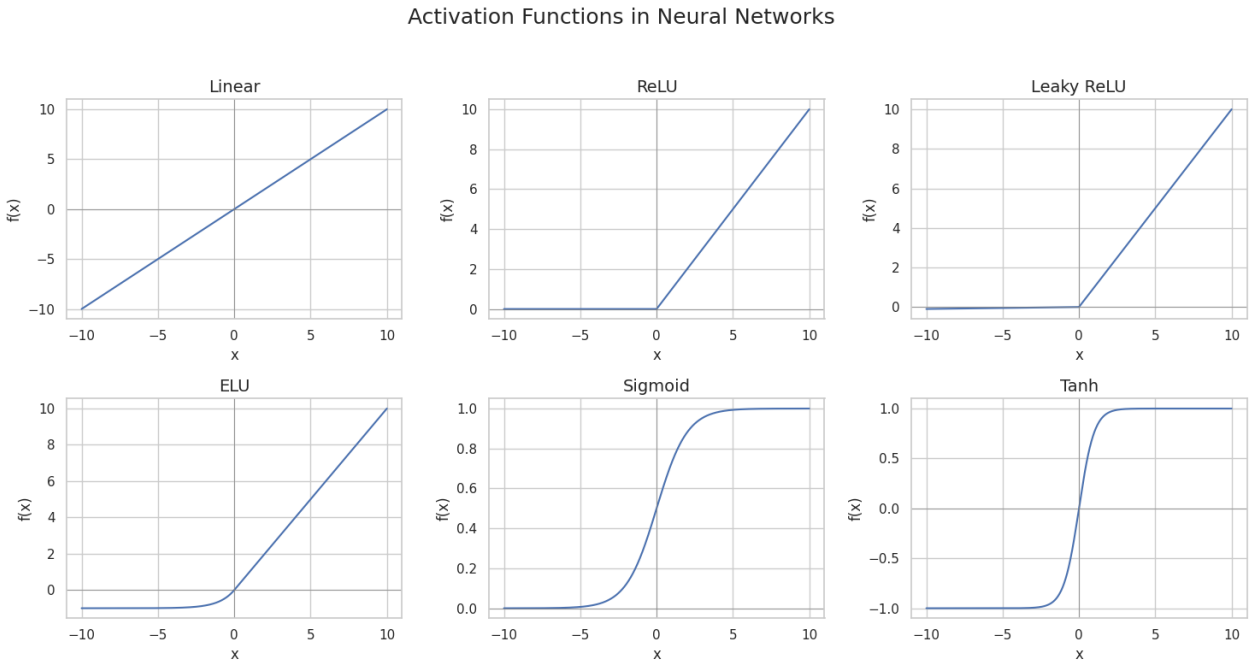


Figure 1.7: Common Activation Functions in Neural Networks

**Linear Activation Function** The linear activation function, also known as the identity function, simply passes the input value directly to the output:

$$f(x) = x \tag{1.5}$$

While simple, this function lacks the ability to learn complex patterns as it does not introduce non-linearity. It is primarily used in the output layer for regression problems.

**Rectified Linear Unit (ReLU)** ReLU is one of the most widely used activation functions due to its computational efficiency:

$$f(x) = \max(0, x) \tag{1.6}$$

It outputs 0 for negative inputs and the input value itself for positive inputs. ReLU helps mitigate the vanishing gradient problem but can suffer from "dying ReLU" where neurons can become inactive.

**Sigmoid Function** The sigmoid or logistic activation function maps input values to a range between 0 and 1:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{1.7}$$

This function is useful for binary classification problems and was historically popular. However, it suffers from the vanishing gradient problem during backpropagation.

**Hyperbolic Tangent (tanh)** The tanh function maps input values to a range between -1 and 1:

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{1.8}$$

It is similar to the sigmoid but provides stronger gradients and is zero-centered, making it easier to optimize.

**Softmax Function** The softmax function is commonly used in the output layer for multi-class classification problems:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{1.9}$$

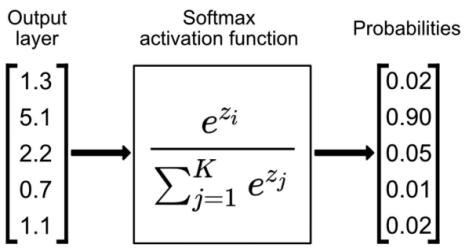


Figure 1.8: Softmax Function for Multi-class Classification

It converts a vector of real numbers into a probability distribution, with outputs summing to 1.

**Leaky ReLU** Leaky ReLU addresses the dying ReLU problem by allowing a small gradient for negative inputs:

$$f(x) = \max(\alpha x, x) \quad \text{where } \alpha \text{ is a small constant (e.g., 0.01)} \tag{1.10}$$

This prevents neurons from becoming completely inactive.

**Exponential Linear Unit (ELU)** ELU combines the benefits of ReLU while addressing the dying neuron problem:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^{x} - 1) & \text{if } x \leq 0 \end{cases} \tag{1.11}$$

It produces negative outputs, allowing for better handling of negative inputs while still preventing the vanishing gradient problem.

## 1.4 Explainable AI (XAI)

### 1.4.1 What's XAI ?

Explainable Artificial Intelligence (XAI) encompasses methodologies and techniques aimed at making AI systems more transparent, interpretable, and understandable to human users. At its core, XAI addresses the "black box" problem in modern machine learning systems (particularly deep learning models) where the internal decision-making processes remain opaque despite impressive performance metrics. As AI systems increasingly impact critical domains like healthcare, finance, and criminal justice, the ability to understand, trust, and validate their decisions becomes paramount [26].

XAI serves multiple purposes beyond mere technical transparency. It enables domain experts to verify model reasoning, helps developers debug and improve systems, assists end-users in building appropriate trust, and supports regulators in ensuring compliance with emerging AI governance frameworks. As Arrieta et al. (2020) [27] note: "Explainability becomes a prerequisite for building trust in intelligent systems, but also a way to enhance the acceptance of AI in society."

### 1.4.2 Historical Development of XAI

#### Origins and Evolution

Explainability in AI dates back to the 1970s and 1980s, when expert systems like MYCIN used transparent, rule-based logic and could justify their decisions step by step. These early systems emphasized explanation as a knowledge-sharing process between the system and the user [28, 29]. However, as machine learning gained traction in the 1990s, the focus shifted toward performance, sidelining interpretability [30].

#### Modern Resurgence

Since the resurgence of deep learning around 2012, the field has witnessed rapid advances driven by increasingly powerful—but often opaque—models. This opacity has renewed concerns about transparency and accountability, particularly in high-stakes domains like healthcare, finance, and law. In response, the European Union's General Data Protection Regulation (GDPR, 2018) introduced the "right to explanation" for individuals affected by automated decisions [31], sparking ongoing legal and ethical concerns [32]. Around the same time, DARPA launched its Explainable AI (XAI) program in 2017 to promote the development of AI systems whose decisions can be understood and trusted by humans [33]. Today, XAI stands at the intersection of machine learning, cognitive science, and human-computer interaction, aiming to produce explanations that are both technically sound and accessible to end users.

### 1.4.3 The Performance-Interpretability Trade-off

#### Understanding the Dichotomy

A persistent narrative in machine learning suggests an inherent trade-off between model performance and interpretability. This view holds that more complex models (like deep neural networks) achieve higher accuracy at the cost of reduced transparency, while simpler models (like decision trees) offer clarity but potentially sacrifice predictive power [34].

This dichotomy stems from fundamental differences in model architecture. Simple models like linear regression or decision trees make predictions through explicitly defined, human-readable structures. In contrast, deep learning models distribute knowledge across thousands or millions of parameters, creating distributed representations that resist straightforward interpretation.

However, recent research challenges the universality of this trade-off. Rudin (2019) [34] argues that in many cases, the performance gap between interpretable models and black-box systems is smaller than commonly assumed when interpretable models are properly optimized.Chen et al. (2022) [35] demonstrates that for structured data problems, carefully designed interpretable models can sometimes match or even exceed the performance of opaque alternatives.

The empirical reality appears more nuanced than a simple inverse relationship between performance and interpretability. Rather, the trade-off varies considerably across problem domains, data types, and specific tasks—suggesting opportunities for domain-specific optimization that balances both objectives.

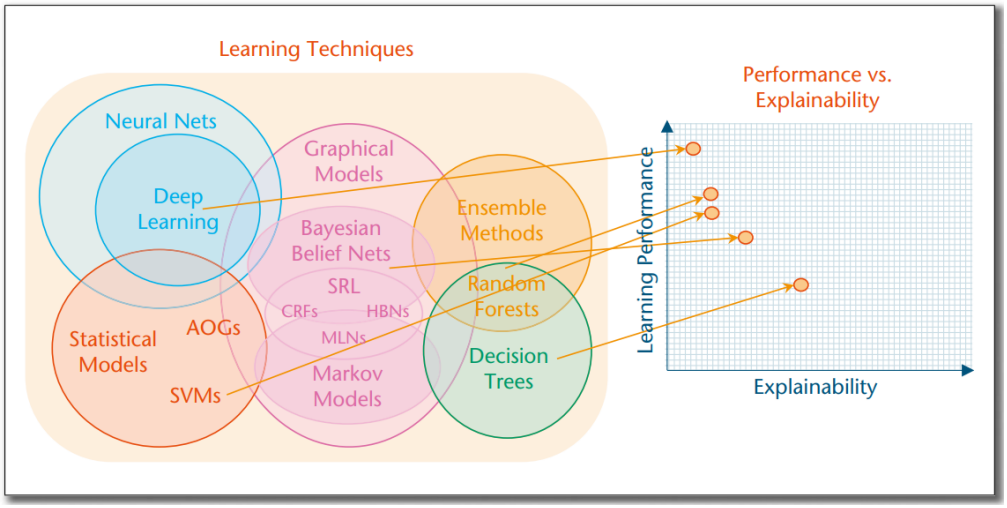


Figure 1.9: The traditional view of the performance-interpretability trade-off in machine learning models - Source : DAPRA XAI Program [36]

#### Application-Specific Considerations

The importance of explainability versus raw performance varies dramatically across application contexts. Medical diagnosis systems, financial credit scoring algorithms, and judicial

risk assessment tools operate in domains where trust, accountability, and fairness considerations may outweigh marginal performance improvements [37].

In these high-stakes domains, explainability serves multiple functions beyond user trust. It enables detection of problematic patterns such as unwanted bias, facilitates regulatory compliance, and supports effective human-AI collaboration. Conversely, applications like image recognition for consumer photo organization or recommendation systems for entertainment may prioritize performance over detailed explanations. Here, users may accept limited explainability if the system delivers superior results [30].

This application-dependent perspective suggests moving beyond the binary question of "interpretable or accurate?" toward a more nuanced approach asking "what type and degree of explainability is necessary for this specific context?" This framing acknowledges that appropriate levels of transparency depend on the stakes involved, regulatory requirements, user needs, and operational constraints.

### 1.4.4 Main Approaches in Explainable AI

XAI methods can broadly be categorized into two families:

#### 1. Model Transparency (Intrinsic Explainability)

Transparency-focused approaches aim to build models that are interpretable *by design*, rather than explaining them after training. These models are often simpler and easier to inspect directly.

- **Decision Trees**: Use a hierarchy of if-then rules, which can be visualized and easily understood by humans [34].
- **Linear and Logistic Regression**: Offer transparency through their coefficients, which directly quantify the influence of each input feature on the output [38].
- **Prototype-based Models (e.g., k-NN)**: Classify instances by comparing them with similar examples from the training set, aligning with human reasoning based on analogies [39].

These models are valuable when *interpretability is prioritized over raw performance*, especially in domains requiring human oversight and accountability.

#### 2. Post-hoc Explanations (Black-box Explainability)

Post-hoc methods aim to interpret **complex, black-box models**, such as deep neural networks, after they have been trained. They do not modify the underlying model architecture but provide insights into its behavior.

**a) Feature Attribution Methods** These techniques explain predictions by assigning importance scores to input features:

- **SHAP (SHapley Additive exPlanations)**: Uses game theory to fairly distribute the model output among the input features [40].
- **LIME (Local Interpretable Model-agnostic Explanations)**: Approximates the model locally using a simple, interpretable surrogate model to explain individual predictions [41].
- **Layer-wise Relevance Propagation (LRP)**: Decomposes the neural network's output by redistributing the prediction score layer by layer, assigning *relevance scores* to each input feature based on its contribution. LRP ensures **relevance conservation**, meaning that the sum of scores is preserved across layers [42].

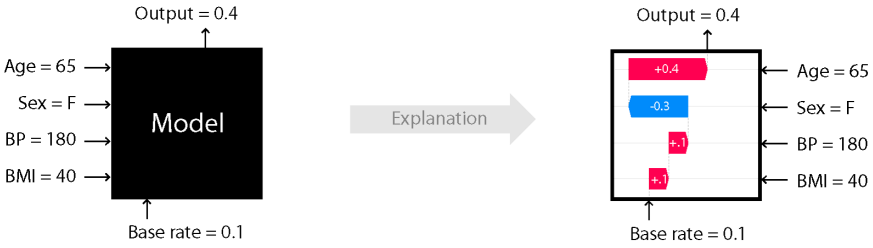


Figure 1.10: SHAP (SHapley Additive exPlanations) method

**b) Gradient-based Visual Explanations** Primarily used for image data:

- **Grad-CAM**: Produces class-specific heatmaps by leveraging the gradients of the output with respect to intermediate feature maps, showing which parts of the input image influenced the decision [43].

**c) Counterfactual Explanations** Focus on *"what-if"* scenarios:

- **Counterfactual Examples**: Identify minimal changes to input features that would alter the model's prediction, helping users understand model behavior in actionable terms [44].

**d) Example-based Explanations** These rely on **influential training examples**:

- Help explain model predictions through comparisons to similar past cases, leveraging cognitive processes based on analogical reasoning [45].

### Why Choose LRP in This Thesis ?

Among the various post-hoc explanation techniques, **Layer-wise Relevance Propagation (LRP)** was chosen as the most appropriate for this thesis for several reasons:

- **Compatibility with Neural Architectures**: LRP is specifically designed for neural networks, making it well-suited for analyzing the behavior of the multi-layer perceptron used in the Colored Petri Neural Network (CPNN) model.
- **Fine-Grained Interpretability**: Unlike global methods such as SHAP or LIME that rely on approximation, LRP provides a detailed, *layer-by-layer* explanation that is closely tied to the internal structure of the network.
- **Relevance Conservation**: The conservation principle in LRP ensures that the total relevance is preserved across layers, enhancing the reliability and trustworthiness of the explanation—a crucial requirement in domains like healthcare.
- **Integration with Formal Methods**: The structured relevance scores produced by LRP can be embedded within Colored Petri Net reachable states, allowing the tracing of explanations to be aligned with formal verification workflows.

Thus, LRP was not only technically compatible but also conceptually aligned with the thesis's goal of combining interpretability with formal verification. It serves as an effective bridge between black-box neural computation and transparent, verifiable explanations.

## 1.5 Formal Verification

In the realm of critical systems, where failure can lead to severe consequences, the need for rigorous guarantees about system behavior has led to the adoption of formal verification techniques. Unlike traditional testing, which samples a system's behavior, formal verification aims to exhaustively prove the correctness of a system with respect to a formal specification.

### 1.5.1 Definition

Formal Methods constitute a collection of mathematically rigorous techniques and tools employed for the specification (Defining what a system is supposed to do), design (Creating a formal model of the system), and verification (Ensuring that a system meets its specifications) of software and hardware systems. These methods are grounded in formal logic, set theory, and discrete mathematics to provide precision and unambiguous representation of system behaviors [46]. Unlike conventional testing approaches that can only demonstrate the presence of errors, formal methods aim to establish correctness guarantees through mathematical proof.

Formal Verification specifically refers to the process of using mathematically-based techniques to confirm that a system meets its formal specification under all possible scenarios within the defined model. The objective is to systematically establish that an implementation adheres to its intended specification through rigorous mathematical analysis rather than empirical testing [47]. This approach provides stronger assurances about system correctness, particularly for safety-critical applications where failures could have severe consequences.

### 1.5.2 Overview of Existing Techniques

Formal verification encompasses several complementary approaches, each with distinct strengths and application domains:

- **Theorem Proving** involves constructing mathematical proofs of system correctness using logical inference rules and axioms. Modern theorem provers like Coq, Isabelle/HOL, [48] and PVS provide interactive environments where human guidance directs the proof development while the system ensures logical consistency. This approach offers exceptional expressive power but often requires significant expertise and manual effort [49].

- **Model Checking** automates the verification process by exhaustively exploring all possible states of a finite-state representation of the system. The technique algorithmically determines whether specified properties hold across the entire state space, providing counterexamples when violations are detected. This automation makes model checking particularly accessible for industrial applications [50]. *Petri Nets*, and especially Colored Petri Nets (CPNs), are widely used as modeling formalisms in model checking due to their expressiveness in capturing concurrent, distributed, and data-dependent behaviors. They enable a structured representation of system states and transitions, making them particularly suited for verifying neural network behavior, protocol correctness, and resource allocation properties.

- **Abstract Interpretation** employs sound approximations of program semantics to verify properties without exploring the complete concrete state space. By mapping concrete states to abstract domains, this technique can verify invariant properties even for infinite-state systems, though with potential loss of precision. The approach has proven effective for detecting runtime errors in large-scale software [51].

- **Symbolic Execution** [52] systematically explores program paths by representing inputs symbolically rather than with concrete values. This technique enables reasoning about multiple execution paths simultaneously and has shown success in detecting subtle software errors and security vulnerabilities.

While each approach has distinct characteristics, industrial adoption has particularly embraced model checking due to its automation capabilities and pragmatic balance between expressiveness and computational feasibility.

| Formal Method | Description | Key Strengths | Common Applications |
|---|---|---|---|
| **Theorem Proving** | Constructs mathematical proofs to verify system correctness. | • High expressiveness<br>• Guarantees correctness with rigorous proofs | • Safety-critical systems (e.g., aviation, automotive)<br>• Cryptographic protocols |
| **Model Checking** | Exhaustively explores system states to check if properties hold. | • Automation<br>• Handles large state spaces efficiently<br>• Provides counterexamples | • Hardware verification<br>• Protocol validation (e.g., network protocols) |
| **Abstract Interpretation** | Uses approximations to verify properties without complete state exploration. | • Handles infinite state spaces<br>• Detects runtime errors | • Static analysis of software<br>• Verification of large systems |
| **Symbolic Execution** | Explores program paths symbolically to detect errors and vulnerabilities. | • Can explore multiple paths simultaneously<br>• Detects subtle software errors and vulnerabilities | • Software security<br>• Bug finding in complex systems |

Table 1.2: Comparison of Formal Methods

## 1.6 Petri Nets

Petri nets represent a powerful mathematical and graphical modeling formalism for describing and analyzing concurrent, asynchronous, distributed, parallel, nondeterministic, and stochastic systems. Developed by Carl Adam Petri in his 1962 dissertation [53] , these nets have evolved into a comprehensive framework with extensive theoretical foundations and practical applications across diverse domains .

The distinctive feature of Petri nets lies in their dual nature—combining graphical representation with formal mathematical semantics [54]. This duality facilitates both intuitive understanding and rigorous analysis, making them accessible to practitioners while ensuring mathematical soundness for theoretical investigation.

### 1.6.1 Basic Structure

A Petri net is formally defined as a tuple PN = (P, T, F, W, $M_0$) [55] where:

- P = {$p_1$, $p_2$, ..., $p_m$} is a finite set of places
- T = {$t_1$, $t_2$, ..., $t_n$} is a finite set of transitions
- F $\subseteq$ (P $\times$ T) $\cup$ (T $\times$ P) is the flow relation representing arcs
- W: F $\to \mathbb{N}_+$ is the weight function assigning positive integers to arcs
- $M_0$: P $\to \mathbb{N}$ is the initial marking (initial distribution of tokens)

The sets P and T are disjoint (P $\cap$ T = $\emptyset$), representing the distinction between states (places) and events (transitions).

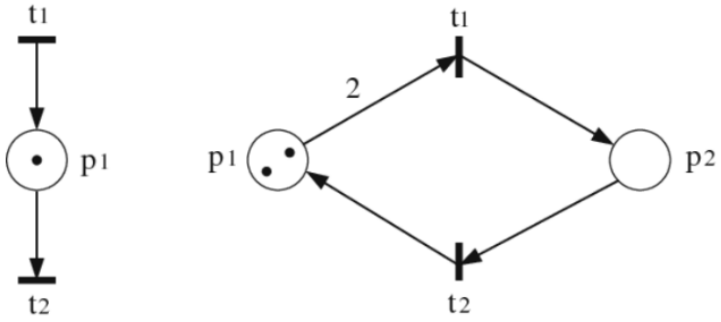


Figure 1.11: Illustration of basic Petri nets: (left) a sequential firing pattern; (right) a loop with token consumption and production

Graphically, places are depicted as circles, transitions as rectangles or bars, and the flow relation as directed arcs connecting them. Tokens, represented as dots within places, indicate the current state of the system [56].

### 1.6.2 Dynamic Behavior

The execution semantics of Petri nets are governed by the firing rule, which defines how transitions change the system state:

- A transition t is enabled when each input place p contains at least W(p,t) tokens.
- An enabled transition may fire, consuming W(p,t) tokens from each input place p.
- When a transition fires, it produces W(t,p) tokens in each output place p.

This firing mechanism creates a state space of possible markings (distributions of tokens) reachable from the initial marking. The nondeterministic nature of transition firing—where any enabled transition may fire at any time—captures the essence of concurrent behavior, where the exact ordering of events may vary across different executions [57].

### 1.6.3 Mathematical Representation

Petri net dynamics can be expressed mathematically using the state equation:

$$M' = M + C \cdot u \tag{1.12}$$

Where:

- M is the current marking (represented as an m×1, m = |P|, vector)
- M' is the resulting marking after firing
- C is the incidence matrix (an m×n, n = |T|, matrix) where C[i,j] = W($t_j$,$p_i$) - W($p_i$,$t_j$)
- u is the firing vector (an n×1 vector) indicating which transitions fire

This algebraic representation facilitates analytical techniques for studying Petri net properties, particularly for structural analysis methods that examine the incidence matrix independent of specific markings.

### 1.6.4 Modeling Power

Petri nets offer a rich set of modeling capabilities that make them suitable for representing complex dynamic systems, particularly those involving concurrency, synchronization, and resource sharing [55], the expressive power of Petri nets lies in their ability to capture a variety of behavioral patterns observed in real-world systems:

- **Sequential Execution**: A transition can only fire after another, imposing precedence constraints typical in operation scheduling and modeling causal relationships between activities.
- **Conflict**: Multiple simultaneously enabled transitions where firing one disables the others. This situation represents choices between alternatives, resolvable non-deterministically or probabilistically.

- **Concurrency**: Independent transitions can activate in parallel, representing processes executing simultaneously without interference, an essential characteristic of distributed systems.
- **Synchronization**: A transition requiring tokens in all its input places models resource or process synchronization, capturing rendezvous points between parallel activities.
- **Mutual Exclusion**: Structure representing processes that cannot execute simultaneously due to shared resources, such as a robot serving multiple machines.
- **Priorities**: Implemented via inhibitor arcs, allowing precedence relationships between transitions, essential for modeling systems with priority orders.

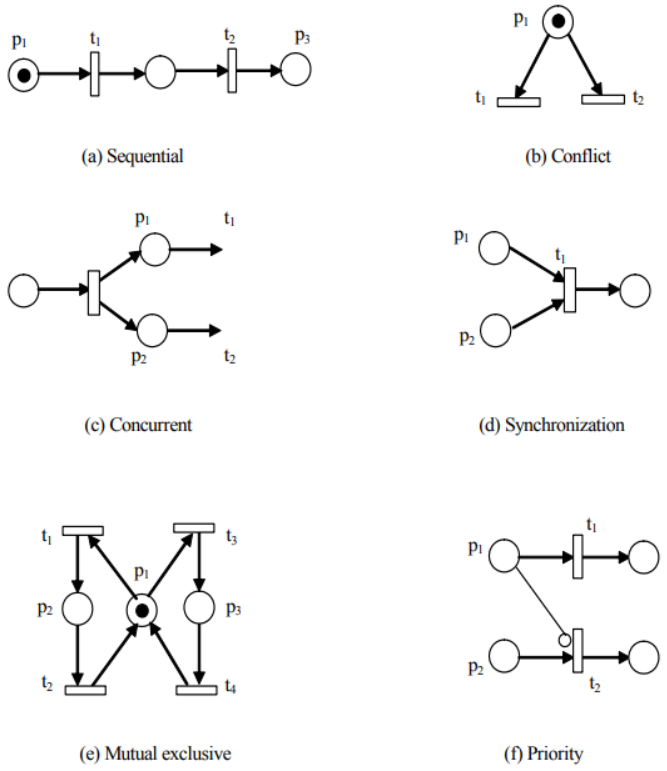


Figure 1.12: Representing complex dynamic systems with Petri Nets

### 1.6.5 Behavioral Properties

Petri nets exhibit several important behavioral properties [55] that characterize system dynamics:

- **Reachability**: A marking M' is reachable from M if there exists a sequence of transition firings that transforms M into M'. The reachability problem—determining whether a specific marking is reachable—is central to many verification questions.

- **Boundedness**: A place is k-bounded if the number of tokens it contains never exceeds k for any reachable marking. A Petri net is bounded if all its places are bounded, indicating finite state space.
- **Liveness**: A transition is live if, from any reachable marking, it can eventually become enabled through some firing sequence. Different levels of liveness characterize the potential for deadlock-free operation.
- **Deadlock-freedom**: A marking is deadlocked if no transition is enabled. Deadlock-freedom ensures the system can always progress from any reachable state.
- **Reversibility**: A Petri net is reversible if the initial marking is reachable from any reachable marking, indicating the system can always return to its starting state [58].

### 1.6.6 Analysis Methods

The analysis of Petri nets encompasses several complementary approaches:

- **Reachability Graph Analysis**: Constructing the full state space by generating all reachable markings and transitions between them. While comprehensive, this approach suffers from state explosion for complex systems.
- **Simulation**: Executing the Petri net model through random or guided firing sequences to explore system behavior and detect potential issues.
- **Model Checking**: Verifying whether specific temporal logic properties hold across all reachable states, often using specialized algorithms for Petri net state spaces.
- **Place and Transition Invariants**: Invariant analysis provides powerful structural insights into Petri net behavior, it can establish boundedness, detect structural deadlocks, and identify subsystems with independent behavior, often without generating the complete state space [56]:
  - **Place Invariants (P-invariants)**: Vectors x that satisfy $x^T \cdot C = 0$, representing weighted sets of places whose token sum remains constant regardless of transition firings. These invariants identify conservation laws in the system.
  - **Transition Invariants (T-invariants)**: Vectors y that satisfy $C \cdot y = 0$, representing transition firing sequences that return the net to its original marking. These invariants identify potential cycles or steady-state behaviors.

### 1.6.7 Extensions and Variants

#### Colored Petri Nets (CPNs)

Colored Petri Nets (CPNs) extend the basic model by attaching data values ("colors") to tokens and defining transition guard conditions and arc expressions. This extension significantly enhances modeling power by incorporating data manipulation while maintaining the fundamental concurrency semantics. CPNs facilitate compact representation of complex

systems by distinguishing between different token types and specifying transformation rules [59].

### Timed Petri Nets (TPNs)

Timed Petri Nets [60] are an extension of classical Petri nets that incorporate the concept of time into transitions. In a standard Petri net, a transition fires as soon as it's enabled. But in a TPN, each transition is associated with a time delay, meaning it only fires after a specified amount of time has passed since it became enabled.

This addition is particularly useful for modeling systems where timing matters, such as real-time controllers, hardware circuits, or neural networks where computations may be layered or time-dependent.

### Hierarchical Petri Nets (HPNs)

Hierarchical Petri Nets [61] are designed to handle the complexity of large systems by supporting abstraction and modularity. They allow you to group a subset of places and transitions into a submodel (or "subnet"), which can be collapsed into a single node in a higher-level net. This way, large systems can be modeled in a top-down fashion, making them easier to manage, visualize, and analyze.

## Conclusion

This chapter outlined the essential concepts of machine learning, neural networks, explainable AI, formal verification, and Petri nets. These foundations provide the necessary background for the methods and model developed in the following chapters.

# Chapter 2

# Existing Approaches to Modeling Neural Networks with Petri Nets

## Introduction

Neural networks have proven effective in modeling complex systems, including those found in critical industries such as drilling operations. However, their black-box nature poses a challenge for applications that demand transparency, interpretability, and formal verification. Colored Petri Nets (CPNs) offer a promising formalism for modeling the structure and behavior of neural networks, enabling both simulation and verification. This chapter presents and analyzes three major papers that propose methods for modeling neural networks using Colored Petri Net formalisms, with varying objectives and outcomes. Each method is reviewed in terms of its methodology, contributions, strengths, and weaknesses.

## 2.1 Key Papers and Their Contributions

### 2.1.1 Paper 1: Simulating Artificial Neural Network Using Hierarchical Coloured Petri Nets (Jitmit & Vatanawood, 2021)

Jitmit and Vatanawood propose a systematic methodology [62] to convert a trained artificial neural network (ANN) into a modular, reusable model using Hierarchical Coloured Petri Nets (HCPNs). Their primary goal is to enhance the formal verification, modularity, and reusability of ANNs within larger symbolic systems, leveraging the structure and semantics of Petri Nets to encapsulate neural behavior in a formally analyzable framework. This paper [62]contributes to the growing intersection between artificial intelligence and formal modeling, offering a visual and structural alternative to purely numerical representations.

#### Methodology and Core Contributions

At the core of their work lies the definition of a formal 7-tuple ANN model that includes inputs, hidden and output neurons, arcs, weights, biases, and activation functions. This ANN is mapped onto a hierarchical CPN through a set of four conversion rules:

Rule 1: Maps the ANN into a Level 0 HCPN module with input and output ports.

Rule 2 & 3: Converts hidden and output neurons into Level 1 modules, with places representing intermediary states (buffers).

Rule 4: Decomposes each neuron into Level 2 submodules with transitions modeling (1) weighted summation and bias addition, and (2) the application of an activation function (specifically, a step function with a 0 threshold).

Each neuron is implemented as a black-box module, which encapsulates its functionality while exposing input and output interfaces through ports and sockets. The modularity of the approach allows for clean composition of neurons into layers and networks. A case study demonstrates the successful conversion and simulation of a neural network solving the classic XOR problem, with correct propagation and activation behavior observed in CPN Tools.

**Strengths**

- **Systematic Conversion Framework** by offering clearly defined transformation rules, the authors ensure that any feedforward ANN can, in principle, be systematically converted into an HCPN structure.
- **Clean Modularization** through the hierarchical decomposition of neurons supports reusability and clarity—crucial for scaling or integrating within larger symbolic systems and it facilitates simulation and verification in CPN Tools.
- Demonstrated correctness through a working XOR example.

**Weaknesses and Critical Reflection**

- **No Support for Learning:** The model is entirely static. It assumes that training has already been performed externally. There is no modeling of weight adaptation, learning dynamics, or error-driven updates—key elements in any realistic neural model. There is no discussion of performance metrics, error margins, or robustness of the CPN-based model when compared to its ANN counterpart. This sharply limits its utility to educational or symbolic representation contexts.
- **Minimal Interpretability Beyond Structure:** Although the HCPN representation clarifies the flow of computation, it offers no embedded mechanisms for explanation or interpretation of decisions beyond simply visualizing the network's structure.
- **Scalability Not Addressed:** While modular, the model is only demonstrated on the XOR problem—arguably the simplest non-linearly separable task. The computational and visual complexity of manually constructing and simulating deeper networks with real-world data remains unaddressed.
- **No Integration with Real Datasets or Pipelines:** All ANN parameters (weights, biases) must be inserted manually or via a pre-defined XML schema. There is no support for importing model parameters directly from common ML libraries (e.g., PyTorch, TensorFlow), nor any tooling for automatic conversion.

### 2.1.2 Paper 2: Coloured Petri Nets Modeling Multilayer Perceptron Neural Networks (Oliveira et al., 2023)

In this paper [63] , Oliveira et al. present a novel method for modeling multilayer perceptron (MLP) neural networks using Coloured Petri Nets (CPNs), with a clear focus on improving interpretability and transparency for critical applications—most notably in the healthcare sector. Unlike many existing works that treat neural networks as black boxes, this approach introduces a formal and visual methodology that allows for tracing the propagation of input influences through the layers of an MLP. The model notably stops short of implementing learning or training dynamics, prioritizing instead the representation and interpretability of already-trained models.

#### Methodology and Technical Contributions

The authors adopt a hierarchical CPN structure, where each neuron is implemented as a modular submodel using substitution transitions. Inputs and weights are encoded as structured color sets, and neural operations (such as weighted sums) are defined using recursive ML functions. The principal innovation of the work [63] is the introduction of a relevance matrix, a dynamic data structure that tracks the influence of each input neuron on each subsequent neuron in the network across layers. This matrix is updated during forward propagation to reflect the cumulative weight impact of inputs, allowing developers to visualize and quantify how each input contributes to final outputs.

In practical terms, the model is validated using a COVID-19 test prioritization dataset from Brazil. An MLP is trained using scikit-learn, and its structure (weights and biases) is manually transposed into the CPN model. The outputs of both models are then compared, demonstrating a high degree of alignment, thus confirming the functional equivalence of the CPN-based representation.

#### Strengths

This work stands out for its concrete and original attempt to bridge the gap between symbolic modeling and neural network behavior representation. Key strengths include:

- **Interpretability through Relevance Tracing:** The use of a relevance matrix to visualize and propagate input influence provides a compelling mechanism for understanding the internal decision-making of an MLP. This can serve as a valuable tool for transparency in high-stakes decision systems, aligning with principles of explainable AI (XAI).
- **Hierarchical and Modular Structure:** By modeling each neuron and layer hierarchically, the authors ensure extensibility and potential reusability across different network architectures.
- **Validation via Output Comparison:** The model is thoroughly validated by comparing its predictions against those of a conventional scikit-learn implementation, lending confidence in its correctness.

- **Domain Applicability:** The healthcare use case illustrates the model's practical relevance, especially where interpretability and formal verification are vital.

**Limitations and Critical Analysis**

- **Lack of Learning or Adaptation:** The model does not support training or weight updates. All weights and biases must be imported from an external system, severely limiting the model's ability to adapt or improve. This omission means the model cannot simulate or analyze learning dynamics—a key limitation in contexts requiring full-cycle neural modeling.
- **Manual Parameter Transfer:** The process of exporting trained weights from scikit-learn and inputting them manually into the CPN model is both error-prone and unscalable. Although the authors suggest future work involving automated XML-based CPN generation, no implementation or prototype of this functionality is yet presented.
- **Scalability Concerns:** While the model is conceptually extensible to larger architectures, the practical visual and computational complexity of tracking relevance matrices across many layers and neurons could become prohibitive. The model shown includes only three hidden layers, each with three neurons—minimal by modern deep learning standards.

### 2.1.3 Paper 3: A Novel Fully Adaptive Neural Network Modeling and Implementation Using Colored Petri Nets (Albuquerque et al., 2023)

In their recent contribution, Albuquerque et al. [64] introduce HTCPN-MLP, a Hierarchical Timed Colored Petri Net (HTCPN) model of a multilayer perceptron that aims to replicate the complete learning dynamics of a neural network, including weight updates via the backpropagation algorithm. This work represents a notable milestone in the formal modeling of neural networks using high-level Petri nets, bridging discrete event systems with adaptive learning paradigms. By leveraging the expressiveness of HTCPNs, the authors manage to encapsulate all essential learning phases—data ingestion, forward propagation, backpropagation, and validation/testing—within modular, visually interpretable subnets.

**Methodological Innovations**

The model [64] is built bottom-up, starting from a CPN-based implementation of the McCulloch-Pitts neuron, which is then extended to a perceptron layer with error-driven learning. These components serve as the foundation for the final HTCPN-MLP model that incorporates a full feedforward and backward training cycle using a tanh activation function.

Notably, time constraints are used to emulate epochs in training, and modular subnets are designed for training, validation, and testing. The authors implement all computations—activations, gradients, and weight updates—using Standard ML (SML) functions inside the CPN Tools framework, showcasing the technical feasibility of full-fledged neural training in a symbolic and formal model.

### Contributions and Strengths

The most significant contribution of this work lies in its completeness: it is arguably the first attempt to implement a fully adaptive, trainable MLP using Colored Petri Nets, rather than merely illustrating structure or flow. The use of time-annotated tokens allows modeling sequential processes like epochs, and the modularity of their HTCPN design allows for reuse and potential scaling. The validation on six benchmark datasets, including Iris, Wine, Cardiotocography, and Seismic-bumps, demonstrates the model's applicability to real-world, diverse tasks, and its performance is shown to be on par with a standard MLP implemented in MATLAB.

Other merits include:

- A rigorous and layered model construction, from single neuron to full MLP.
- Explicit coding of neural dynamics using Standard ML functions.
- Support for both classification and regression tasks.

### Limitations and Critical Evaluation

- **Scalability:** While the authors claim modularity and generality, their model is visually and computationally heavy, even for relatively small datasets. Scaling it to deeper architectures or real-time applications would likely be impractical without substantial abstraction or automation layers.
- **Limited Architecture:** The network modeled contains only one hidden layer, which, although pedagogically convenient, is not sufficient for more complex real-world problems requiring deep learning. A growing network is mentioned as future work but not yet realized.
- **Genericity vs. Usability:** While the model is declared "general" and capable of handling arbitrary datasets, in practice, each change in architecture or data format still requires manual intervention. Thus, full generality remains more theoretical than applied.

## Conclusion

These three approaches illustrate the evolving capacity of Colored Petri Nets to model, simulate, and verify neural network architectures. While the first method offers modularity and reuse for already-trained networks, the second enhances interpretability and visualization. The third approach stands out for incorporating full learning dynamics within a formal framework. Together, they provide a foundation for future work, including our aim to develop a formal, interpretable, and adaptive modeling.

| Method | Strengths | Weaknesses |
|---|---|---|
| **Paper 1: Simulating Artificial Neural Network Using Hierarchical Coloured Petri Nets (Jitmit & Vatanawood, 2021)** | • Clear modularization for reuse.<br>• Provides a systematic conversion framework.<br>• Demonstrated with XOR case study. | • No support for learning; static model.<br>• Minimal interpretability beyond structure.<br>• Scalability issues with larger networks.<br>• No integration with real datasets. |
| **Paper 2: Coloured Petri Nets Modeling Multilayer Perceptron Neural Networks (Oliveira et al., 2023)** | • Relevance tracing improves interpretability.<br>• Hierarchical and modular structure.<br>• Validated with a real dataset (COVID-19). | • No support for training or weight updates.<br>• Manual parameter transfer is error-prone.<br>• Scalability issues with larger architectures.<br>• No support for full-cycle neural modeling. |
| **Paper 3: A Novel Fully Adaptive Neural Network Modeling and Implementation Using Colored Petri Nets (Albuquerque et al., 2023)** | • Complete and adaptive MLP modeling.<br>• Validated with multiple benchmark datasets.<br>• Support for both classification and regression. | • High computational cost and scalability issues.<br>• Model only includes one hidden layer.<br>• Lack of formal verification for learning correctness.<br>• No explicit explainability techniques. |
| **Our Proposal: CPNN for Neural Networks with Petri Nets** | • Full learning cycle with backpropagation.<br>• Formal and visual representation.<br>• Integrates explainability with formal methods.<br>• Scalable with more hidden layers. | • Computational complexity in larger networks.<br>• Lack of automatic model parameter import.<br>• Requires further validation with other types of NN like CNN, RNN, etc. |

Table 2.1: Comparison of Existing Approaches to Modeling Neural Networks with Petri Nets

# Chapter 3

# A New Approach of Modeling Neural Network Using Colored Petri Nets

## Introduction

In this chapter, we present our approach to modeling the training and evaluation of a feedforward neural network using Colored Petri Nets (CPNs), implemented in CPN Tools. This model provides a formal, visual, and verifiable representation of the learning process, offering transparency into every step, from input processing to weight updates and performance evaluation.

The integration of neural networks with Colored Petri Nets offers several advantages:

- Formal representation of complex neural network architectures.
- Clear visualization of data flow and computational processes.
- Ability to verify structural properties and detect potential issues.
- Mathematical foundation for analyzing network behavior.
- Unified framework for representing both structure and functionality.

However, This work establishes the foundation for our ultimate goal: explaining neural network behavior through model checking and features importance tracking within the CPN framework.

## 3.1 Neural Network Architecture

We modeled a three-layer feedforward neural network with the following structure:

- **Input Layer ($L_0$)**: 3 binary features (0 or 1).
- **Hidden Layer 1 ($L_1$)**: 2 neurons with ReLU activation.
- **Hidden Layer 2 ($L_2$)**: 2 neurons with ReLU activation.

- **Output Layer** ($L_3$): 1 neuron with Sigmoid activation for binary classification.

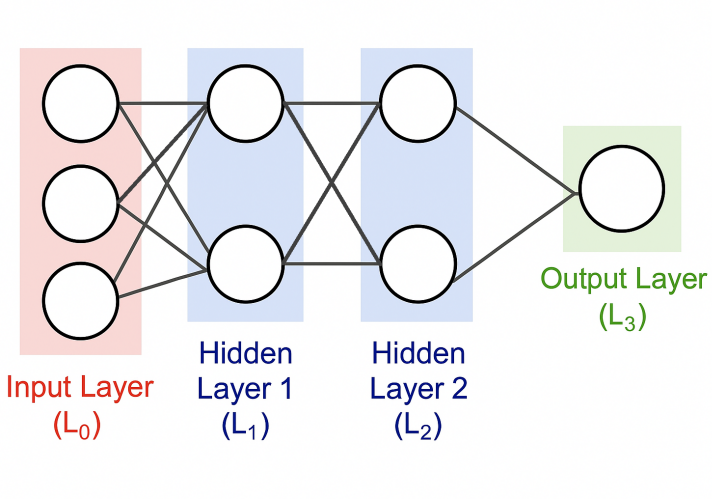


Figure 3.1: The Neural Network Architecture

Each layer is formally modeled with tokens representing activations, weights, and biases. Transitions represent the functional transformations (e.g., weighted sum, activation, error computation), and arcs implement the data dependencies and flow. This modular architecture is visualized in Figure 3.1. To support clarity and traceability, Table 3.1 provides a description of the role and function of each place and transition in the model. This structural mapping is essential for understanding how data and computations propagate across the network during both forward and backward passes.

| Place/Transition | Colorset | Description | Connected To | Role |
|---|---|---|---|---|
| Data (P) | Dataset | Contains the full dataset with IDs and features | [Load] | Forward |
| Load (T) | UNIT | Triggers splitting dataset into batches | [inputs, A0, ControlUpdate, NBepoch] | Control |
| Inputs (P) | Inputs | Input data features | [Sum1] | Forward |
| A0 (P) | Inputs | Stores batch input samples | [Update1st] | Forward |
| WB1 (P) | WB | Weight matrix for Layer 1 | [Sum1, Update1st] | Forward |
| Sum1 (T) | – | Computes weighted sum for Layer 1 | [H1] | Forward |
| H1 (P) | Inputs | Stores pre-activations for Layer 1 | [Actv1] | Forward |
| Actv1 (T) | – | Applies Activation Function to Layer 1 | [h1] | Forward |
| h1 (P) | Inputs | Activated outputs of Layer 1 | [Sum2] | Forward |
| WB2 (P) | WB | Weight matrix for Layer 2 | [Sum2, Update2nd] | Forward |
| Sum2 (T) | – | Computes weighted sum for Layer 2 | [H2] | Forward |
| H2 (P) | Inputs | Pre-activations of Layer 2 | [Actv2] | Forward |
| Actv2 (T) | – | Applies Activation Function to Layer 2 | [h2] | Forward |
| h2 (P) | Inputs | Activated outputs of Layer 2 | [Sum3] | Forward |
| WB3 (P) | WB | Weight matrix for Output Layer | [Sum3, Update3rd] | Forward |
| Sum3 (T) | – | Computes weighted sum and applies sigmoid | [H3] | Forward |
| H3 (P) | Outputs | Output predictions | [Actv3] | Forward |
| Actv3 (T) | – | Applies sigmoid | [Output] | Forward |
| Output (P) | Outputs | Final predictions | [LossFun] | Forward |
| True Labels (P) | Labels | Ground truth labels | [LossFun, Error] | Backward |
| LossFun (T) | – | Computes loss from predictions | [Acc, Error] | Backward |
| Acc (P) | REAL | Tracks accuracy on current batch | [] | Forward |
| Error (P) | VectReal | Prediction error (delta3) | [Update3rd] | Backward |
| Delta2 (T) | – | Error propagated to Layer 2 | [Delta2] | Backward |
| Delta2 (P) | Matrix | Error propagated to Layer 2 | [Update2nd] | Backward |
| Delta1 (T) | – | Error propagated to Layer 1 | [Delta1] | Backward |
| Delta1 (P) | Matrix | Error propagated to Layer 1 | [Update1st] | Backward |
| W2Delta (P) | Matrix | Weighted delta from Output to L2 | [Delta2] | Backward |
| W1Delta (P) | Matrix | Weighted delta from L2 to L1 | [Delta1] | Backward |
| A2 (P) | Matrix | Activations of Layer 2 | [Delta2] | Backward |
| A1 (P) | Matrix | Activations of Layer 1 | [Delta1] | Backward |
| Actv'2 (P) | Matrix | Activation's derivations of Layer 2 | [Update3rd] | Backward |
| Actv'1 (P) | Matrix | Activation's derivations of Layer 1 | [Update2nd] | Backward |
| Update3rd (T) | – | Updates WB3 weights | [WB3] | Backward |
| Update2nd (T) | – | Updates WB2 weights | [WB2] | Backward |
| Update1st (T) | – | Updates WB1 weights | [WB1] | Backward |
| ControlBatch (P) | INTEGER | Manages minibatch updates | [Sum1] | Control |
| NBEpoch (P) | INTEGER | Stores the number of epochs | [Load] | Control |
| CountEpoch (P) | INTEGER | Tracks number of completed epochs | [LossFun, Update3rd] | Control |
| ControlUpdate (P) | UNIT | Controls new epoch initiation | [Load] | Control |

Table 3.1: Description of Places and Transitions in the CPN Model

*Note: See Appendix B for complete color set definitions used in the model.*

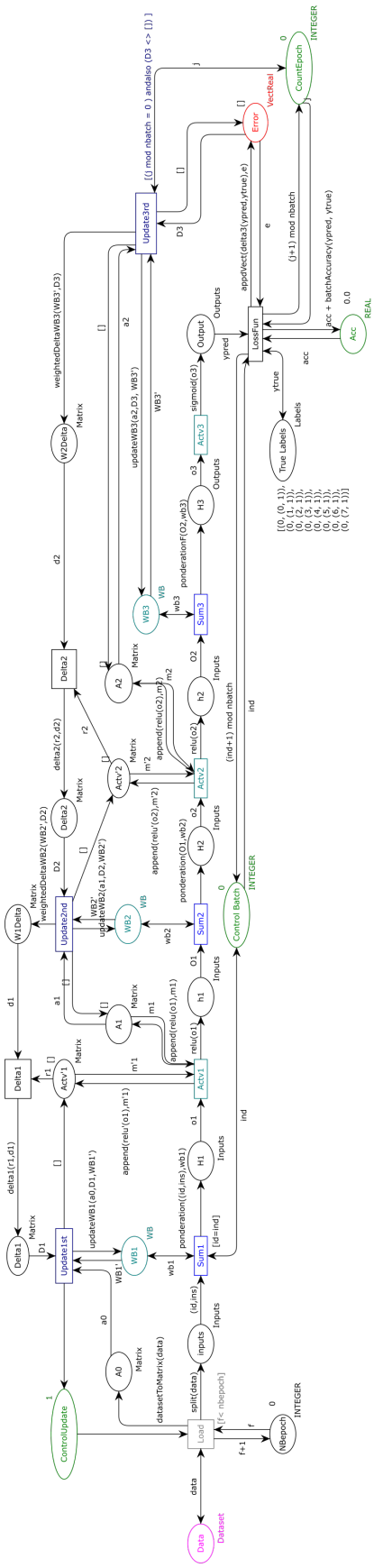


**Figure 3.1.** Complete CPN model showing forward and backward propagation paths

This three-layer structure lays the foundation for a **generalized, modular architecture** that can be extended to neural networks with **any number of hidden layers**. Each layer in the current model is implemented as a repeatable block consisting of a standardized set of CPN components: places for inputs, weights, and activations, and transitions for linear transformation, non-linear activation, and error propagation. By decoupling the operations per layer and ensuring all data dependencies are passed through tokens, we can simply replicate the structural pattern of a hidden layer to introduce further depth into the network.

Notably, this represents, to our knowledge, the **first Colored Petri Net-based implementation of a neural network with more than one hidden layer**, moving beyond simple perceptron or single-layer MLP models typically found in the literature, and integrating the entire learning algorithm from forward propagation through backpropagation. Furthermore, We analyze how the number of hidden layers, epochs, and mini-batch sizes affect model complexity. The model preserves semantic clarity and execution traceability even as depth increases, thus demonstrating that **CPNs can scale structurally with the complexity of modern neural network topologies**. Each additional layer simply involves extending the weight structures (with generalized color sets) and maintaining the forward and backward propagation sequences already established in the model.

## 3.2 Formalization with Colored Petri Nets

In our approach, we establish the following mapping between the components of the neural network and the elements of the CPN:

Each layer in the neural network Fig. 3.2 is modeled using a place to hold the inputs (for input layer) or activations (for hidden layers) and a corresponding transition to compute the transformation it applies.

The input processing begins with the `Data` place containing input samples. The `Load` transition extracts individual samples and forwards them to the first layer.

For each layer $L_i$ in the neural network, the forward propagation process is decomposed into two core operations, represented by two successive transitions:

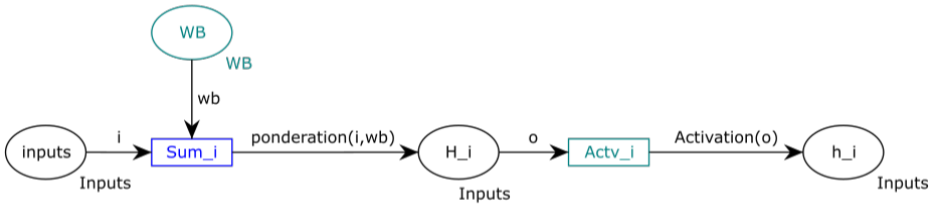


Figure 3.2: A Layer in CPN Tools

### 3.2.1 Weighted Sum (Ponderation) — Transition $Sum_i$

This transition computes the pre-activation outputs for each neuron in layer $L_i$ as a weighted sum of the activations from the previous layer $L_{i-1}$, plus a bias term.

Formally, for an input vector $\mathbf{X}^{(i-1)} \in \mathbb{R}^n$, and a weight matrix $\mathbf{W}^{(i)} \in \mathbb{R}^{(n+1)\times m}$ (including bias as the $(n+1)^{\text{th}}$ input), the output of the ponderation step is:

$$\mathbf{z}^{(i)} = \mathbf{W}^{(i)\top} \cdot \begin{bmatrix} \mathbf{X}^{(i-1)} \\ 1 \end{bmatrix} \tag{3.1}$$

In the CPN model:

- The place `Inputs` (or $h_{i-1}$) contains tokens of type `Inputs`, holding the vector $\mathbf{X}^{(i-1)}$.
- The place `WB_i` holds the current weight matrix $\mathbf{W}^{(i)}$, typed as `WB`.
- The transition `Sum_i` consumes both tokens and applies the function:

$$\texttt{ponderation}(\mathbf{X}^{(i-1)}, \mathrm{WB}_i) = \mathbf{z}^{(i)} \tag{3.2}$$

- The result is stored in place $\texttt{H}_i$.

### 3.2.2 Activation Function — Transition $Actv_i$

This transition applies a non-linear activation function element-wise to each component of $\mathbf{z}^{(i)}$ to obtain the activations of the current layer:

$$\mathbf{a}^{(i)} = f^{(i)}\left(\mathbf{z}^{(i)}\right) \tag{3.3}$$

The activation function $f^{(i)}$ is:

- ReLU for $i = 1, 2$.
- Sigmoid for $i = 3$.

The transition `Actv_i` applies this function to the contents of $\texttt{H}_i$ and produces the activated vector in $h_i$.

### 3.2.3 Modular Propagation Pattern

This pattern — $Sum_i$ followed by $Actv_i$ — is replicated for each layer $i$, enabling a modular and scalable modeling of forward propagation across arbitrarily deep feedforward neural networks.

The data flow is implemented using arcs that link transitions to places. The weights connecting neurons are defined as structured color sets (`WB`) which store vectors or matrices of real numbers (i.e., `colset WB = list VectReal`), enabling dynamic updating during training. The model also accommodates the accumulation of activations for batch processing by appending them to matrices during forward propagation.

By organizing each layer as a modular unit with consistent internal logic (places for inputs and activations, transitions for ponderation and activation), we can reproduce the full behavior of a neural network in a visually and formally tractable CPN model.

## 3.3 Color Sets, Data Types and Functions Definition

The CPN model employs a comprehensive type system to represent the various data elements used in neural networks. Since CPN Tools operates with a language called Standard ML (SML), all declarations and code examples in this chapter are written in SML.

### 3.3.1 Basic Color Sets

There are several primitive color sets already defined in the CPN Tools Left panel ( Appendix A ) to represent fundamental data types:

Listing 3.1: Basic Color Sets in CPN Tools

```
colset UNIT = unit;
colset BOOL = bool;
colset TIME = time;
colset REAL = real;
colset STRING = string;
colset INTEGER = int;
```

### 3.3.2 Complex Color Sets

Building upon the basic types, we define more complex data structures:

Listing 3.2: Complex Color Sets

```
colset ID = string;
colset InputBatch = product ID * VectReal;    (* e.g., (0, [0.0, 0.0,
   0.0]) *)
colset Ins = list InputBatch;                 (* list of inputs *)
colset Inputs = product ID * Ins;             (* e.g., (batch_id, list
   of inputs) *)
colset Dataset = list Inputs;                 (* e.g., [Inputs, Inputs,
   ...] *)
colset VectReal = list REAL;
colset Matrix = list VectReal;
colset WB = list VectReal;
colset Label = product ID * INT;
colset Labels = list Label;
```

### 3.3.3 Variables and Functions

The computational core of our CPN-based neural network relies on a comprehensive suite of Standard ML (SML) functions that encapsulate the mathematical operations required for deep learning algorithms [65]. These functions serve as the operational backbone for implementing complex neural network behaviors within the Petri Net framework, transforming the traditional imperative programming approach into a token-based, event-driven computational model.

To maintain mathematical precision and computational efficiency, each function is designed with specific input and output signatures that align with the color sets defined in our CPN model. For instance, the ponderation functions operate on structured data types representing weight matrices and input vectors, while the error computation functions handle vector-based loss calculations.

Listing 3.3: Ponderation Function Implementation

```
fun ponderation ((batch_id, inputs) : Inputs, wb : WB) : Inputs =
  let
    fun dotproduct ([], []) = 0.0
      | dotproduct (x::xs, y::ys) = x * y + dotproduct(xs,ys)
      | dotproduct _ = raise Fail "Vector length mismatch";
    fun computeOne ((id, inputVect) : InputBatch) =
      let
        val weights = List.take (wb, List.length wb - 1);  (* Exclude
           bias vector *)
        val biases = List.nth (wb, List.length wb - 1);    (* Bias
           vector *)
        fun computeNeuron (w,b) = dotproduct(inputVect, w) + b;
        val outputs = ListPair.map computeNeuron (weights, biases);
      in
        (id, outputs)
      end;
  in
    (batch_id, List.map computeOne inputs)
  end;
```

The **variable declaration** system within our model serves as the binding mechanism between the functional operations and the token-based execution model of Colored Petri Nets. These variables act as formal parameters that capture the state of inputs, intermediate computations, and outputs at each transition firing. The variable binding ensures that data dependencies are explicitly maintained and that token consumption and production follow the mathematical constraints of neural network operations.

Listing 3.4: Variable Declarations

```
var i : Inputs;
var wb1, wb2, wb3 : WB;
var d1, D1, d2, D2: Matrix;
```

## 3.4 Learning and Backpropagation Mechanism

### 3.4.1 Loss Calculation

For binary classification with sigmoid activation, the cross-entropy loss function is defined as:

$$L = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \tag{3.4}$$

Where:

- $m$ is the dataset size ( number of examples )
- $y_i$ is the true label (0 or 1)
- $\hat{y}_i$ is the predicted probability from sigmoid activation: $\hat{y}_i = \sigma(z_i^{(3)}) = \frac{1}{1+e^{-z_i^{(3)}}}$

### 3.4.2 Error Terms Computation

**Error of the Output Layer**

Our neural network employs a sigmoid activation function at the output layer combined with cross-entropy loss for binary classification. This combination leads to a simplified gradient computation. The derivative of the loss with respect to the pre-activation output $z^{(3)}$ becomes:

$$\delta^{(3)} = \frac{\partial L}{\partial z^{(3)}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{(3)}} \tag{3.5}$$

$$= \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right) \cdot \hat{y}(1-\hat{y}) \tag{3.6}$$

$$= -y(1-\hat{y}) + (1-y)\hat{y} \tag{3.7}$$

$$= \hat{y} - y \tag{3.8}$$

**Error of the Hidden Layer $l_2$**

Using the chain rule:

$$\delta^{(2)} = \left(\mathbf{W}^{(3)} \cdot \delta^{(3)}\right) \odot f'(\mathbf{z}^{(2)}) \tag{3.9}$$

Where ReLU derivative is:

$$f'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{3.10}$$

Implemented in `Delta2`:

- Compute `d2 = weightedDeltaWB3(WB3, D3)`
- Multiply element-wise with ReLU derivative from `Actv'2'`

**Error of the Hidden Layer $l_1$**

$$\delta^{(1)} = \left(\mathbf{W}^{(2)} \cdot \delta^{(2)}\right) \odot f'(\mathbf{z}^{(1)}) \tag{3.11}$$

Implemented in `Delta1` :

- Compute `d1 = weightedDeltaWB2(WB2, D2)`
- Multiply element-wise with ReLU derivative from `Actv'1'`

### 3.4.3 Weight Update Mechanism

**Gradient Computation for Weights and Biases**

For each layer $l$, the gradients are computed as:

$$\frac{\partial L}{\partial \mathbf{W}^{(l)}} = \delta^{(l)} \cdot \mathbf{a}^{(l-1)\top} \tag{3.12}$$

$$\frac{\partial L}{\partial \mathbf{b}^{(l)}} = \delta^{(l)} \tag{3.13}$$

Where $\mathbf{a}^{(l-1)}$ represents the activations from the previous layer.

**Weight Update for $WB_3$ (Output Layer)**

$$\Delta \mathbf{W}^{(3)} = \frac{1}{m} \sum_{i=1}^{m} \delta_i^{(3)} \cdot \mathbf{a}_i^{(2)} \tag{3.14}$$

$$\Delta \mathbf{b}^{(3)} = \frac{1}{m} \sum_{i=1}^{m} \delta_i^{(3)} \tag{3.15}$$

Update rule:

$$\mathbf{W}^{(3)} := \mathbf{W}^{(3)} - \eta \cdot \Delta \mathbf{W}^{(3)} \tag{3.16}$$

$$\mathbf{b}^{(3)} := \mathbf{b}^{(3)} - \eta \cdot \Delta \mathbf{b}^{(3)} \tag{3.17}$$

Implemented in `updateWB3(a2, D3, WB3)` function.

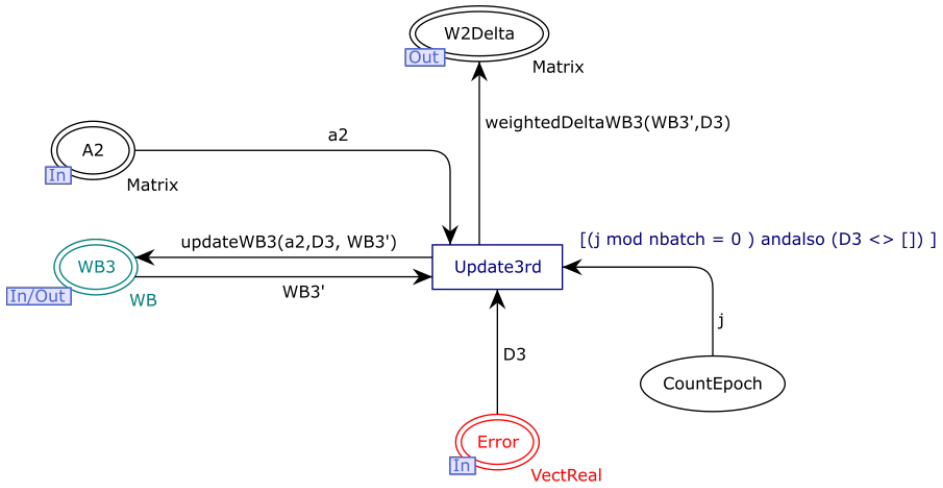


Figure 3.3: Update part of the net from CPN Tools (update3)

### Weight Update for $WB_2$ (Second Hidden Layer)

$$\Delta \mathbf{W}^{(2)} = \frac{1}{m} \sum_{i=1}^{m} \delta_i^{(2)} \cdot \mathbf{a}_i^{(1)} \tag{3.18}$$

$$\Delta \mathbf{b}^{(2)} = \frac{1}{m} \sum_{i=1}^{m} \delta_i^{(2)} \tag{3.19}$$

Update rule:

$$\mathbf{W}^{(2)} := \mathbf{W}^{(2)} - \eta \cdot \Delta \mathbf{W}^{(2)} \tag{3.20}$$

$$\mathbf{b}^{(2)} := \mathbf{b}^{(2)} - \eta \cdot \Delta \mathbf{b}^{(2)} \tag{3.21}$$

Implemented in `updateWB2(A1, D2, WB2)` function.

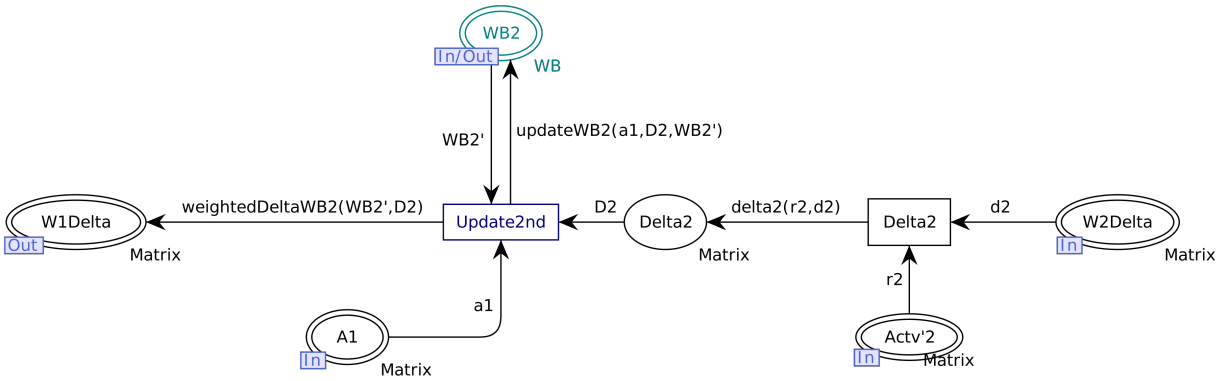


Figure 3.4: Update mechanism in the second hidden layer (update2)

### Weight Update for $WB_1$ (First Hidden Layer)

$$\Delta \mathbf{W}^{(1)} = \frac{1}{m} \sum_{i=1}^{m} \delta_i^{(1)} \cdot \mathbf{x}_i \tag{3.22}$$

$$\Delta \mathbf{b}^{(1)} = \frac{1}{m} \sum_{i=1}^{m} \delta_i^{(1)} \tag{3.23}$$

Where $\mathbf{x}_i$ is the input stored in matrix `A0`.
Update rule:

$$\mathbf{W}^{(1)} := \mathbf{W}^{(1)} - \eta \cdot \Delta \mathbf{W}^{(1)} \tag{3.24}$$

$$\mathbf{b}^{(1)} := \mathbf{b}^{(1)} - \eta \cdot \Delta \mathbf{b}^{(1)} \tag{3.25}$$

Implemented in `updateWB1(A0, D1, WB1)` function.

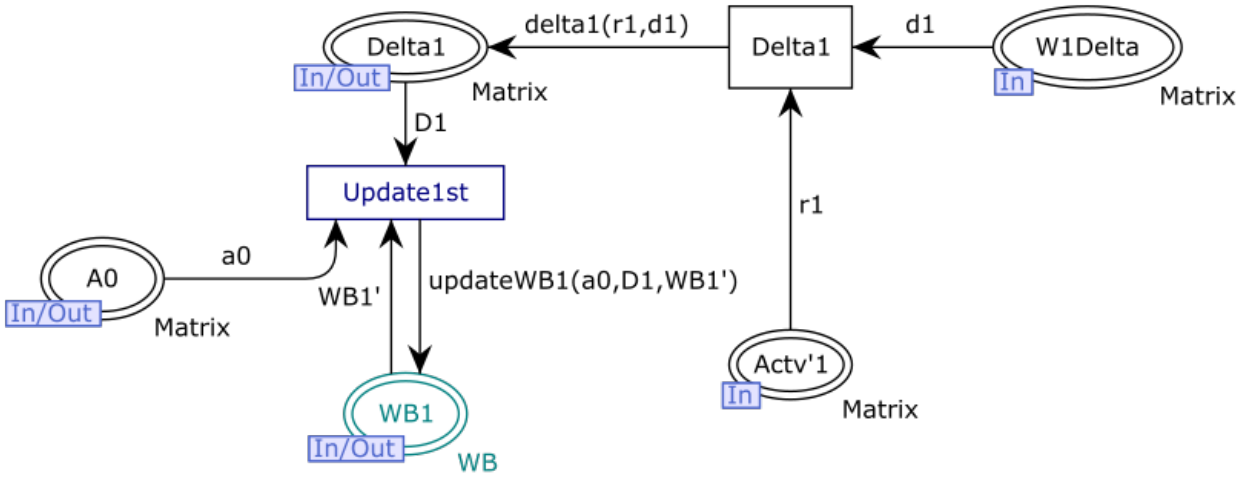


Figure 3.5: Update mechanism in the first hidden layer (update1)

We give the example of `Update3rd` (for third vector of weights):

Listing 3.5: Weight Update Example for Layer 3

```
fun updateWB3 (a : Matrix, deltaSum : REAL, wb : WB) : WB =
  let
    val lr = 0.01;
    val m = Real.fromInt(List.length a);

    fun transpose ([]::_) = []
      | transpose rows = List.map List.hd rows :: transpose (List.map
         List.tl rows);

    val aT = transpose a;
    val avgDelta = deltaSum / m;
```

```
    fun avgActivation row = List.foldl op+ 0.0 row / m;
    val avgActivations = List.map avgActivation aT;

    val w = List.nth(wb, 0);
    val newW = ListPair.map (fn (wi, ai) => wi - lr * avgDelta * ai) (w
        , avgActivations);

    val b = List.nth(wb, 1);
    val newB = List.map (fn bval => bval - lr * avgDelta) b;
  in
    [newW, newB]
  end;
```

### 3.4.4 Control and Synchronization Mechanisms

A core strength of our CPN modeling lies in its ability to explicitly encode control logic that ensures deterministic and synchronized training across epochs and batches. To achieve this, we have designed several dedicated control places that orchestrate the flow of data and trigger updates only under precise conditions.

#### Control of Epoch Transitions: `ControlUpdate` and `NBEpoch`

The place `ControlUpdate` acts as a trigger signal to initiate the loading of a new dataset segment at the beginning of each epoch. Its marking reflects the state of the training cycle and ensures that the transition `Load` is only enabled once a new epoch is ready to begin. This allows for clean synchronization between epochs, avoiding premature or repeated data reloads.

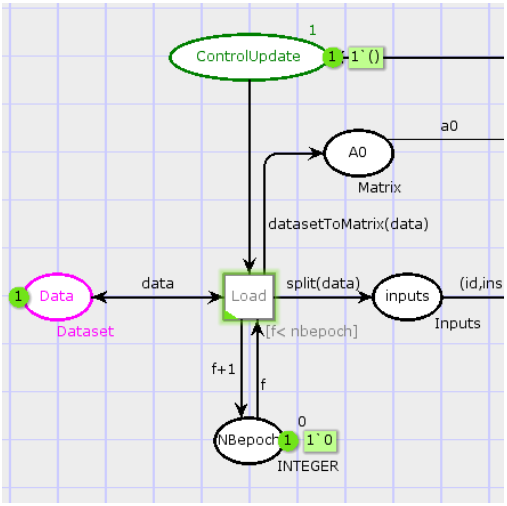


Figure 3.6: ControlUpdate place

To prevent indefinite looping through epochs, the place `NBEpoch` (an `INTEGER`-typed place) is introduced. It carries a counter `f` , initially set to 0, which is incremented after every full pass through the dataset. A guard `[f < nbeepoch]` on the `Load` transition uses this value to ensure that training is terminated once the number of epochs reaches a predefined constant `nbeepoch`. This constant is declared in the CPN declaration panel and provides configurable control over training length.

### Batch Sequencing: `ControlBatch`

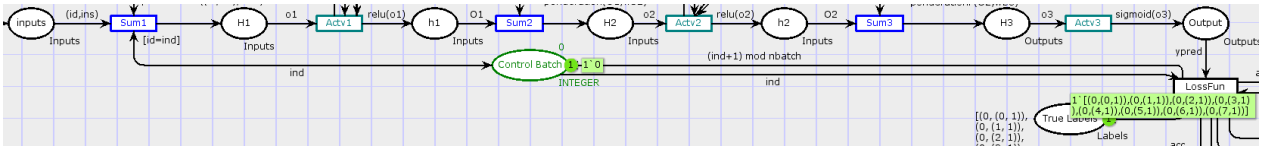


Figure 3.7: ControlBatch place

In order to enforce strict mini-batch sequencing, we introduced a dedicated place `ControlBatch`, also of type `INTEGER`. This place stores a counter *ind* that tracks the current batch index. The design ensures that batches are processed one after another in a deterministic order, avoiding non-determinism in state space generation.

The first responsible transition for processing a batch (`Sum1`) include the guard `[id = ind]`, meaning only the batch whose index matches the current counter is permitted to proceed. This sequential activation eliminates branching in the state space and ensures full coverage of the dataset in an ordered manner. After each batch is processed, the `ControlBatch` counter is incremented by one using `(ind + 1) mod nbatch`, maintaining a cyclic yet controlled batch loop.

### Epoch Finalization and Weight Updates: `CountEpoch`

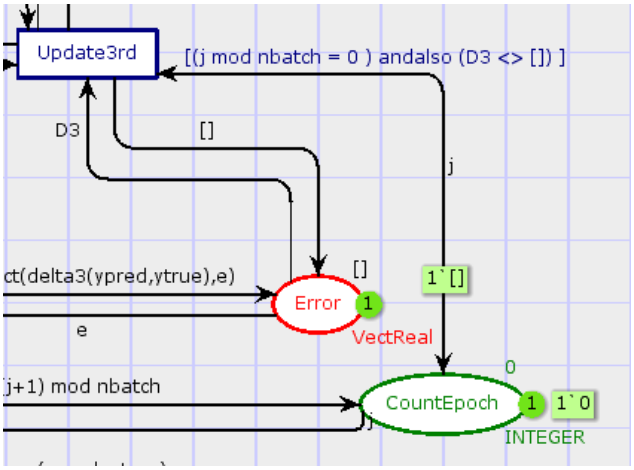


Figure 3.8: Counter of Epoch's number

To ensure that model updates only occur after all batches in the dataset have been processed, we introduced the place `CountEpoch`, another `INTEGER` counter that synchronizes with `ControlBatch`. A guard such as:

```
[(j mod nbatch = 0) andalso (D3 <> [])]
```

is applied on the transition `Update3rd`, which updates the final layer weights. This guard guarantees that:

- All batches have been passed through (`j mod nbatch = 0`)
- The error vector `D3` is not empty, i.e., error accumulation has occurred

Only under these two conditions will the network update its weights and start a new epoch cycle. This mechanism ensures batch-wise accumulation of errors followed by centralized backpropagation at the end of each epoch, mimicking traditional gradient descent with full-batch error averaging.

| Control Place | Type | Function |
|---|---|---|
| `ControlUpdate` | UNIT | Triggers loading of new epoch dataset |
| `NBEpoch` | INTEGER | Limits training to a fixed number of epochs via guard `[f < nbepoch]` |
| `ControlBatch` | INTEGER | Ensures sequential and deterministic mini-batch processing |
| `CountEpoch` | INTEGER | Enables updates only after full dataset pass with complete error vector |

Table 3.2: Summary of control places and their roles in the synchronization of training

## 3.5 Performance Evaluation

### Accuracy Calculation

The accuracy is calculated as the proportion of correct predictions:

$$Accuracy = \frac{1}{m}\sum_{i=1}^{m} I(y_i = \hat{y}_i) \tag{3.26}$$

Where $I$ is the indicator function that equals 1 when the prediction matches the true label.

Listing 3.6: Accuracy Function

```
fun accuracy(yp : REAL, ytrue : INT) : REAL =
  let
    val predicted = if yp >= 0.5 then 1 else 0
  in
    if predicted = ytrue then 1.0 else 0.0
  end;
```

# Conclusion

In this chapter, we proposed a modular and scalable modeling of neural networks using Colored Petri Nets. Our approach formalizes each step of the learning process—from input handling and forward propagation to error backpropagation and weight updates—within a visually traceable and verifiable framework. By leveraging CPN Tools, we demonstrated how deep learning architectures can be structured, synchronized, and analyzed, laying the foundation for formal verification and explainability in neural networks.

# Chapter 4

# Comparative Analysis of Neural Networks Hyperparameters and Their Impact on State Space Complexity

## Introduction

In this chapter, we explore the effects of different hyperparameters on the state space complexity of neural networks modeled using CPNs. **This analysis focuses on how neural network hyperparameters systematically influence the state space complexity in Coloured Petri Net models.**

Understanding how hyperparameters affect the behavior of the CPNN state space is critical not only for optimizing performance but also for enabling effecient formal analysis and verification. When neural networks are modeled using CPNs, the state space—which represents all possible system configurations—becomes a central consideration.

Through rigorous empirical investigation and theoretical derivation, we establish a closed-form formula that precisely predicts the number of **reachable states ( nodes )** as a function of network architecture ***depth, mini-batch configuration***, and ***training epochs***.

The methodology we employed follows a structured approach where each component of the model was carefully analyzed. This included the observation of how forward propagation, backpropagation, and the interplay of mini-batch processing influence the growth of state space with respect to epochs. By extending this investigation to multiple hidden layers and mini-batches, we derive a formula that accurately captures the complexity of state space in relation to these hyperparameters.

Before delving into the empirical effects of each hyperparameter, it is essential to clarify the core concepts and terminologies used throughout this chapter.

### 4.0.1 State Space in Coloured Petri Nets

In the context of Coloured Petri Nets, the **state space** represents the collection of all possible *reachable markings* (states) that the model can assume during its execution. Each state is defined by the distribution of tokens across the places in the net, including their associated data values (colors).

**Nodes (States):** Each *node* in the state space corresponds to a unique marking — that is, a specific assignment of tokens to places. These markings reflect the progression of the neural network's computation, including weight values, activations, errors, and other relevant data structures.

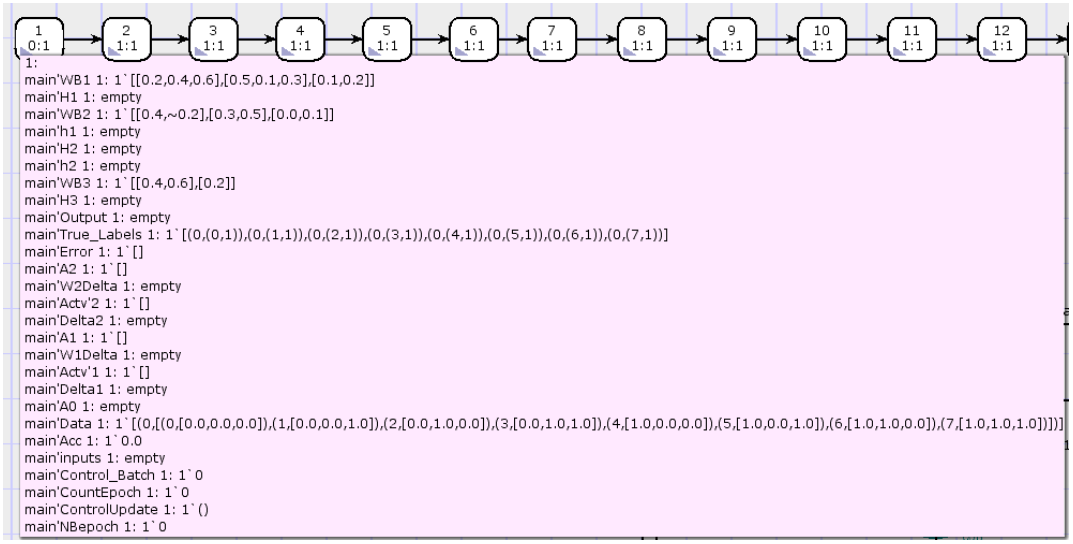


Figure 4.1: Excerpt from the state space graph (occurrence graph) generated by CPN Tools. Each box represents a unique node, i.e., a reachable marking that defines the state of the system at a given point. The marking includes the token values in places such as weight matrices (WB), activation vectors (H1, H2, etc.), and control variables. Arcs between nodes correspond to transition firings, representing steps such as forward propagation or weight updates

**Arcs (Transitions):** *Arcs* in the state space represent the firing of transitions that cause the system to evolve from one marking (state) to another. They encode computational steps such as forward propagation, error backpropagation, or weight updates.

## 4.1 Initial Observations: Transition Complexity in Simple Model

Our investigation begins with a comprehensive analysis of the baseline two-hidden-layer neural network architecture, Figure 3.1., which served as the foundational model throughout our methodology development. This architecture, consisting of an input layer, two hidden layers, and an output layer, provides the essential framework for understanding the fundamental transition patterns inherent in neural network training processes. The architecture was simple yet effective in showing how the network's complexity changes with each hyperparameter.

### 4.1.1 Epoch Number Effect on State Space Growth

***Definition*** : An epoch refers to a complete pass through the entire training dataset. During one epoch, the neural network processes all available training examples once.

To understand how the training duration affects the complexity of our model, we systematically analyzed the transition patterns within each training epoch. Our baseline two-hidden-layer architecture exhibits a well-defined sequence of computational steps that repeat consistently across training iterations.

During each epoch, the network executes a complete training cycle consisting of two distinct phases: forward propagation for prediction and backward propagation for parameter optimization. Through careful observation and analysis of the CPN model behavior, we identified the fundamental operations that constitute each training iteration.

**Forward Propagation Phase Analysis:** The forward pass through our baseline architecture involves a sequence of computational transitions that process input data through the network layers:

- **Data Loading Operation:** Initial transfer of training samples to the input layer, preparing the network for computation = 1
- **First Hidden Layer Processing:** Computation of weighted linear combinations followed by activation function application = 2
- **Second Hidden Layer Processing:** Similar weighted sum computation and activation transformation at the deeper layer = 5
- **Output Layer Computation:** Final layer processing including weighted sum calculation, activation function application, and loss computation = 3

This systematic progression through the network architecture results in exactly **8 distinct forward propagation transitions** per epoch.

**Backward Propagation Phase Analysis:** The backward pass implements the gradient-based learning mechanism through error backpropagation and parameter updates:

- **Output Layer Weight Update:** Application of computed gradients to modify the final layer's parameters = 1
- **Second Hidden Layer Error Computation:** Calculation of error terms for gradient propagation = 1
- **Second Hidden Layer Weight Update:** Parameter modification based on computed gradients = 1
- **First Hidden Layer Error Computation:** Error term calculation for the initial hidden layer = 1
- **First Hidden Layer Weight Update:** Final parameter update completing the learning cycle = 1

This backward propagation sequence consistently generates **5 distinct transitions** per epoch, regardless of the specific training data or network parameters.

**State Space Formula Derivation:** Combining both phases, each complete training epoch produces exactly $8+5=13$ transitions within our CPN model. As training progresses through multiple epochs, these transition patterns repeat, creating a linear growth in the state space complexity.

The total number of reachable states in our CPN model includes an additional component: the **initial state node**, which represents the network's starting configuration ( initial marking $M_0$ ) before any training operations commence. This initial state is crucial for CPN modeling as it serves as the entry point for all subsequent transitions and must be included in the complete state space enumeration.

Therefore, for a number of epochs $\boldsymbol{e}$, our comprehensive analysis yields the fundamental relationship:

$$\boxed{\text{Total Nodes} = (13 \times e) + 1} \tag{4.1}$$

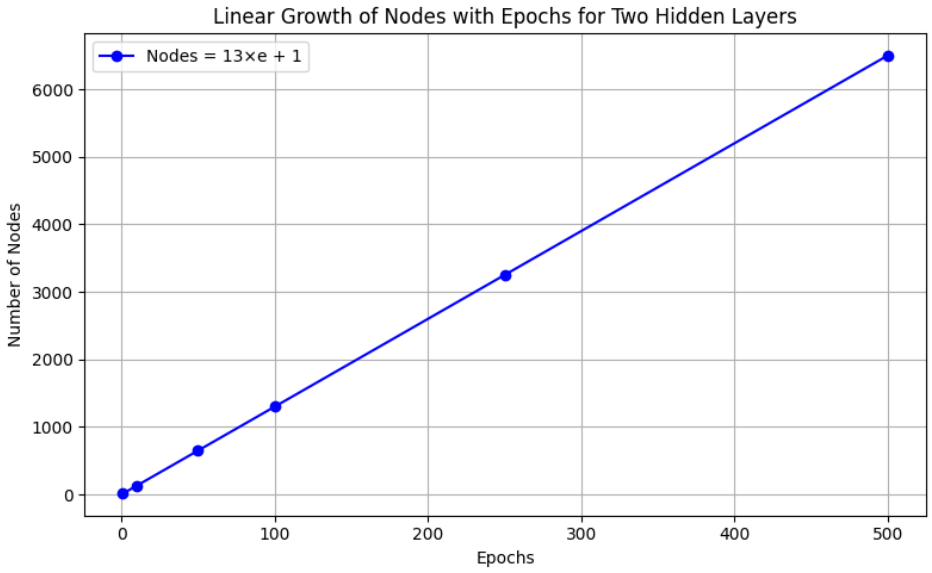


Figure 4.2: Linear Growth of State Space Nodes for Baseline architecture

## 4.2 Mini-Batch Effects on State Space Complexity: Introducing Batch-Based State Space Complexity

### 4.2.1 Mini-Batch Processing Mechanism

The introduction of mini-batching into the model further increased the complexity of the state space, as the dataset was divided into smaller subsets, denoted as $b$ (where $b \in \mathbb{N}$ represents the number mini-batches, and $nb_i$ representing the batch number $i$ with $i \in [1, b]$). This division enabled more frequent updates during training, which in turn had a noticeable impact on the size of the state space.

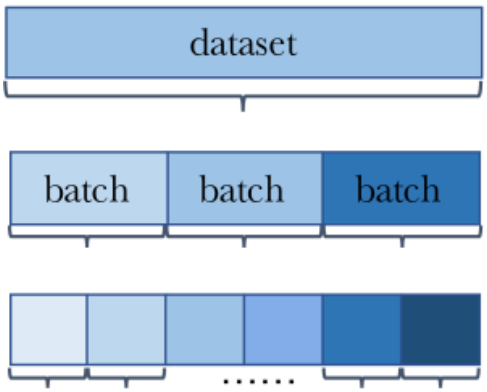


Figure 4.3: Dividing the dataset into smaller batchs

### 4.2.2 First Mini-Batch Processing Pattern

Our analysis showed that the first mini-batch in each epoch follows the full forward propagation pattern, involving all **eight** (8) forward propagation transitions. This is due to the fact that the initial mini-batch triggers the Load transition only once per mini-batch.

### 4.2.3 Subsequent Mini-Batch Optimization

For subsequent mini-batches (i.e., $nb_i$ with $i = 2, 3, \ldots, b$), while the first mini-batch passes by all 8 transitions, each subsequent batch after that one processes only 7 transitions. This reduction occurs because the Load transition is no longer needed after the first batch, as the data has already been loaded.

### 4.2.4 Backward Propagation Batch Independence

In contrast, backward propagation maintains a consistent pattern of 5 transitions regardless of the mini-batch configuration. This consistency is due to the fact that gradient computation and parameter updates must process the accumulated information from all mini-batches within an epoch before progressing to the next iteration.

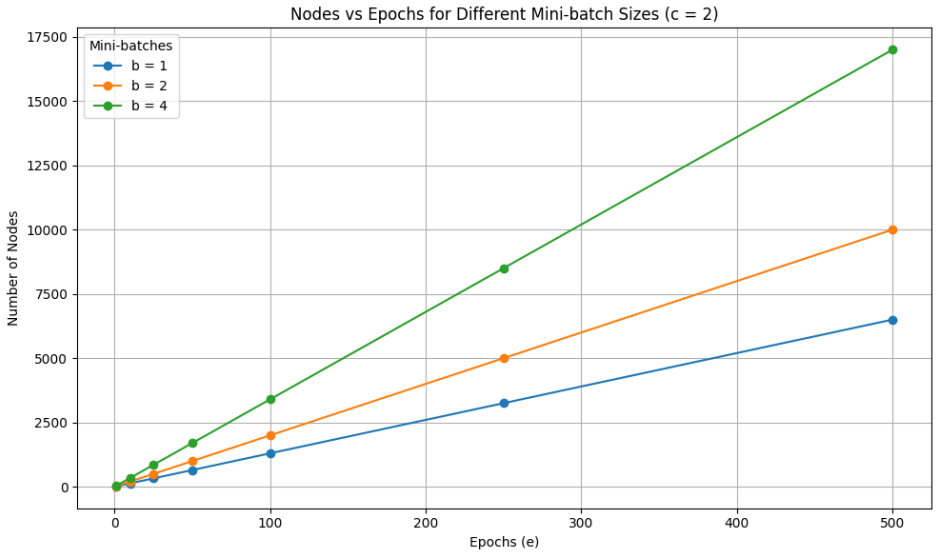


Figure 4.4: Nodes vs Epochs for Different Mini-batch Sizes for Baselien Architecture

### 4.2.5 Mini-Batch State Space Formula Derivation

After analyzing the effects of mini-batches on the state space, we derived the following relationship:

$$\text{Forward Propagation Transitions per Epoch} = 8 + 7(b - 1) \tag{4.2}$$

The total number of transitions per epoch is therefore:

$$\text{Total Transitions per Epoch} = 8 + 7(b - 1) + 5 = 13 + 7(b - 1) \tag{4.3}$$

Consequently, the total number of nodes in the state space is given by:

$$\boxed{\text{Total Nodes} = [13 + 7(b - 1)] \times e + 1} \tag{4.4}$$

Where:

- $b$ is the number of mini-batches,
- $e$ is the number of epochs.

This formula captures the relationship between mini-batches and their effect on the state space. As the number of mini-batches increases, the number of transitions during forward propagation also increases, leading to a more complex state space.

## 4.3 The Impact of Hidden Layer Count on State Space Complexity

The next phase of our analysis examined how the number of hidden layers systematically affects state space complexity. We incrementally varied the hidden layer count while maintaining consistent network width and tracked the resulting state space evolution.

### 4.3.1 Transition Pattern Analysis per Additional Layer

Our systematic investigation revealed a consistent pattern: each additional hidden layer introduces exactly four new transitions to the overall training process, distributed equally between forward and backward propagation phases.

**Forward Propagation Impact:** Each new hidden layer adds 2 transitions:

- **Sum operation transition:** Computes weighted linear combinations from the previous layer
- **Activation function transition:** Applies non-linear transformation to produce layer outputs

**Backward Propagation Impact:** Each new hidden layer similarly adds 2 transitions:

- **Delta computation transition:** Calculates error gradients for the layer's parameters
- **Weight update transition:** Applies computed gradients to modify layer weights

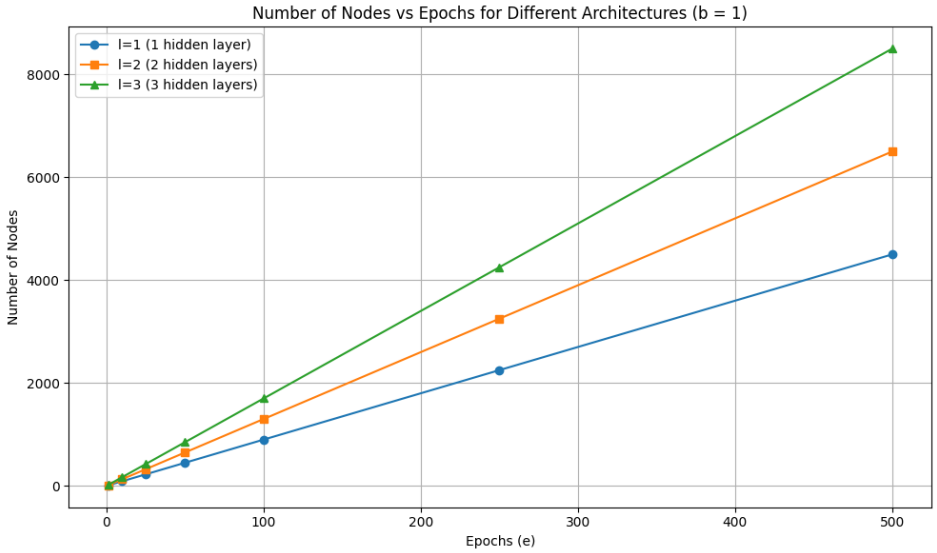


Figure 4.5: Number of Nodes vs Epochs for Different Architectures using one batch

### 4.3.2 Empirical Validation Across Layer Configurations

To validate this pattern, we analyzed three distinct architectures with the following results:

| **Architecture** | **Layers (l)** | **Forward Propagation** | **Backward Propagation** | **Total per Epoch** |
|---|---|---|---|---|
| Single Layer | $l = 1$ | $6 + 5(b-1)$ | 3 | $9 + 5(b-1)$ |
| Baseline | $l = 2$ | $8 + 7(b-1)$ | 5 | $13 + 7(b-1)$ |
| Triple Layer | $l = 3$ | $10 + 9(b-1)$ | 7 | $17 + 9(b-1)$ |

Table 4.1: Transition patterns across different layer configurations

| **Architecture** | **Complete Formula** |
|---|---|
| Single Layer | Nodes $= [9 + 5(b-1)] \times e + 1$ |
| Baseline | Nodes $= [13 + 7(b-1)] \times e + 1$ |
| Triple Layer | Nodes $= [17 + 9(b-1)] \times e + 1$ |

Table 4.2: Complete state space formulas for each architecture

The transition count progression confirms our theoretical model:

- 1 hidden layer: 9 base transitions + $5(b-1)$ mini-batch overhead
- 3 hidden layers: 17 base transitions + $9(b-1)$ mini-batch overhead

## 4.4 Mathematical Derivation and Observations

Through extensive calculations, we derived the general pattern for the number of nodes in the state space based on *the number of hidden layers*, *mini-batches*, and *epochs*. The final expression, which accounts for the complexity introduced by each hyperparameter, is:

$$\boxed{\text{Nodes}(l, b, e) = [(2l + 2) + b(2l + 3)] \times e + 1} \tag{4.5}$$

Where:

- $l$ is the number of hidden layers,
- $b$ is the number of mini-batches,
- $e$ is the number of epochs.

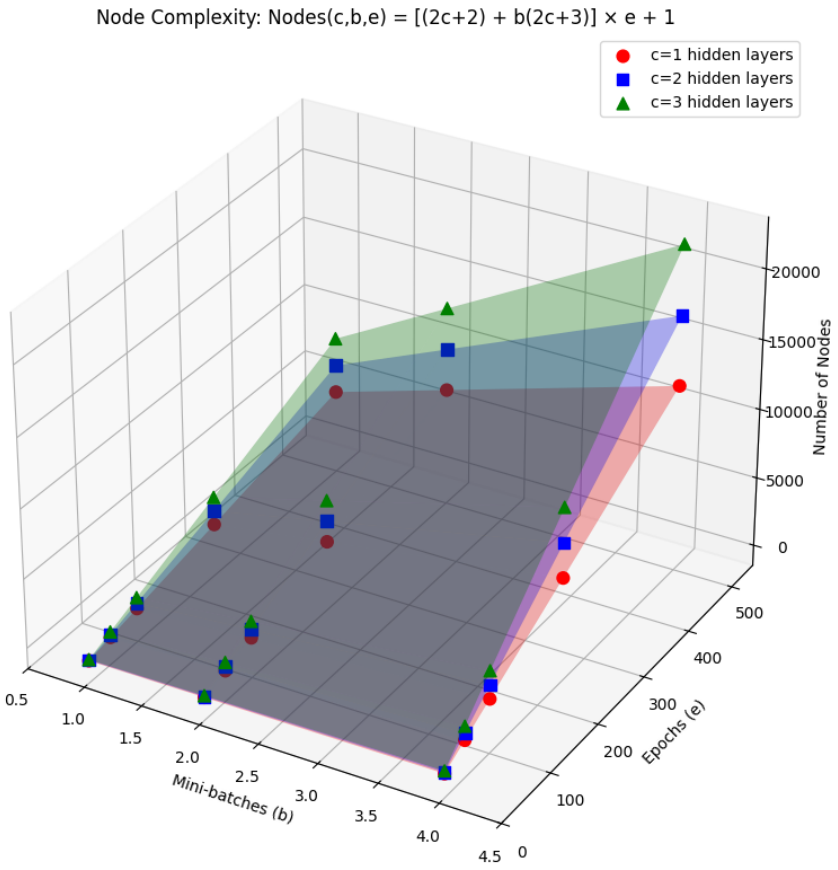


Figure 4.6: Growth of State Space Complexity as a Function of Hyperparameters

The plot illustrated in Fig 4.6 provides strong empirical validation. It effectively illustrates how deeper architectures and increased batch granularity can dramatically increase the state space, which is crucial for formal verification scalability.

# Conclusion and Insights

This comprehensive analysis establishes a rigorous mathematical foundation for understanding the effects of **hyperparameters** on state space complexity in our neural network modeling verification. The derived formula $\text{Nodes}(l, b, e) = [(2l + 2) + b(2l + 3)] \times e + 1$ provides unprecedented precision in predicting the computational requirements for formal verification tasks.

Our work demonstrates that a systematic empirical investigation, combined with theoretical analysis, can yield powerful predictive models for complex computational processes. The **linear** relationship with epochs, along with the quadratic scaling effects of layer depth and mini-batch processing, underscores the significance of hyperparameter choices. Additionally, the fact that the number of neurons per layer does not affect the complexity of the state space graph—an important feature of our modeling—makes this approach both efficient and effective.

These insights not only advance our understanding of neural network training dynamics but also lay the foundation for optimizing network architectures and batch processing strategies in practice. By utilizing these models, practitioners can predict the computational requirements and complexity of state space exploration, which is critical for tasks such as model verification, explainability, and efficiency analysis in neural network design.

# Chapter 5

# Experiments

## Introduction

In this section, we present the experimental results for our Colored Petri Neural Network (CPNN) model applied to Parkinson's disease classification. The goal of these experiments is to validate the performance of the proposed model, evaluate its formal verifiability using model checking with Computation Tree Logic (CTL), explore its explainability through feature importance, and compare it to a standard model such as Random Forest. We first introduce the dataset and experimental setup, then analyze the results and discuss the explainability aspects of the model.

## 5.1 Validating the CPNN model

### 5.1.1 Setup Environment

**Hardware:** Experiments were conducted on a Windows 11 system with an Intel(R) Core(TM) i7-7600U CPU @ 2.80GHz, 16GB RAM.

**Software:** Code was developed in Python 3.11.12 and CPN Tools v4.0.0 for Petri net modeling.

### 5.1.2 Dataset Description

The dataset used for validating the CPNN model is from **"Collection and Analysis of a Parkinson Speech Dataset With Multiple Types of Sound Recordings"** [66]. This dataset consists of 288 samples ( vowels voice samples ), of 48 Parkinson's patients and 20 healthy person, each with 26 acoustic features extracted from voice recordings. These features are crucial in diagnosing Parkinson's disease, as they capture the changes in voice patterns associated with the disease.

The **features** include:

- **Jitter metrics:** Local, absolute, RAP, PPQ5, DDP.
- **Shimmer metrics:** Local, dB, APQ3, APQ5, APQ11, DDA.

- **Pitch statistics:** Median, mean, standard deviation, minimum, and maximum pitch.
- **Voice break metrics:** Number of unvoiced frames, number of breaks, degree of voice breaks.
- **Harmonicity metrics:** AC, NTH, HTN.

Additionally, the **Status** column indicates whether a sample corresponds to a Parkinson's patient (1) or a healthy individual (0).

**Preprocessing:**

- Features were normalized using Min-Max scaling (range [0, 1]).
- The dataset was split into two parts: **70% for training** and **30% for testing**. This split ensures the model can generalize well to unseen data.

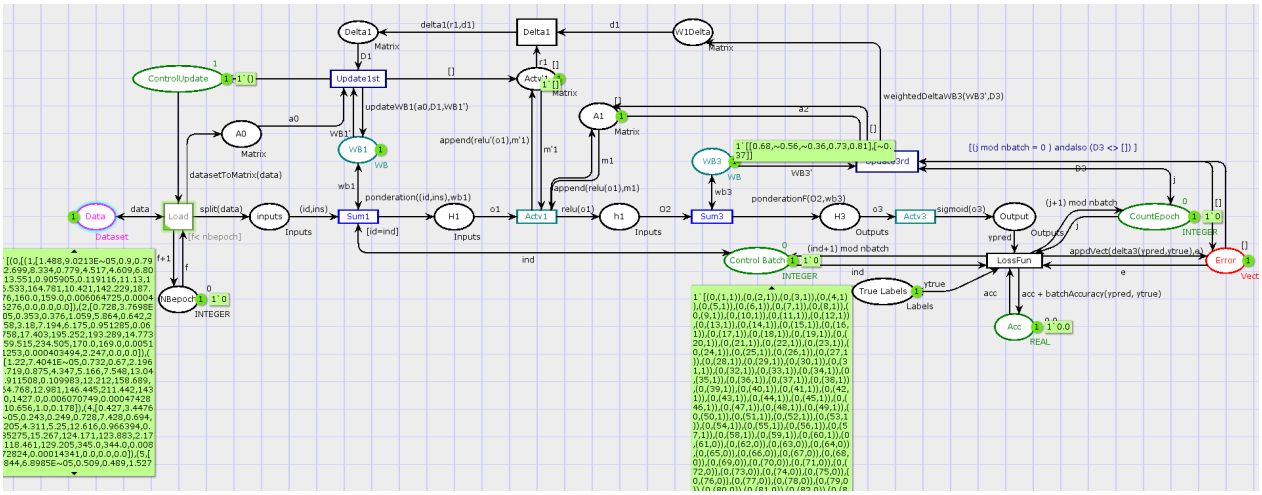


Figure 5.1: Training screenshot from cpn tools

### 5.1.3 Neural Network Architecture

We adopted an MLP-based CPNN inspired by the architecture of the model proposed in [67] :

**Input Layer:** 26 nodes, corresponding to each of the acoustic features in the dataset. Additionally, we included a unique sample ID to track tokens within the Petri net.

**Hidden Layer:** 5 neurons with **ReLU activation**.

**Output Layer:** A single neuron with **sigmoid activation**, used for binary classification (Parkinson's vs. Healthy).

**Loss Function: Binary cross-entropy** was used, which is suitable for binary classification tasks.

### 5.1.4 Results and Analysis

The results obtained from the CPNN model were analyzed in terms of **accuracy**:

- **Training Accuracy:** The model achieved an accuracy of **71.01%** after 100 epochs of training.

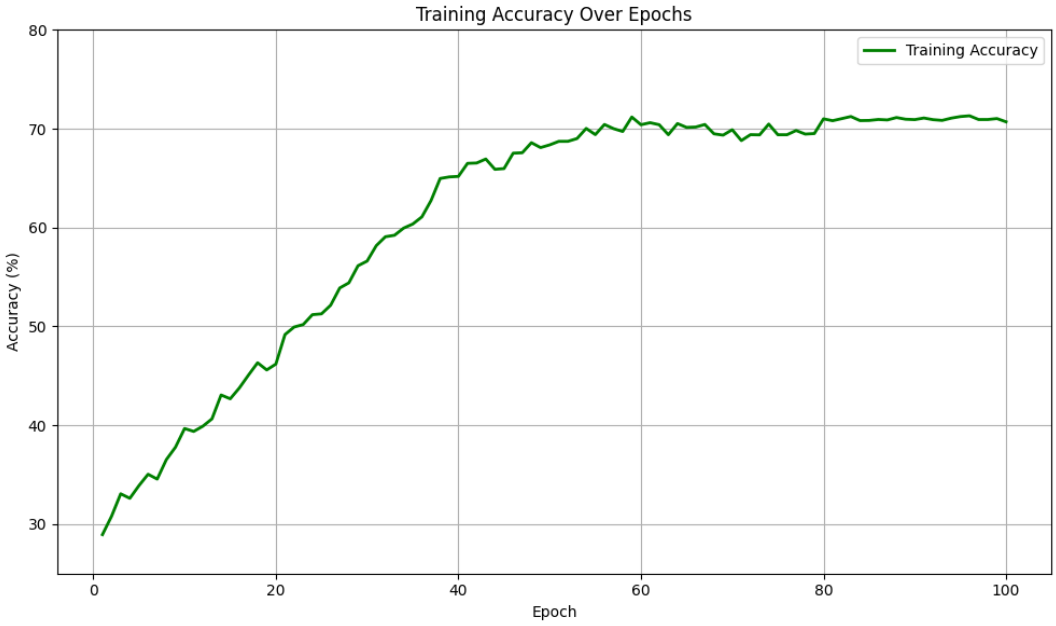


Figure 5.2: Training Accuracy Over Epochs for the CPNN model

- **Test Accuracy:** Upon evaluation on the test set, the model achieved a test accuracy of **82%**, indicating reasonable generalization capability.

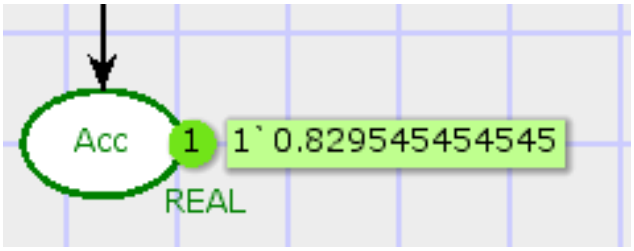


Figure 5.3: Testing Accuracy for the CPNN model

## 5.2 Exploring the explainability aspect of the CPNN model

To bridge the gap between traditional post-hoc explainability methods and intrinsic interpretability approaches, we use the formal verification tools provided by Petri Net theory—specifically, the generated state space report. This report allows us to extract clear and mathematically sound explanations of how the neural network makes decisions.

Our experimental framework, centered around the analysis of the CPN-generated state space, is organized into three core phases:

1. Systematic extraction of intermediate computational states from CPN reachability reports with parsing algorithms.
2. Implementation of a layer-wise feature importance calculation methodology based on relevance propagation principles.
3. Comparative validation against established interpretability baselines using Random Forest classifiers.

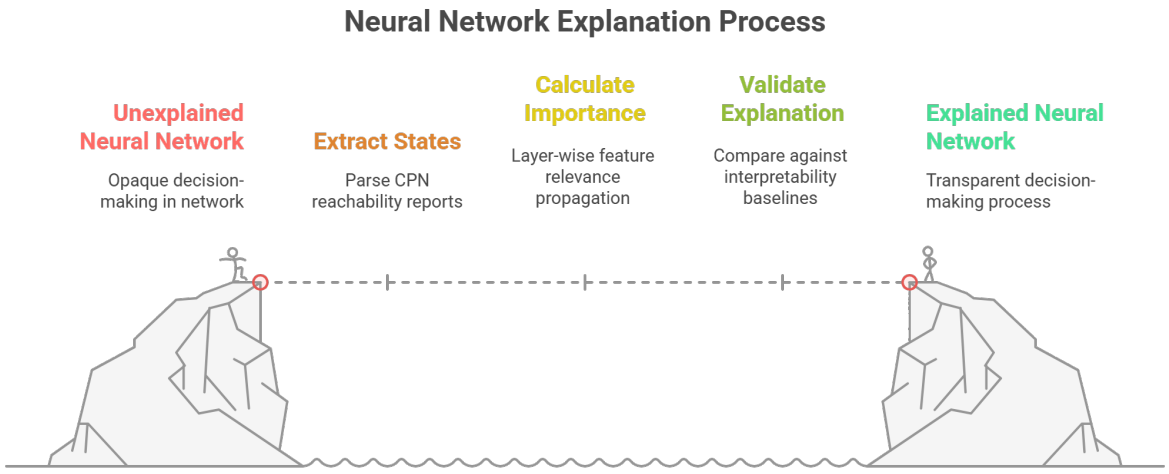


Figure 5.4: Illustration of the CPN-based neural network explanation process

### 5.2.1 Data Extraction Methodology from CPN Reachability Analysis

The CPN Tools state space report [.txt file generated using the SS tool *Appendix. A ] serves as a comprehensive computational trace that captures the complete state evolution of the neural network during forward propagation. Unlike conventional black-box approaches that only observe input-output relationships, the CPN framework provides access to all

possible paths between transitions of the system. Therefore, we can extract intermediate layer activations, weight matrices, and bias vectors at each computational step, enabling fine-grained analysis of information flow through the network architecture.

Traditional approaches provide **static feature importance,** typically computed after training. In contrast, our report provides a pathway-level analysis: we can observe how each feature's relevance propagates at each layer of the neural network and how intermediate computations affect the final output, at every iteration.

**Algorithm 1** Data Extraction from CPN Report

```
function ExtractDataFromReport(report_file_path)
    // Extract h1, wb1, and wb2 from the CPN report
    h1_raw ← ExtractPlaceData(report_file_path, "main'h1 ")
    wb1_raw ← ExtractPlaceData(report_file_path, "main'WB1 ")
    wb2_raw ← ExtractPlaceData(report_file_path, "main'WB2 ")
    // Parse the extracted raw data
    h1_data ← ParseH1Data(h1_raw, hidden_dim = 5)
    wb1_data ← ParseWeightBias(wb1_raw, expected_num_vectors = 6, expected_lengths = [26, 26, 26, 26, 26, 6])
    wb2_data ← ParseWeightBias(wb2_raw, expected_num_vectors = 2, expected_lengths = [5, 1])
    return h1_data, wb1_data, wb2_data
```

The algorithm starts by calling the `ExtractPlaceData` function to retrieve raw data for `h1_raw`, `wb1_raw`, and `wb2_raw` from the CPN report, corresponding to hidden layer activations and the weight matrices between input-hidden (WB1) and hidden-output (WB2) layers.

It then parses this data using two functions: `ParseH1Data` converts `h1_raw` into a numerical array of shape (samples, hidden_dim) with hidden_dim = 5, and `ParseWeightBias` transforms `wb1_raw` and `wb2_raw` into numerical arrays—expecting six vectors for WB1 (five weight vectors of size 26 and one bias vector of size 6), and two vectors for WB2 (one weight vector and one bias vector, each of size [5, 1]). The processed data—`h1_data`, `wb1_data`, and `wb2_data`—is then returned for further model computation.

→ The extracted data can be utilized within any explainability approach. See Fig. 5.5. For the purposes of this thesis, we have chosen to employ it for calculating feature importance.


Figure 5.5: Pipeline for explainability via state space analysis in CPN Tools. (1) The state space and state space report tools are entered within the CPN Tools environment. (2) A full state space report is generated (3) Algorithm 1 extracts and structures the relevant numerical data into a tabular format. (4) This data is then used as input for Explainable Artificial Intelligence (XAI) methods to analyze and interpret the behavior of the modeled neural network.

### 5.2.2 Feature Importance Calculation Methodology

#### Theoretical Framework: Layer-wise Relevance Propagation

The feature importance calculation is grounded in the principles of Layer-wise Relevance Propagation (LRP) [42] , which provides a principled approach for decomposing neural network predictions into input feature contributions. The core insight of LRP is that the relevance of each input feature can be computed by propagating relevance scores backward through the network layers while preserving the total relevance at each step.

#### Relevance Score for a Single Feature and Sample

Given the components above, we calculate the relevance score $R_{i,j}$ for each input feature $j$ in sample $i$. The relevance score indicates how much feature $j$ contributed to the output decision of the neural network for that sample.

The formula for relevance calculation is:

$$R_{i,j} = \sum_{k=1}^{h} \frac{X_{i,j} \cdot W_{1_{i,j}} \cdot H_{i,k} \cdot W_{2_{i,k}}}{\sum_{l=1}^{d} \sum_{k=1}^{h} (X_{i,l} \cdot W_{1_{i,l}} \cdot H_{i,k} \cdot W_{2_{i,k}})} \tag{5.1}$$

Where:

- $X_{i,j}$ is the value of the input feature $j$ for sample $i$,
- $W_{1_{i,j}}$ is the weight for feature $j$ from the input layer to the hidden layer,
- $H_{i,k}$ is the activation of the $k$-th hidden neuron for sample $i$,

- $W_{2_{i,k}}$ is the weight from hidden neuron $k$ to the output.

The relevance score $R_{i,j}$ calculates how much input feature $j$ contributed to the network's output for sample $i$, by considering the contributions from all hidden neurons $k$.

The total feature importance for feature $j$ across all samples is then computed as the average relevance score across all samples:

$$FI_j = \frac{1}{N} \sum_{i=1}^{N} R_{i,j}$$

Where:

- $FI_j$ is the total importance of feature $j$,
- $N$ is the number of samples.

This calculation gives us the overall feature importance by averaging the individual relevance scores for feature $j$ across all samples. The result is a global measure of how important each feature is to the model's predictions.

**Algorithmic Implementation**

**Algorithm 2** Layer-wise Relevance Propagation from CPN Report

**procedure** ComputeLRPFromReport(*report_file_path*, $X$, *hidden_size*)
  $h1\_data$, $wb1\_data$, $wb2\_data$ ← ExtractDataFromReport(*report_file_path*)
  $wb1\_flat$ ← FlattenData($wb1\_data$)
  $wb2\_flat$ ← FlattenData($wb2\_data$)
  $feature\_importance$ ← Zeros(number of features in $X$)
  **for** $j = 1$ to number of features **do**
    $importance\_sum \leftarrow 0$
    **for** $i = 1$ to number of samples **do**
      $denominator \leftarrow 0$
      **for** $k = 1$ to *hidden_size* **do**
        $numerator \leftarrow X[i, j] \cdot wb1\_flat[i, j] \cdot h1\_data[i, k] \cdot wb2\_flat[i, k]$
        $denominator \leftarrow denominator + numerator$
      **if** $denominator \neq 0$ **then**
        $importance\_sum \leftarrow importance\_sum + \frac{numerator}{denominator}$
    $feature\_importance[j] \leftarrow \frac{importance\_sum}{\text{number of samples}}$
  **return** $feature\_importance$

The algorithm begins by calling the `ExtractDataFromReport` function to retrieve raw data for the `h1`, `wb1`, and `wb2` matrices from the CPN report. After extracting the raw data, the `wb1` and `wb2` matrices are flattened using the `FlattenData` function to convert them into one-dimensional arrays for easier processing. The algorithm then proceeds by looping through

each input feature (j) and each data sample (i), calculating the numerator for each feature and sample as the product of the input feature value `X[i, j]`, the corresponding weight from `wb1_flat` `wb1_flat[i, j]`, the hidden layer activation value from `h1_data[i, k]`, and the corresponding weight from `wb2_flat` `wb2_flat[k]`. This numerator is added to the denominator, which is the sum of these terms for each hidden neuron (k). If the denominator is not zero, the numerator divided by the denominator is added to the `importance_sum`. After looping over all samples, the `importance_sum` for each feature is normalized by dividing it by the total number of samples to calculate the final feature importance. The algorithm then returns the `feature_importance` array, which contains the relevance score for each input feature, representing its contribution to the model's predictions.

→ Unlike gradient-based methods that only consider local derivatives, this approach provides a complete decomposition of prediction contributions across network layers.

### 5.2.3 Comparative Validation Against Random Forest Baselines

#### Benchmark Selection Rationale

We chose Random Forest classifiers' feature importance to serve as a benchmarking baseline for feature importance validation due to their simplicity and interpretability. After obtaining both results, we generated a comparative analysis figure.

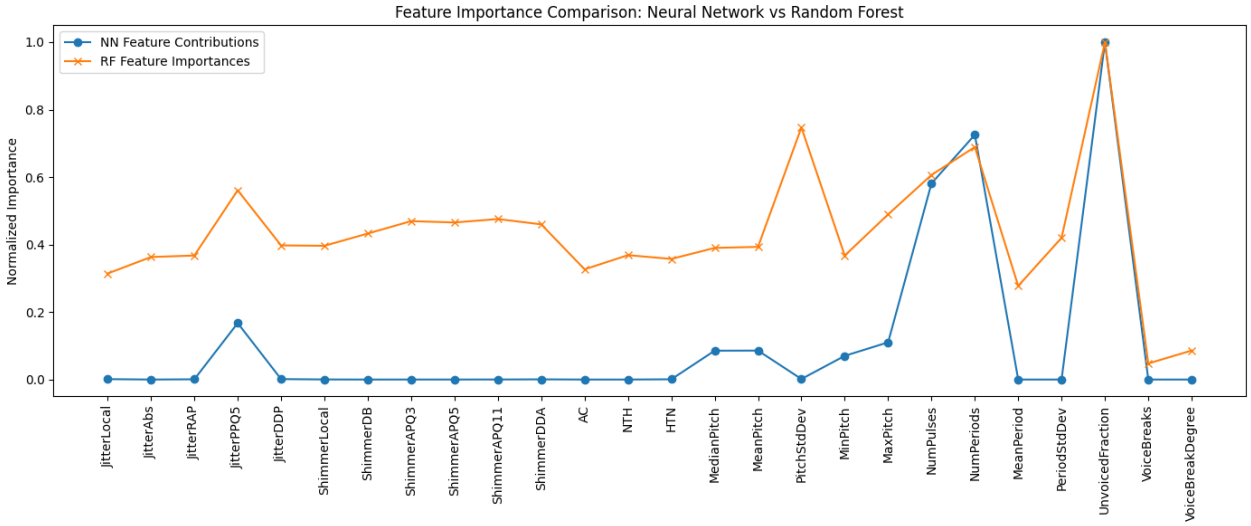


Figure 5.6: Comparison between Feature Relevance between CPNN and RF

All values are normalized. However, we observe that CPNN contributions are mostly close to zero for many features, suggesting the model relies on only a few features to make decisions.

Notably, the most important features for the neural network appear to be:

- NumPeriods
- UnvoicedFraction

- NumPulses

RF assigns more distributed importance across features compared to the CPNN. Some standout features for RF include:

- PitchStdDev
- NumPeriods
- UnvoicedFraction
- JitterPPQ5

**Relevance Analysis Results**

- The Random Forest model appears to leverage a broader range of features, possibly due to its ensemble nature, which benefits from many weak learners picking up weak signals.
- The CPNN seems to focus on a few dominant features, due to its optimization process favoring stronger gradient signals.
- The high concordance on UnvoicedFraction suggests this feature is likely a robust biomarker for Parkinson's across model types.

## Conclusion

This section validates our CPNN model through comprehensive experiments, exploring explainability aspects and providing comparative validation against baselines to demonstrate interpretability advantages. The data extracted from the state space report proved invaluable, as it gave us access to all possible states, allowing for detailed analysis and enabling us to perform various calculations. Given the limited time during the internship, we focused on making a solid start in feature importance analysis, which forms a crucial part of understanding the model's decision-making process.

# General Conclusion

## Summary of Findings

This thesis successfully introduced the Colored Petri Neural Network (CPNN) framework, a pioneering approach that integrates Colored Petri Nets with Multi-Layer Perceptron architectures to address the critical "black box" problem in deep learning. The research demonstrated that CPNs can effectively model neural network computations, providing explicit mechanisms for capturing intermediate computational states during forward propagation and enabling fine-grained analysis of feature importance and decision-making processes. The CPNN framework proved particularly valuable for applications requiring high transparency, such as medical diagnostics.

## Limitations and Challenges

The research identified two primary limitations that constrain the current framework's applicability :

- **Computational Complexity**: The method involves generating a state space and computing the reachability for each epoch, which is directly influenced by the number of hidden units and mini-batches used in training. This results in linear complexity under these parameters, as defined by Equation (4.5). However, as the number of training samples or dataset size increases, the time required for graph generation grows significantly. The trade-off between model explainability and computational efficiency thus becomes an important consideration for future work.
- **Architecture Dependency**: The current implementation of the CPNN framework has been specifically designed for Multi-Layer Perceptron (MLP) networks. As a result, it may not be directly applicable to other neural network architectures such as Convolutional Neural Networks (CNNs) or Recurrent Neural Networks (RNNs) without significant modifications.

## Future Directions for Research

Several promising avenues emerge for advancing this work, ensuring that the Colored Petri Neural Network (CPNN) framework can be applied to more complex, real-world scenarios. These include:

- **Automated Transformation Tools:** The development of automated tools for transforming diverse neural network architectures into CPNNs could significantly reduce manual effort and broaden the applicability of this methodology. This would make the process more efficient and scalable, particularly in large-scale applications.
- **Computational Efficiency Optimization:** By leveraging parallel computing, approximation methods, and state space reduction techniques, we could address the scalability challenges posed by large neural network models. This would allow the CPNN to handle more complex networks while maintaining performance and efficiency.
- **Hybrid Models and Reinforcement Learning:** Extending the framework to incorporate hybrid models (e.g., combining CPNNs with other types of machine learning models) and reinforcement learning systems would significantly enhance the versatility of CPNNs. This extension could enable applications in dynamic environments where learning and adaptation are continuous.
- **ASK-CTL Model Checking:** One exciting area for future research is the integration of *ASK-CTL model checking* within the CPNN framework. ASK-CTL (extension of CTL logic used within CPN Tools to query the state space and transitions of a model) provides the ability to query the state space and transitions of the model to ensure correctness, deadlock-free operation, and proper behavior under all conditions. By integrating ASK-CTL, we can formalize and verify dynamic processes such as backpropagation, weight updates, and learning convergence before deployment in mission-critical systems. This is a promising step toward guaranteeing that AI systems operate as expected without unexpected failures, thus increasing trust in AI-driven decisions, especially in high-stakes applications like healthcare and autonomous driving.

The ultimate goal is to create a flexible, computationally efficient framework that balances high performance and interpretability, which are essential for mission-critical AI deployments. As these avenues of research are explored, the potential for real-world applications that require both transparency and performance—such as automated diagnosis, autonomous driving, and financial modeling—will continue to grow.

While this thesis has laid the foundation for the development of interpretable and verifiable neural network models using Petri Nets, future work will continue to expand upon these initial steps. The inclusion of more complex network architectures, deeper layers, and more sophisticated verification techniques will ultimately lead to more robust, reliable, and explainable AI systems.

# Bibliography


[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. USA: Prentice Hall Press, 2009. ISBN: 0136042597.

[2] Moez Krichen. *Exploring the Feasibility of Formal Methods in Machine Learning and Artificial Intelligence*. https://hal.science/hal-04373389. HAL: ffhal-04373389f. 2024.

[3] Benedikt Knüsel et al. "Argument-based assessment of predictive uncertainty of data-driven environmental models." In: *Environmental Modelling & Software* 134 (2020), p. 104754. ISSN: 1364-8152. DOI: https://doi.org/10.1016/j.envsoft.2020.104754. URL: https://www.sciencedirect.com/science/article/pii/S1364815220300463.

[4] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." In: *Nature* 521.7553 (2015), pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

[5] Cynthia Rudin. "Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead." In: *Nature Machine Intelligence* 1.5 (2019), pp. 206–215. DOI: 10.1038/s42256-019-0048-x. URL: https://doi.org/10.1038/s42256-019-0048-x.

[6] Feiyu Xu et al. "Explainable AI: A Brief Survey on History, Research Areas, Approaches and Challenges." In: *Natural Language Processing and Chinese Computing*. Ed. by Jie Tang et al. Cham: Springer International Publishing, 2019, pp. 563–574. ISBN: 978-3-030-32236-6.

[7] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. "Toward verified artificial intelligence." In: *Commun. ACM* 65.7 (June 2022), pp. 46–55. ISSN: 0001-0782. DOI: 10.1145/3503914. URL: https://doi.org/10.1145/3503914.

[8] Christian Janiesch, Patrick Zschech, and Kai Heinrich. "Machine Learning and Deep Learning." In: *Electronic Markets* 31.3 (2021), pp. 685–695. DOI: 10.1007/s12525-021-00475-2. URL: https://doi.org/10.1007/s12525-021-00475-2.

[9] Jacek Kufel et al. "What Is Machine Learning, Artificial Neural Networks and Deep Learning? – Examples of Practical Applications in Medicine." In: *Diagnostics* 13.15 (2023), p. 2582. DOI: 10.3390/diagnostics13152582. URL: https://doi.org/10.3390/diagnostics13152582.

[10] Samreen Naeem et al. "An Unsupervised Machine Learning Algorithms: Comprehensive Review." In: *IJCDS Journal* 13 (Apr. 2023), pp. 911–921. DOI: 10.12785/ijcds/130172.

[11] J. A. M. Sidey-Gibbons and C. J. Sidey-Gibbons. "Machine Learning in Medicine: A Practical Introduction." In: *BMC Medical Research Methodology* 19.1 (2019), p. 64. DOI: 10.1186/s12874-019-0681-4. URL: https://doi.org/10.1186/s12874-019-0681-4.

[12] Vladimir Nasteski. "An overview of the supervised machine learning methods." In: *Horizons. b* 4.51-62 (2017), p. 56.

[13] Nathalie Japkowicz and Mohak Shah. *Evaluating Learning Algorithms: A Classification Perspective*. Cambridge University Press, 2011.

[14] Kristina P Sinaga and Miin-Shen Yang. "Unsupervised K-means clustering algorithm." In: *IEEE access* 8 (2020), pp. 80716–80727.

[15] K. J. Cios et al. "Unsupervised Learning: Association Rules." In: *Data Mining: A Knowledge Discovery Approach*. Ed. by K. J. Cios et al. Springer, 2007, pp. 289–306. ISBN: 978-0-387-33333-5. DOI: 10.1007/978-0-387-33333-5_10. URL: https://doi.org/10.1007/978-0-387-33333-5_10.

[16] Miao Jin et al. "Hyperparameter Tuning of Artificial Neural Networks for Well Production Estimation Considering the Uncertainty in Initialized Parameters." In: *ACS Omega* 7 (June 2022). DOI: 10.1021/acsomega.2c00498.

[17] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain [J]." In: *Psychol. Review* 65 (Nov. 1958), pp. 386–408. DOI: 10.1037/h0042519.

[18] Daniel Svozil, Vladimir Kvasnicka, and Jiří Pospíchal. "Introduction to multi-layer feed-forward neural networks." In: *Chemometrics and Intelligent Laboratory Systems* 39 (Nov. 1997), pp. 43–62. DOI: 10.1016/S0169-7439(97)00061-0.

[19] J J Hopfield. "Neural networks and physical systems with emergent collective computational abilities." In: *Proceedings of the National Academy of Sciences* 79.8 (1982), pp. 2554–2558. DOI: 10.1073/pnas.79.8.2554. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.79.8.2554. URL: https://www.pnas.org/doi/abs/10.1073/pnas.79.8.2554.

[20] Y. Lecun et al. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[21] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory." In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[22] Kyunghyun Cho et al. "On the Properties of Neural Machine Translation: Encoder–Decoder Approaches." In: *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*. Ed. by Dekai Wu et al. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 103–111. DOI: 10.3115/v1/W14-4012. URL: https://aclanthology.org/W14-4012/.

[23] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." In: *Nature* 323.6088 (1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[24] Shih-Chia Huang and Trung-Hieu Le. "Chapter 2 - Neural networks." In: *Principles and Labs for Deep Learning*. Ed. by Shih-Chia Huang and Trung-Hieu Le. Academic Press, 2021, pp. 27–55. ISBN: 978-0-323-90198-7. DOI: https://doi.org/10.1016/B978-0-323-90198-7.00006-9. URL: https://www.sciencedirect.com/science/article/pii/B9780323901987000069.

[25] GeeksforGeeks. *Backpropagation in Neural Network*. https://www.geeksforgeeks.org/machine-learning/backpropagation-in-neural-network/. Accessed: 2025-07-06. 2023.

[26] A. Adadi and M. Berrada. "Peeking inside the black-box." In: *IEEE Access* 6 (2018), pp. 52138–52160.

[27] A. B. Arrieta et al. "Explainable artificial intelligence." In: *Information Fusion* 58 (2020), pp. 82–115.

[28] W. R. Swartout. "XPLAIN system." In: *Artificial Intelligence* 21.3 (1983), pp. 285–325.

[29] E. H. Shortliffe and B. G. Buchanan. "Inexact reasoning in medicine." In: *Mathematical Biosciences* 23.3–4 (1975), pp. 351–379.

[30] T. Miller. "Explanation in AI." In: *Artificial Intelligence* 267 (2019), pp. 1–38.

[31] European Union. *General Data Protection Regulation (GDPR)*. https://eur-lex.europa.eu/eli/reg/2016/679/oj. Regulation (EU) 2016/679, applicable from May 25, 2018. 2018.

[32] S. Wachter, B. Mittelstadt, and L. Floridi. "No right to explanation." In: *International Data Privacy Law* 7.2 (2017), pp. 76–99.

[33] D. Gunning. *Explainable artificial intelligence*. Tech. rep. Defense Advanced Research Projects Agency (DARPA), 2017.

[34] C. Rudin. "Stop explaining black boxes." In: *Nature Machine Intelligence* 1.5 (2019), pp. 206–215.

[35] C. Chen et al. "Interpretable model for credit risk." In: *Decision Support Systems* 155 (2022), p. 113701.

[36] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep Learning." In: *Nature* 521.7553 (2015), pp. 436–444. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

[37] R. Caruana et al. "Intelligible models for healthcare." In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2015, pp. 1721–1730.

[38] C. Molnar. *Interpretable machine learning*. 2nd. Leanpub, 2022.

[39] E. M. Kenny and M. T. Keane. "Counterfactual explanations for deep learning." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 13. 2021, pp. 11575–11585.

[40] S. M. Lundberg and S. I. Lee. "Unified approach to model interpretation." In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017, pp. 4765–4774.

[41] M. T. Ribeiro, S. Singh, and C. Guestrin. "Why should I trust you?" In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 1135–1144.

[42] Sebastian Bach et al. "On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation." In: *PLOS ONE* 10.7 (July 2015), pp. 1–46. DOI: 10.1371/journal.pone.0130140. URL: https://doi.org/10.1371/journal.pone.0130140.

[43] R. R. Selvaraju et al. "Grad-CAM." In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 618–626.

[44] S. Wachter, B. Mittelstadt, and C. Russell. "Counterfactuals and the GDPR." In: *Harvard Journal of Law & Technology* 31.2 (2018), pp. 841–887.

[45] C. J. Cai et al. "Hello AI." In: *Proceedings of the ACM on Human-Computer Interaction* 3.CSCW (2019), pp. 1–24.

[46] E. M. Clarke and J. M. Wing. "Formal methods: State of the art and future directions." In: *ACM Computing Surveys* 28.4 (1996), pp. 626–643.

[47] C. Baier and J. P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[48] Artem Yushkovskiy and Stavros Tripakis. "Comparison of Two Theorem Provers: Isabelle/HOL and Coq." In: *CoRR* abs/1808.09701 (2018). arXiv: 1808.09701. URL: http://arxiv.org/abs/1808.09701.

[49] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[50] B. Bérard et al. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2013.

[51] P. Cousot and R. Cousot. "Basic concepts of abstract interpretation." In: *Building the Information Society*. Springer, 2004, pp. 359–366.

[52] R. Baldoni et al. "A survey of symbolic execution techniques." In: *ACM Computing Surveys* 51.3 (2018), pp. 1–39.

[53] C. A. Petri. "Kommunikation mit Automaten [Communication with Automata]." PhD thesis. University of Bonn, 1962.

[54] W. Reisig. *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013.

[55] J. Wang. "Petri Nets for Dynamic Event-Driven System Modeling." In: *Handbook of Dynamic System Modeling*. CRC Press, 2007, pp. 24-1–24-17.

[56] T. Murata. "Petri nets: Properties, analysis and applications." In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580.

[57] C. Girault and R. Valk. *Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications*. Springer, 2003.

[58] Javier Esparza and Mogens Nielsen. "Decidability Issues for Petri Nets - a survey." In: *Elektronische Informationsverarbeitung und Kybernetik* 30 (Jan. 1994), pp. 143–160.

[59] K. Jensen and L. M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009.

[60] Thomas Bourdeaud'huy and Pascal Yim. "A MATHEMATICAL PROGRAMMING APPROACH FOR THE IDENTIFICATION OF TIMED PETRI NETS." In: *IFAC Proceedings Volumes* 40.6 (2007). 1st IFAC Workshop on Dependable Control of Discrete Systems, pp. 247–252. ISSN: 1474-6670. DOI: https://doi.org/10.3182/20070613-3-FR-4909.00044. URL: https://www.sciencedirect.com/science/article/pii/S1474667015311277.

[61] Rainer Fehling. "A concept of hierarchical Petri nets with building blocks." In: *Advances in Petri Nets 1993*. Ed. by Grzegorz Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 148–168. ISBN: 978-3-540-47631-3.

[62] Chutiakrn Jitmit and Wiwat Vatanawood. "Simulating Artificial Neural Network Using Hierarchical Coloured Petri Nets." In: *Proceedings of the 2021 6th International Conference on Machine Learning Technologies* (2021). URL: https://api.semanticscholar.org/CorpusID:237424619.

[63] Yuri Resende Matias de Oliveira et al. "Coloured Petri Nets Modeling Multilayer Perceptron Neural Networks." In: *2024 IEEE International Conference on Consumer Electronics (ICCE)*. 2024, pp. 1–4. DOI: 10.1109/ICCE59016.2024.10444319.

[64] Rosângela Albuquerque et al. "A novel fully adaptive neural network modeling and implementation using colored Petri nets." In: *Discrete Event Dynamic Systems* 33 (June 2023), pp. 1–32. DOI: 10.1007/s10626-023-00377-9.

[65] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. English. MIT Press, 1990. ISBN: 0-262-63132-6.

[66] Betul Erdogdu Sakar et al. "Collection and Analysis of a Parkinson Speech Dataset With Multiple Types of Sound Recordings." In: *IEEE Journal of Biomedical and Health Informatics* 17.4 (2013), pp. 828–834. DOI: 10.1109/JBHI.2013.2245674.

[67] Wei Liu et al. "Prediction of Parkinson's disease based on artificial neural networks using speech datasets." In: *Journal of Ambient Intelligence and Humanized Computing* 14 (Apr. 2022). DOI: 10.1007/s12652-022-03825-w.

# Appendix A

# CPN Tools

## A.1 General Presentation of the Software

CPN Tools is a comprehensive modeling and simulation software originally developed by Aarhus University, Denmark. It is specifically tailored for modeling, simulating, and analyzing systems using Colored Petri Nets (CPN). The tool emerged from the necessity to have a robust, intuitive, and formally precise environment capable of effectively handling complex, data-driven concurrent systems [59].


Figure A.1: CPN Tools Logo

The primary goal of CPN Tools is to provide users, researchers, and engineers with an intuitive graphical platform for designing Colored Petri Nets models, simulating their behavior, and performing rigorous formal analysis to verify critical system properties.

## A.2 Main Features of CPN Tools

CPN Tools includes an extensive set of functionalities to leverage the full capabilities of Colored Petri Nets, notably:

- **Intuitive Graphical Editor:** Users can visually design models by drawing and connecting places, transitions, and arcs, and by annotating these components with colors, variables, and expressions directly within the graphical interface.

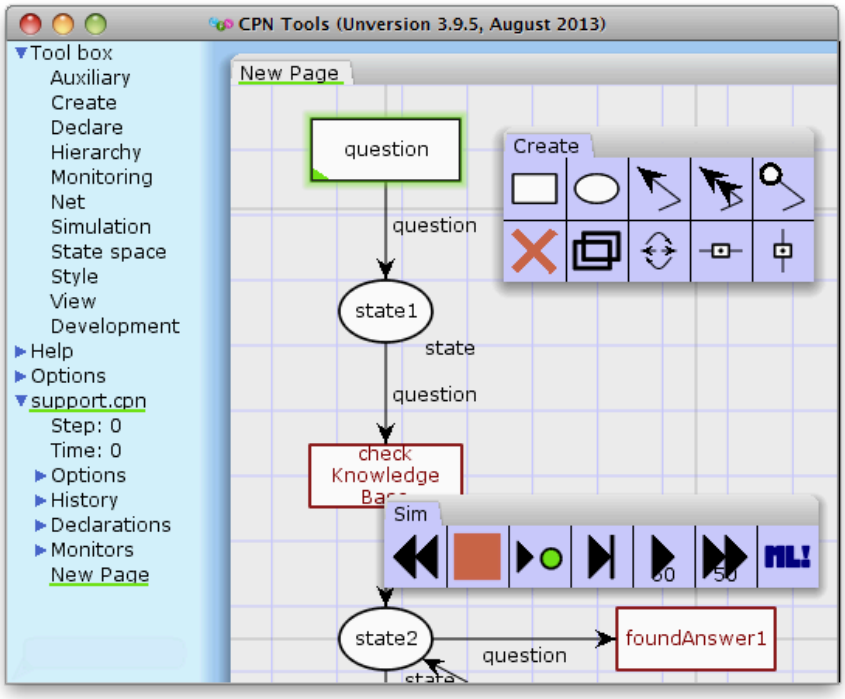


Figure A.2: CPN Tools User Interface

- **Dynamic Simulation:** A core strength of CPN Tools is its ability to perform interactive, real-time simulation. During simulations, users can dynamically observe the movement and transformation of colored tokens through the network. This visual feedback facilitates rapid validation of the model's intended behavior.

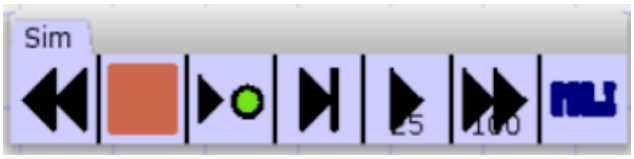


Figure A.3: Dynamic Simulation in CPN Tools

- **Formal Analysis via State-Space Exploration:** CPN Tools incorporates robust state-space analysis functionality, enabling users to formally verify essential system properties, such as:
  - Reachability of particular states.
  - Detection of potential deadlocks or infinite loops.
  - Verification of invariants and safety properties.
  - Critical-path analysis.

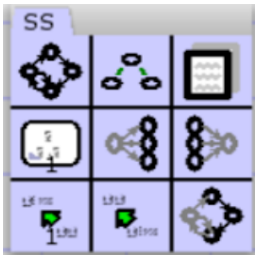


Figure A.4: State Space Analysis Tool

- **Hierarchical Modeling:** CPN Tools allows hierarchical structuring of models through sub-networks or modules, significantly enhancing clarity, manageability, and scalability of large and complex systems.


Figure A.5: Hierarchical Modeling Feature

- **Integrated Programming Language (CPN ML):** The software integrates a variant of Standard ML known as CPN ML, enabling users to write complex and domain-specific functions. This feature substantially enhances modeling flexibility and expressiveness.

## A.3 Graphical User Interface

The graphical user interface of CPN Tools emphasizes visual clarity and ease of use, structured around several tool palettes:

- **Tool box:** Contains fundamental modeling components such as places, transitions, and arcs, nets management, view and styling, etc.
- **Working Area (Canvas):** Allows direct drawing and configuration of the CPN model.

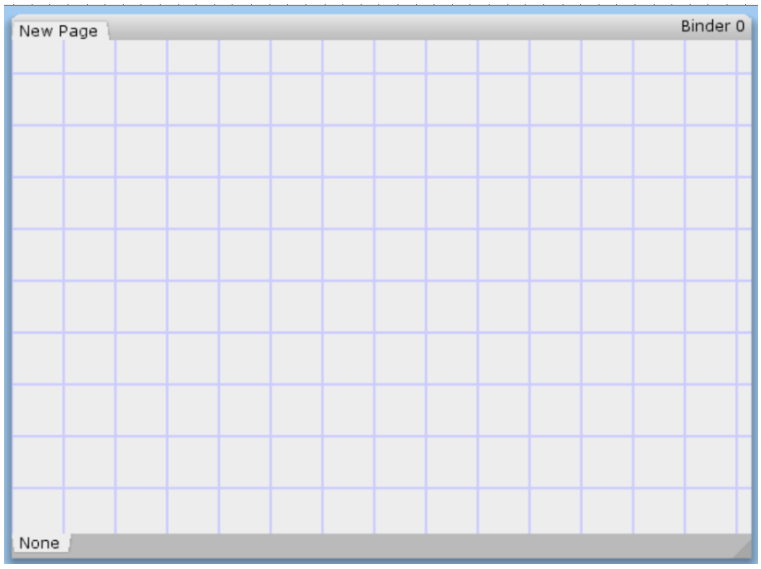


Figure A.6: Canvas Working Area

- **Simulation Pane:** Displays dynamic simulation outputs, token flows, and intermediate states.
- **Declaration Windows:** Enables definition of color sets, variables, functions, and expressions necessary for formal specification of the network.

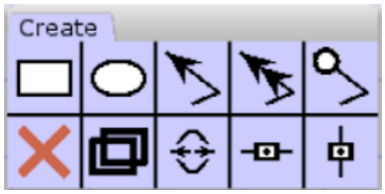


Figure A.7: Declaration Windows in CPN Tools

## A.4 Formal Analysis via State-Space Exploration

The State Space Analysis Report provides detailed information about the behavior and properties of a Colored Petri Net (CPN) model by exploring its reachability graph (also called the occurrence graph). This report helps verify system correctness, detect deadlocks, and check other behavioral properties.

### A.4.1 What's Included in the State Space Analysis Report?

The report typically contains the following sections:

1. **Statistics**
   - Number of nodes (states) – Total unique states in the system.

- Number of arcs (transitions) – Total possible state changes.
- Dead markings (deadlocks) – States where no further transitions are possible.
- Strongly Connected Components (SCCs) – Groups of states where each state is reachable from any other state in the group.

2. **Boundedness Properties**
   - Checks if places in the net are k-bounded (i.e., no place ever exceeds k tokens).
   - Reports unbounded places (if any).
3. **Home Properties**
   - Home markings – States that can be reached from any other state in the state space.
   - Useful for checking if the system can always return to a desired state.
4. **Liveness Properties**
   - Dead transitions – Transitions that can never fire in any reachable state.
   - Live transitions – Transitions that can always fire again in some future state.
   - L1-L4 Liveness (varying degrees of liveness for transitions).
5. **Fairness Properties**
   - Impartial / Just / Fair transitions – Checks if transitions occur infinitely often under certain conditions.
6. **Model Checking (Temporal Logic)**
   - CTL (Computation Tree Logic) / LTL (Linear Temporal Logic) properties (if explicitly checked).
   - Examples:
     - "Is it always possible to reach a given state?"
     - "Does the system eventually deadlock?"

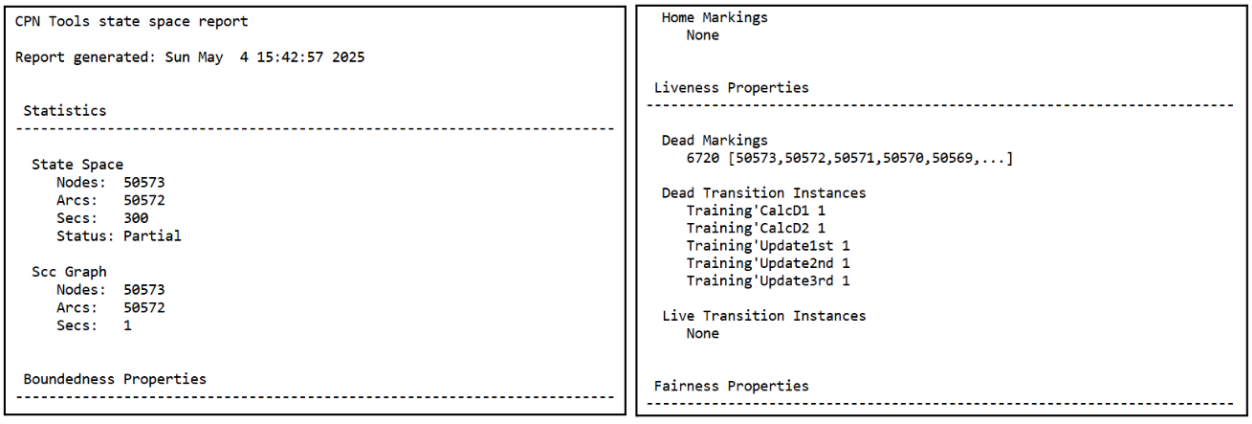


Figure A.8: Reachability Graph Visualization

# Appendix B

# Colorset Definitions for the CPNN Model

Listing B.1: Colorset Declarations Used in the CPNN Model

```
(* Basic types *)
colset UNIT = unit;
colset BOOL = bool;
colset TIME = time;
colset REAL = real;
colset STRING = string;
colset INT = int with 0..1;     (* Binary integer for label *)
colset INTEGER = int ;

(* Integer vectors *)
colset VectINT = list INT;
colset ID = INTEGER;

(* Real-valued vectors and matrices *)
colset VecReal = list REAL;
colset Matrix = list VecReal;

(* Input structure *)
colset InputBatch = product ID * VecReal;    (* One sample with ID and
   features *)
colset Ins = list InputBatch;                (* A batch of input samples
    *)
colset Inputs = product ID * Ins ;           (* Batch with batch ID *)
colset Dataset = list Inputs;                (* Full dataset: list of
   batches *)

(* Variables and functions *)
var data : Dataset ;
fun split (batch : Dataset) : Inputs list = batch;
var batch : Inputs ;
```

```
(* Label structure *)
colset Label = product ID * INT;              (* Label with ID and class
   value *)
colset IDxLabel = product ID * Label ;
colset Labels = list IDxLabel;

(* Output structure *)
colset Output = product ID * REAL;            (* Prediction with ID and
   value *)
colset IdxOut = product ID*Output;
colset Outputs = list IdxOut;

(* Weight structure *)
colset WB = list VecReal;                     (* Weight matrix: list of
   real vectors *)
```

**Notes :**

- `INT` is constrained to `0..1`, suitable for binary classification labels.
- `VecReal` and `Matrix` define vector and matrix structures of real values, essential for encoding neural network weights and activations.
- `Inputs`, `Dataset`, and `Ins` represent structured ways to store inputs, batched per epoch.
- `WB` captures a generalized weight configuration, where each list element is a weight vector going into one output neuron.
- `Output` and `Outputs` are used to track prediction values from the network with their sample IDs.