Solving Some Nonlinear Optimization Problems with Deep Learning

Dawen Wu

université PARIS-SACLAY

Solving Some Nonlinear Optimization Problems with Deep Learning

Résolution de quelques problèmes d'optimisation non linéaire avec l'apprentissage profond

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580 Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat : Informatique mathématique
Graduate School : Informatique et sciences du numérique.
Référent : Faculté des sciences d'Orsay

Thèse préparée dans l'unité de recherche **Laboratoire des Signaux et Systèmes (Université Paris-Saclay, CNRS, CentraleSupélec)**, sous la direction d'**Abdel LISSER**, Professeur, Université Paris-Saclay

Thèse soutenue à Paris-Saclay, le 30 novembre 2023, par

Dawen WU

Composition du jury

Membres du jury avec voix délibérative

Salah-Eddine EL AYOUBI Professeur, Laboratoire des Signaux et Systèmes, Université Paris-Saclay	Président
Alexei GAIVORONSKI Professeur, Norwegian University of Science and Technology	Rapporteur & Examinateur
Andrea SIMONETTO Professeur, ENSTA-Paris, Institut Polytechnique de Paris	Rapporteur & Examinateur
Jia LIU Professeur associé, Xi'an jiaotong university	Examinateur
Sihem TEBBANI Professeur, Laboratoire des Signaux et Systèmes, Université Paris-Saclay	Examinatrice

THESE DE DOCTORAT
NNT : 2023UPASG083

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title : Solving Some Nonlinear Optimization Problems with Deep Learning

Keywords : Nonlinear optimization problem, Deep learning, Neurodynamic optimization, Physics-informed neural network, Ordinary differential equation, Game theory

Abstract :

This thesis considers four types of nonlinear optimization problems, namely bimatrix games, nonlinear projection equations (NPEs), nonsmooth convex optimization problems (NCOPs), and chance-constrained games (CCGs). These problems find extensive applications in various domains such as engineering, computer science, economics, and finance. We aim to introduce deep learning-based solution algorithms for these problems.

For bimatrix games, we use Convolutional Neural Networks (CNNs) to compute Nash equilibria. Specifically, we design a CNN architecture where the input is a bimatrix game and the output is the predicted Nash equilibrium for the game. To construct a training dataset, we generate a set of bimatrix games by a given probability distribution and use the Lemke-Howson algorithm to find their true Nash equilibria. The proposed CNN is trained on this dataset to improve its prediction accuracy. After training, the CNN is capable of predicting Nash equilibria for unseen bimatrix games. Experimental results demonstrate the exceptional computational efficiency of our CNN-based approach, at the cost of sacrificing some accuracy.

For NPEs, NCOPs, and CCGs, which are more complex optimization problems, they cannot be directly fed into neural networks. Therefore, we need more advanced tools to handle these problems, namely neurodynamic optimization and Physics-Informed Neural Networks (PINNs). Specifically, we first use a neurodynamic approach to model a nonlinear optimization problem as a system of Ordinary Differential Equations (ODEs). We then use a PINN-based model as an approximate solution to the resulting ODE system, where the end state of the model represents a prediction to the original optimization problem. The neural network is trained toward solving the ODE system, thereby solving the original optimization problem. A key contribution is that our approach transforms a nonlinear optimization problem into a neural network training problem. As a result, we solve the optimization problems using only deep learning infrastructure such as PyTorch, Tensorflow, Jax, without relying on convex optimization solvers such as CVXPY, CPLEX, or Gurobi.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Résolution de quelques problèmes d'optimisation non linéaire avec l'apprentissage profond
Mots clés : Problème d'optimisation non linéaire, Apprentissage profond, Optimisation neurodynamique, Réseau neuronal informé par la physique, Équation différentielle ordinaire, Théorie des jeux

Résumé :

Cette thèse examine quatre types de problèmes d'optimisation non linéaire, à savoir les jeux à bimatrices, les équations de projection non linéaires (NPEs), les problèmes d'optimisation convexe non lisse (NCOPs) et les jeux à contraintes aléatoires (CCGs). Ces quatre classes de problèmes d'optimisation non linéaire trouvent de nombreuses applications dans divers domaines tels que l'ingénierie, l'informatique, l'économie et la finance. Nous visons à introduire des algorithmes de solution basés sur l'apprentissage profond pour ces problèmes.

Pour les jeux à bimatrices, nous utilisons des réseaux neuronaux à convolution (CNNs) pour calculer les équilibres de Nash. Plus précisément, nous concevons une architecture de CNN où l'entrée est un jeu à bimatrice et la sortie est l'équilibre de Nash prédit pour le jeu. Pour construire un ensemble de données d'entraînement, nous générons un ensemble de jeux à bimatrices selon une distribution de probabilité donnée et utilisons l'algorithme de Lemke-Howson pour trouver leurs véritables équilibres de Nash. Le CNN proposé est entraîné sur cet ensemble de données pour améliorer sa précision de prédiction. Après l'entraînement, le CNN est capable de prédire les équilibres de Nash pour des jeux à bimatrices non vus. Les résultats expérimentaux démontrent l'efficacité computationnelle exceptionnelle de notre approche basée sur CNN, au prix de sacrifier un peu de précision.

Pour les NPEs, les NCOPs et les CCGs, qui sont des problèmes d'optimisation plus complexes, ils ne peuvent pas être directement introduits dans les réseaux neuronaux. Par conséquent, nous avons besoin d'outils plus avancés pour gérer ces problèmes, à savoir l'optimisation neurodynamique et les réseaux neuronaux informés par la physique (PINNs). Plus précisément, nous utilisons d'abord une approche neurodynamique pour modéliser un problème d'optimisation non linéaire comme un système d'équations différentielles ordinaires (ODEs). Nous utilisons ensuite un modèle basé sur PINN comme solution approximative au système d'ODE résultant, où l'état final du modèle représente une prédiction au problème d'optimisation original. Le réseau neuronal est formé en vue de résoudre le système d'ODE, résolvant ainsi le problème d'optimisation original. Une contribution clé est que notre approche transforme un problème d'optimisation non linéaire en un problème d'entraînement de réseau neuronal. En conséquence, nous résolvons les problèmes d'optimisation en utilisant uniquement l'infrastructure d'apprentissage profond comme PyTorch, Tensorflow, Jax, sans compter sur les solveurs d'optimisation convexe tels que CVXPY, CPLEX ou Gurobi.

Acknowledgements

In November 2020, I set off on a solo journey from my hometown to Paris, a city then under strict lockdown due to the pandemic. I spent three wonderful years at a school called CentraleSupélec, which is located in the southern suburbs of Paris. During the PhD, I randomly studied new subjects, played sports and made new friends. I hope that many years from now, when I look back on this manuscript, I will still be proud of myself.

I would like to express my sincere gratitude to my supervisor, Prof. Abdel Lisser, for giving me the invaluable opportunity to research and study in France. As an excellent mentor, he has always encouraged me to delve deeply into mathematics and computer science, especially in areas such as optimization, differential equations and machine learning. The collection of this knowledge forms this doctoral thesis. Throughout my research journey, he has consistently identified the critical point and significantly enhanced the focus and depth of my work. His exceptional interpersonal skills and deep empathy have also kept me optimistic and calm throughout the PhD journey. Over the past three years, he has helped and supported me in many ways, both academically and personally, for which I am immensely grateful.

I would also like to thank my thesis reporters, Prof. Alexei Gaivoronski and Prof. Andrea Simonetto. Prof. Gaivoronski not only expressed his confidence in my research, but also wrote a letter of recommendation for my postdoctoral application. Prof. Simonetto carefully reviewed my thesis and provided insightful and detailed feedback, which undoubtedly improved the quality of my work. I would also like to thank the other three members of the jury : Prof. Salah-Eddine El Ayoubi, Prof. Jia Liu, and Prof. Sihem Tebbani. Thank you for taking the time to review my thesis, attend my defence and provide insightful feedback.

I would also like to thank my friends and colleagues at L2S. Shangyuan Zhang has helped me a lot to adapt to life in France. I always have lunch with Siham Tassouli, who has been very supportive throughout my time at the school. I often find a lot of common thoughts with Hoang-Nam Nguyen, and conversations with him are always inspiring. In addition, the arrival of new colleagues, Ange Valli, Lucas Osmani and Heng Zhang, adds a lot of colour to the last stage of my PhD.

Finally, I would like to express my deepest gratitude to my parents and two sisters. Their companionship and care gave me a wonderful childhood in Guangzhou and continued support throughout my studies in Paris. I hope they see a return in this thesis.

Gif-sur-Yvette,
Dawen Wu

Résumé

Cette thèse explore quatre types de problèmes d'optimisation non linéaire : les jeux bimatrix, les équations de projection non linéaire (NPEs), les problèmes d'optimisation convexe non lisse (NCOPs), et les jeux sous contraintes de chance (CCGs). Ces problèmes trouvent des applications étendues dans divers domaines tels que l'ingénierie, l'informatique, l'économie et la finance. L'objectif principal est d'introduire des algorithmes de solution basés sur l'apprentissage profond pour ces problèmes.

Dans le cas des jeux bimatrix, nous utilisons des Réseaux Neuronaux Convolutifs (CNNs) pour calculer les équilibres de Nash. Plus précisément, nous concevons une architecture CNN où l'entrée est un jeu bimatrix et la sortie est l'équilibre de Nash prédit pour le jeu. Pour construire un ensemble de données d'entraînement, nous générons un ensemble de jeux bimatrix selon une distribution de probabilité donnée et utilisons l'algorithme de Lemke-Howson pour trouver leurs véritables équilibres de Nash. Le CNN proposé est entraîné sur cet ensemble de données pour améliorer sa précision de prédiction. Après l'entraînement, le CNN est capable de prédire les équilibres de Nash pour des jeux bimatrix non vus. Les résultats expérimentaux démontrent l'efficacité computationnelle exceptionnelle de notre approche basée sur CNN, au prix de sacrifier une certaine précision.

Pour les NPEs, NCOPs et CCGs, qui sont des problèmes d'optimisation plus complexes, ils ne peuvent pas être directement introduits dans les réseaux neuronaux. Par conséquent, nous avons besoin d'outils plus avancés pour gérer ces problèmes, à savoir l'optimisation neurodynamique et les Réseaux Neuronaux Informés par la Physique (PINNs). Spécifiquement, nous utilisons d'abord une approche neurodynamique pour modéliser un problème d'optimisation non linéaire comme un système d'Équations Différentielles Ordinaires (ODEs). Nous utilisons ensuite un modèle basé sur PINN comme solution approximative au système d'ODE résultant, où l'état final du modèle représente une prédiction du problème d'optimisation original. Le réseau neuronal est entraîné à résoudre le système d'ODE, résolvant ainsi le problème d'optimisation original. Une contribution clé est que notre approche transforme un problème d'optimisation non linéaire en un problème d'entraînement de réseau neuronal. En conséquence, nous résolvons les problèmes d'optimisation en utilisant uniquement l'infrastructure d'apprentissage profond telle que PyTorch, Tensorflow, Jax, sans dépendre de solveurs d'optimisation convexe tels que CVXPY, CPLEX ou Gurobi.

Contribution 1 : Résolution du Jeu Bimatrix avec CNN. Considérons la résolution de l'équilibre de Nash d'un jeu bimatrix. Ce problème a été démontré comme étant complet PPAD [1], ce qui signifie que trouver un équilibre de Nash exact est coûteux en calcul et peu probable en temps polynomial. Cela motive l'utilisation de l'apprentissage profond pour trouver des solutions approximatives. Nous tirons avantage du fait qu'un jeu bimatrix peut être représenté simplement par ses deux matrices de paiement. Par conséquent, nous concevons une architecture CNN qui prend les deux matrices de paiement en entrée et produit un équilibre de Nash prédit pour le jeu. Le CNN proposé est entraîné sur des données de jeu auto-générées pour améliorer la précision de prédiction. Les résultats expérimentaux montrent que notre approche est nettement plus rapide que les solveurs traditionnels, en particulier lors de la résolution de plusieurs instances. Par exemple, considérant un jeu de taille 20×20, notre approche CNN résout une seule instance plus de 5 fois plus rapidement que l'algorithme LH, et résout 10 000 instances plus de 5 000 fois plus rapidement (Voir Figure 3.4, Chapitre 3.5.2), avec une erreur moyenne de 13% (Voir Table 3.2, Chapitre 3.5.3).

Contribution 2 : Résolution des NPE et NCOP avec Optimisation Neurodynamique et PINN.

Nous proposons un solveur basé sur l'apprentissage profond pour les NPE et NCOP introduits dans la Section 1.1. Les étapes principales de la solution sont décrites ci-dessous :

(i) Le problème d'optimisation non linéaire est modélisé par un système d'ODE utilisant une approche neurodynamique.

(ii) Un réseau neuronal de type PINN est utilisé comme solution approximative à ce système d'ODE, où l'état final du réseau est une prédiction pour le problème d'optimisation original.

(iii) Le réseau neuronal est entraîné à résoudre le système d'ODE, améliorant ainsi la précision de prédiction pour le problème d'optimisation original.

Pour différents types de problèmes d'optimisation, soit NPE soit NCOP, nous avons apporté des ajustements algorithmiques pour exploiter la structure du problème pour améliorer les performances. Une contribution clé de l'algorithme proposé est qu'il transforme un problème d'optimisation en un problème d'entraînement de réseau neuronal. Par conséquent, notre solveur n'utilise que des logiciels d'apprentissage profond tels que PyTorch, TensorFlow ou JAX pour résoudre le problème d'optimisation cible.

Contribution 3 : Résolution des CCG à Différents Niveaux de Confiance avec Apprentissage Profond. Nous considérons la résolution des CCG à différents niveaux de confiance α, ce qui équivaut à résoudre un ensemble de plusieurs problèmes d'optimisation non linéaires. De manière similaire à la Contribution 2, nous résolvons les problèmes en utilisant l'optimisation neurodynamique et les PINNs. Ici, la différence est que nous utilisons un seul réseau neuronal pour résoudre plusieurs problèmes d'optimisation au lieu d'un seul. Spécifiquement, nous concevons une architecture de réseau qui prend le niveau de confiance α en entrée et produit l'équilibre de Nash prédit pour le CCG à α. Les résultats expérimentaux montrent que, une fois entraîné, le modèle de réseau neuronal proposé peut prédire les équilibres de Nash à plusieurs niveaux de confiance α en un temps CPU extrêmement court au coût d'une précision raisonnable. Par exemple, considérant un CCG avec 5 joueurs, le modèle prend seulement 1 seconde pour prédire pour 500 différents α, ce qui est 1000 fois plus rapide que la méthode RK, au coût d'une erreur moyenne de 0.16 (Voir Table 6.7, Chapitre 6.4.3).

Perspectives : Finalement, nous esquissons quelques directions futures possibles :

— Côté méthodologie : Intégrer les approches proposées avec des idées, concepts ou théories de pointe en apprentissage profond pour améliorer l'efficacité computationnelle et la précision.

— Côté application : Appliquer les approches proposées pour résoudre des problèmes réels.

— Côté théorique : Mener une analyse de convergence pour les approches proposées.

Table des matières

1 - Introduction

Nonlinear optimization problems serve as critical tools in diverse fields including engineering, physics, economics, and finance. Such problems are pivotal in decision-making processes, ranging from the enhancement of industrial production to financial investment. They also contribute to mission-critical tasks such as optimal design, resource allocation, and risk assessment. The landscape of challenges is continuously evolving, marked by increasing competition and constrained resources. Consequently, the quest for more efficient and economical solutions has become critical. Advances in computing power have synergized with the rapid development of optimization models, thereby significantly broadening their applicability and scope.

Nonlinear optimization originated from linear programming, a specific subclass where both the objective function and constraints are linear. Initiated during World War II for resource allocation and scheduling, linear programming evolved into a standalone academic field following Dantzig's seminal introduction of the simplex method [2]. Subsequently, various algorithms such as the interior-point and ellipsoid methods have further diversified the field, extending its capabilities to tackle a broader spectrum of complex, real-world challenges [3].

The shift from linear to nonlinear frameworks gives rise to a diverse array of nonlinear optimization problems. These problems involve at least one nonlinear function in the objective or constraints and can be divided into convex and nonconvex categories. Convex optimization problems are typically well-posed, with the optimal solution solvable by algorithms such as interior-point methods or Newton methods [4]. In contrast, nonconvex problems present the challenge of multiple local optima, and solving the global optimum is difficult. While heuristics such as simulated annealing, genetic algorithms, and gradient-based methods offer some solutions, they do not guarantee convergence to the global optimum [5].

Furthermore, nonsmooth optimization problems introduce an additional layer of complexity due to their discontinuous derivatives. These problems find diverse applications, from LASSO regression in statistics to robust optimization in supply chain management. Their resolution frequently necessitates specialized approaches such as subgradient methods or various approximation techniques. Recent advancements have shed light on the convergence properties of these algorithms [6, 7] and introduced innovative techniques like derivative-free optimization [8].

Nonlinear optimization is closely related to game theory. Game theory studies the interactions among rational individuals, serving as a robust framework for understanding choice, conflict, and cooperation in various decision-making contexts [9]. The cornerstone of this framework is the concept of Nash equilibrium, which defines a stable state in which no player benefits from a unilateral change in strategy [10]. In addition to the Nash equilibrium, several other types of equilibria

have been extensively studied in non-cooperative game theory [11, 12, 13]. Their existence is usually proved by fixed point theorems.

While the existence of a Nash equilibrium in a game is generally guaranteed, finding it can be computationally challenging. In special cases like two-player zero-sum games, Nash equilibria can be reformulated as linear programming problems [2] and efficiently solved using interior-point methods. However, for more general scenarios such as n-player non-zero-sum games, determining the Nash equilibrium becomes considerably more complex. Such problems fall into the category of PPAD-complete, meaning that there is no polynomial-time algorithm for solving them [14].

This thesis explores the use of machine learning as a potent tool for addressing nonlinear optimization problems [15]. As a cornerstone of artificial intelligence, machine learning has permeated diverse domains, from data analysis and decision making to pattern recognition. Traditional machine learning algorithms such as decision trees and support vector machines laid the initial groundwork. However, these algorithms often stumble when dealing with high-dimensional and complex data structures. This limitation has catalyzed the rise of neural networks and deep learning [16].

Characterized by multi-layer architectures and activation functions, deep neural networks are revolutionizing fields from computer science to engineering and beyond. In 2012, AlexNet, a convolutional neural network, set an unprecedented performance benchmark in the ImageNet competition [17, 18]. Since that milestone, deep learning has rapidly evolved and diversified, finding applications in a variety of fields, including computer vision [16], natural language processing [19], robotics [20], and bioinformatics [21], among others.

In response to a diverse array of challenges, numerous specialized model architectures have emerged to handle specific tasks. For example, Convolutional Neural Networks (CNNs) for computer vision [22, 16], Recurrent Neural Networks (RNNs) for natural language processing [23, 16], Generative Adversarial Networks (GANs) for data generation [24], Graph Neural Networks (GNNs) for structured data [25], and Physically-Informed Neural Networks (PINNs) for solving partial differential equations [26]. These specialized architectures not only offer robust and efficient means for analyzing complex problems and data but also broaden the scope of possibilities across various scientific and industrial domains.

Training a neural network essentially boils down to solving a large-scale, unconstrained, non-convex optimization problem. Stochastic Gradient Descent (SGD) is a widely used training method for neural networks and has several variants, such as Adam [27] and AdaGrad [28]. These variants often incorporate adaptive learning rates and momentum terms to improve the convergence speed and stability of the original SGD algorithm. Choosing the appropriate variant and tuning its hyperparameters can significantly influence the training efficiency and the quality of the resulting model.

1.1 . Four Nonlinear Optimization Problems

In this thesis, we aim to solve the following four optimization problems.

Problem 1 : Bimatrix Game. We consider solving two-player general-sum games with finite actions for both players. Such a game is also known as a bimatrix game, since its two payoff matrices contain all the information about the game. Player 1 and player 2 have the payoff matrices $\mathbf{A}$ and $\mathbf{B}$, respectively, of the form

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{bmatrix}, \tag{1.1}$$

where m and n denote the numbers of actions for player 1 and player 2, respectively. When player 1 selects the i-th action and Player 2 selects the j-th action, player 1 and player 2 receive payoffs of a_{ij} and b_{ij}, respectively. The objective is to solve for the Nash equilibrium $(x^* \in X, y^* \in Y)$ that satisfies the following system of inequalities :

$$\begin{aligned} (x^*)^T \mathbf{A} y^* &\geq x^T \mathbf{A} y^* \quad \forall x \in X, \\ (x^*)^T \mathbf{B} y^* &\geq (x^*)^T \mathbf{B} y \quad \forall y \in Y, \end{aligned} \tag{1.2}$$

where $X = \{x \in \mathbb{R}^m \mid \mathbf{1}_m^T x = 1, x \geq \mathbf{0}\}$ and $Y = \{y \in \mathbb{R}^n \mid \mathbf{1}_n^T y = 1, y \geq \mathbf{0}\}$, and $\mathbf{1}_m^T \in \mathbb{R}^m$ represents an all-ones vector.

Traditional solution algorithms for solving bimatrix games include the Lemke-Howson algorithm [29] and enumeration methods [9]. The Lemke-Howson (LH) algorithm is the state-of-the-art exact solution algorithm that guarantees to find a Nash equilibrium for a game. The enumeration methods, including support enumeration and vertex enumeration, are used to find all Nash equilibria for a game, but they are computationally more expensive. Existing research indicates that solving bimatrix games is challenging, as the computational steps for both the LH algorithm and the enumeration methods grow exponentially with the size of the game, even in the best-case scenario [30].

Problem 2 : Nonlinear Projection Equation (NPE). We consider solving NPEs that take the following form :

$$P_\Omega(x^* - G(x^*)) = x^*, \tag{1.3}$$

where $G : \mathbb{R}^n \to \mathbb{R}^n$ is a nonlinear function, and $\Omega \subset \mathbb{R}^n$ is a feasible set. The projection function $P_\Omega : \mathbb{R}^n \to \Omega$ projects any vector $z \in \mathbb{R}^n$ onto the feasible set Ω, defined as,

$$P_\Omega(z) = \arg\min_{x \in \Omega} \|z - x\|_2 \tag{1.4}$$

The objective of an NPE problem is to find an optimal solution $x^* \in \Omega$ that solves (1.3). As demonstrated by Harker & Pang (1990) [31], NPEs can be viewed as a unified framework for many optimization problems, including nonlinear complementarity problems, variational inequaties, and equilibrium point problems.

The NPE problem is commonly solved by neurodynamic optimization methods [32, 33, 34]. These methods use a system of first-order ordinary differential equation (ODE) to model the NPE problem. Under certain conditions on the function G and the feasible set Ω, the ODE system exhibits a global convergence property, meaning that its state solution converges to the optimal solution of the NPE problem.

Problem 3 : Nonsmooth Convex Optimization Problem (NCOP). We consider solving NCOPs that take the following form

$$\begin{cases} \min\limits_{x} f(x) \\ \text{s.t.} \\ \quad g(x) \leq \mathbf{0}, \\ \quad x \geq \mathbf{0}, \end{cases} \tag{1.5}$$

where $x \in \mathbb{R}^n$ represents the decision variables. $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$ denote the objective function and the constraint function, respectively. Problem (1.5) can be classified into various types depending on the properties of f and g, such as linear programming when both f and g are linear, quadratic programming when f is quadratic and g is linear, convex optimization when both f and g are convex, and nonsmooth optimization when the functions are nonsmooth.

The problem(1.5) has a wide range of applications spanning in statistical learning, compressed perception, optimal transportation, signal processing, image processing, financial engineering, power systems, and other fields. There are standard and well-established algorithms to solve the optimization problem (1.5), particularly in cases where both the objective function and the constraint function are convex and smooth. For example, the barrier method and the primal-dual interior-point method are commonly used solution algorithms. See more details in [35, 4].

Problem 4 : Chance Constrained Game (CCG). An n-player general-sum game is represented by the tuple of $\left(N, (A_i)_{i\in N}, (u_i)_{i\in N}\right)$, where

— $N = \{1, 2, \ldots, n\}$ is the set of players.

— $A_i = \{1, \ldots, a_i\}$ is the action set of player i.

— $u_i : \prod_{i=1}^{n} A_i \to \mathbb{R}$ is the payoff function for player i.

When the game contains randomness in its payoff functions, such a game is called a Stochastic Nash Game. Research works [36, 37, 38, 39, 40] leverage the framework of chance-constrained programming to accommodate such randomness, culminating in the formulation of (CCGs). In CCGs, players are guaranteed to achieve payoffs with a predefined confidence level.

When the probability distributions in CCGs satisfy certain conditions, they can be reformulated as solvable mathematical optimization problems. For example, the two-player zero-sum game, with a random strategy set following an elliptical distribution, is reformulated as a second-order cone programming problem [40]. The n-player game general-sum game, with the random payoff following an elliptical distribution, is reformulated as a variational inequality [39].

1.2 . Contributions

The goal of this thesis is to solve the four nonlinear optimization problems presented in Section 1.1 using deep learning. The main contributions of this thesis are summarized below.

Contribution 1 : Solving Bimatrix Game using CNN. Consider solving the Nash equilibrium of a bimatrix game. This problem has been shown to be PPAD-complete [1], which means that finding an exact Nash equilibrium is computationally expensive and unlikely to be done in polynomial time. This motivates the use of deep learning to find approximate solutions. We take advantage of the fact that a bimatrix game can be represented simply by its two payoff matrices. Therefore, we design a CNN architecture that takes the two payoff matrices as an input and outputs a predicted Nash equilibrium for the game. The proposed CNN is trained on self-generated game data to improve the prediction accuracy. Experimental results show that our approach is significantly faster than traditional solvers, especially when solving multiple instances. For example, considering a game of size 20×20, our CNN approach solves a single instance over 5 times faster than the LH algorithm, and solves 10,000 instances over 5,000 times faster (See Figure 3.4, Chapter 3.5.2), with an average gap error of 13% (See Table 3.2, Chapter 3.5.3).

Contribution 2 : Solving NPE and NCOP using Neurodynamic Optimization and PINN. We propose a deep learning-based solver for the NPE and NCOP introduced in Section 1.1. The main solution steps are outlined below :

(i) The nonlinear optimization problem is modeled by an ODE system using an neurodynamic approach.

(ii) A PINN-like neural network is used as an approximate solution to this ODE system, where the end state of the network is a prediction for the original optimization problem.

(iii) The neural network is trained toward solving the ODE system, thereby improving the prediction accuracy for the original optimization problem.

For different types of optimization problems, either NPE or NCOP, we have made algorithmic adjustments to exploit the problem structure for improved performance. A key contribution of the proposed algorithm is that it transforms an optimization problem into a neural network training problem. Therefore, our solver only uses deep learning software such as PyTorch, TensorFlow or JAX to solve the target optimization problem.

Contribution 3 : Solving CCGs at Different Confidence Levels using Deep Learning. We consider solving CCGs at different confidence levels α, which is equivalent to solving a set of multiple nonlinear optimization problems. Similar to Contribution 2, we solve the problems based on the use of neurodynamic optimization and PINNs. Here, the difference is that we use a single neural network to solve multiple optimization problems instead of one. Specifically, we design a

network architecture that takes the confidence level α as input and outputs the predicted Nash equilibrium for the CCG at α. Experimental results show that, the proposed neural network model, once trained, can predict Nash equilibria at multiple confidence levels α in an extremely short CPU time at a reasonable cost of accuracy. For example, considering a CCG with 5 players, the model takes only 1 second to predict for 500 different α, which is 1000 times faster than the RK method, at the cost of an average error of 0.16 (See Table 6.7, Chapter 6.4.3).

The remainder of this thesis is organized as follows : Chapter 2 provides an in-depth review of three different topics : neurodynamic optimization, PINNs, and machine learning for solving optimization problems. Chapter 3 solves bimatrix games with CNNs. Chapters 4 and 5 solve NPEs and NCOPs, respectively, and both chapters are based on neurodynamic optimization and PINNs. Chapter 6 introduces CCGnet, a network architecture for solving CCGs at different confidence levels. Finally, Chapter 7 concludes the thesis and discusses potential avenues for future research.

Please note that each chapter is self-contained and has its own conventions for mathematical notation and abbreviations.

2 - Related Works

This chapter first introduces two important methods, neurodynamic optimization and physics-informed neural networks, which are used extensively throughout this thesis. Then, we present some recently emerging research on deep learning for solving optimization problems.

2.1 . Neurodynamic Optimization

This section provides a comprehensive review of the literature on neurodynamic optimization. In particular, in Section 2.1.1 we describe in detail how the neurodynamic approach solves an optimization problem. We also introduce the most basic model in neurodynamic optimization, the recurrent model. Section 2.1.2 introduces projection neurodynamic models, which are a special class of neurodynamic models dedicated to solving nonlinear projection equations. Section 2.1.3 discusses collaborative neurodynamic optimization, which has been a hot topic in recent years.

2.1.1 . Recurrent Neurodynamic Models

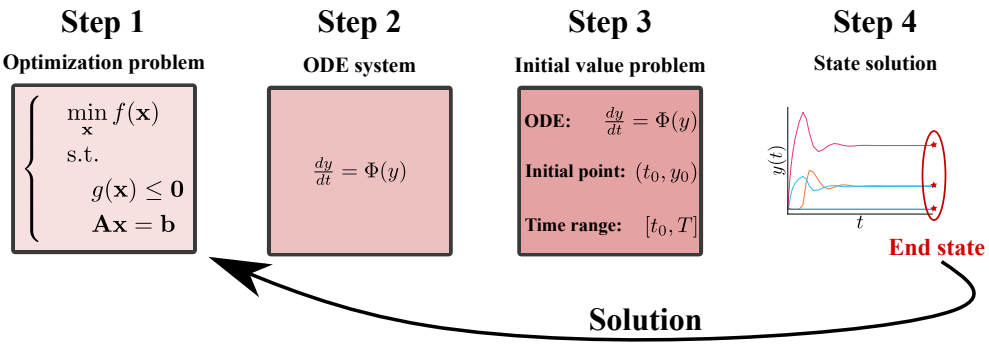


Figure 2.1 – A neurodynamic approach for solving a nonlinear optimization problem.

Fig. 2.1 outlines the procedure for using a neurodynamic approach to solve a nonlinear optimization problem. The details are explained below :

— Step 1 : Consider an optimization problem, such as one of the four problems listed in Section 1.1.

— Step 2 : The neurodynamic approach constructs a first-order Ordinary Differential Equation (ODE) system that should have a global convergence property.

— Step 3 : Specify an initial point and an appropriate time range to formulate an initial value problem for the ODE system.

— Step 4 : Solve the initial value problem using a numerical integration method and obtain the state solution. In the state solution, the end state represents the solution to the original optimization problem.

The most critical step here is the second one, which involves proving the global convergence property of the ODE system. The global convergence property here means that the state solution of the ODE system converges to the optimal solution of the original optimization problem as the time variable goes to infinity. The proof of this property usually consists of two parts : 1) The equilibria of the ODE system coincide with the optimal solutions of the original optimization problems. 2) The ODE system converges globally to these equilibria. Typically, the proof is based on the use of the Lyapunov method [41].

In the following, we discuss the development and applications of the most classical neurodynamic model, i.e., Recurrent Neurodynamic Models (RNMs). Tank and Hopfield proposed the first RNM and used it to solve linear programming problems [42], and such a method is also called the Hopfield network. Since then, a variety of RNMs have been proposed to solve various optimization problems, such as quadratic programming problems [43, 44, 45], nonlinear convex optimization problems[46, 47, 48, 49], nonsmooth convex optimization problems [50, 51, 52], and minimax optimization problems [53, 54]. These RNMs have been proven to be globally convergent to the optimal solution of the corresponding problem. In addition to model development and analysis, RNMs have found applications in many areas, including sparse signal reconstruction [55, 56, 57], feature selection [58, 59], and portfolio selection [60, 61], to name a few.

In recent years, RNMs have also been used to solve nonconvex problems. Finding the global minimum in nonconvex problems is a well-known challenging task. To the best of our knowledge, there is no solution algorithm that solves this problem efficiently. Fortunately, recent studies have shown that the stationary points of nonconvex optimization problems perform well in many practical applications, such as some statistical problems in machine learning [62]. Therefore, there have been studies using RNMs to solve the first-order stationary point of the nonconvex problem.

Kurdyka et al. proved that, under certain conditions, the state solutions of negative gradient systems can converge to the stationary points of unconstrained non-convex optimization problems [63]. Based on this, RNMs were proposed to find the first-order stationary point by negative gradient systems. In addition, with the use of penalty methods, RNMs have been developed to solve for constrained nonconvex problems [64, 65]. These methods have subsequently been extended to nonsmooth problems by replacing gradient to subgradient [66, 67]. However, it is worth noting that the aforementioned RNMs all rely on the use of penalty parameters. The disadvantage of using penalty parameters is that in many cases

optimal solutions can only be achieved when these parameters are set extremely high, which is computationally expensive. This has led to the development of RNMs based on Tikhonov regularization to avoid the use of penalty parameters [68, 51].

2.1.2 . Projection Neurodynamic Models

Projection Neurodynamic Models (PNMs) are a special class of neurodynamic models that are closely related to nonlinear projection equations (NPEs). According to the literature [31], NPEs can be viewed as a framework for unifying the treatment of many constrained optimization problems, including variational inequalities, nonlinear complementarity problems, and nonlinear programming problems. PNMs are used to solve NPEs, thus addressing a wide range of nonlinear optimization problems.

Initially, PNMs were focused on solving the general form of NPE [69, 32]. Over time, PNMs have been specifically modified to solve various optimization problems [70, 71], especially variational inequalities [72, 73]. In addition, PNMs have found applications in areas such as robotic manipulators, model predictive control, and data fusion [74].

Similar to the RNMs described in Section 2.1.1, PNMs can also be used to solve standard constrained optimization problems. The core strategy is to reformulate the optimization problem into its optimality conditions, such as the KKT conditions, which can be further reformulated as an NPE. In particular, PNMs have been adapted to solve standard nonlinear optimization problems such as quadratic optimization problems [75], nonsmooth optimization problems [64], and pseudoconvex optimization problems [71].

An important issue in PNMs is the model size. Early PNMs often introduced auxiliary variables to facilitate problem solving [76, 69]. However, this approach leads to computational inefficiency as the problem size increases. Therefore, recent PNM research has focused on reducing the use of auxiliary variables, thereby reducing the model dimensions and improving computational efficiency [77, 78, 79]. For example, the reduced dimensional PNMs use a projection function to ensure that the equality constraints are satisfied, thereby avoiding auxiliary variables with respect to the equality constraint and thus improving computational performance [78, 69, 80].

2.1.3 . Collaborative Neurodynamic Optimization

In Sections 2.1.1 and 2.1.2, we presented the use of a single neurodynamic model to solve a nonlinear optimization problem. However, these single-model approaches cannot solve for more complex problems, such as mixed-integer optimization problems, or nonconvex optimization problems. Recently, a popular topic in neurodynamic optimization is using multiple neurodynamic models to collaboratively solve complex optimization problems. This approach is called collaborative neurodynamic optimization (CNO), which is essentially a combination of neurodynamic approaches and particle swarm optimization [81]. Specifically, the initial

states of a set of neurodynamic models are updated using particle swarm optimization.

CNO has been effectively applied to a wide range of complicated problems, including but not limited to, distributed optimization problems [82, 83, 84], bilevel optimization problems [85], biconvex optimization problems[86], mixed integer problems [87], nonconvex optimization problems [34], multi-objective optimization [88, 84], and combinatorial optimization problems [34, 87].

The basic neurodynamic models used in CNO are of particular importance. A CNO approach based on multiple PNMs has been proposed to solve combinatorial optimization problems [34]. In addition, a two-time scale CNO approach based on two RNMs has been proposed for mixed integer optimization problems [87]. Both of these two approaches were shown to almost surely converge to the global optimal solution.

The CNO framework has found various applications, such as in model predictive control [89, 90], nonnegative matrix factorization [91], vehicle-task assignment [92, 89], robust portfolio selection [60], spiking neural network regularization [93], sparse bayesian learning [94], and hash bit selection [95, 96].

2.2 . Deep Learning for Solving Differential Equations

The advent of deep learning has opened up a new paradigm for solving ordinary differential equations and partial differential equations (ODEs and PDEs). This section is organized as follows :

— In Section 2.2.1, we demonstrate the use of Physics-Informed Neural Networks (PINNs) to solve PDEs, followed by a literature review on PINNs.

— In Section 2.2.2, we introduce two advanced research topics closely related to PINNs, namely deep energy methods and DeepONet.

2.2.1 . Physics-Informed Neural Networks

A typical PDE problem can be expressed in the following general form :

$$\begin{aligned} D_x(u;\lambda) &= f(x), \qquad x \in \Omega \subset \mathbb{R}^d \\ \mathcal{B}_k(u) &= g_k(x), \quad x \in \partial\Omega \subset \mathbb{R}^d, \quad \text{for } k = 1, 2, \ldots, n_b, \end{aligned} \tag{2.1}$$

where $D_x(\cdot)$ is the differential operator, and $u : \Omega \cup \partial\Omega \to \mathbb{R}$ is the solution to be found. Ω and $\partial\Omega$ represent the domain and the boundary of the domain in $\mathbb{R}^d$, respectively. d denotes the dimension of the PDE. λ denotes the parameters of the PDE. $\mathcal{B}_k(\cdot)$ denotes to the boundary conditions, which can be of the Dirichlet, Neumann, or mixed type. n_b represents the number of boundary conditions. For problems involving temporal dynamics, time t is considered as part of x, and the initial conditions can be treated as a unique type of boundary condition.

Remark 2.1 *The ODE systems considered in Section 2.1 are a special case of the PDE problem* (2.1) *considered in PINNs. The main differences are :*

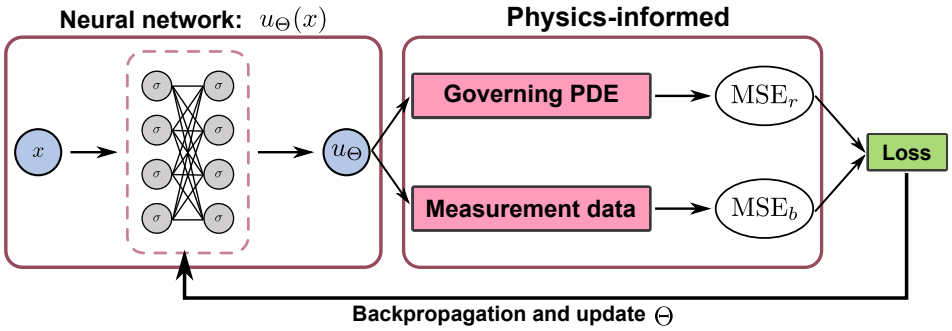


Figure 2.2 – A typical PINN framework for solving a nonlinear PDE.

— *The solution of the ODE system is a function with only one input, while the PDE solution is a function with multiple inputs.*

— *The ODE system involves multiple differential equations, each corresponding to a state dynamic, whereas the PDE typically involves only a single equation with multiple variables.*

The solution of (2.1) *has the form of* $u : \mathbb{R}^d \to \mathbb{R}$*, where* d *is the number of variables. In contrast, the solution of the ODE system has the form of* $u : \mathbb{R} \to \mathbb{R}^n$*, where* n *is the number of differential equations in the system.*

Let $\{x_b^{(i)}, u_b^{(i)}\}_{i=1}^{N_b}$ be a set of randomly chosen points for the boundary conditions. Let $\{x_r^{(i)}\}_{i=1}^{N_r}$ be a set of randomly chosen points for the PDE residual. These points are usually drawn from unknown distributions. Let u_Θ be a surrogate function based on a neural network with model parameters Θ. The goal of the PINN approach is to learn a surrogate function u_Θ that solves the target PDE problem. Figure 2.2 summarizes the solution procedure of PINNs.

The loss function for the PINN approach can be expressed as :

$$\mathcal{L}(\Theta) = W_b \,\mathrm{MSE}_b\left(\Theta; \{x_b^{(i)}, u_b^{(i)}\}_{i=1}^{N_b}\right) + W_r \,\mathrm{MSE}_r\left(\Theta; \{x_r^{(i)}\}_{i=1}^{N_r}\right), \tag{2.2}$$

where W_b and W_r represent the weights for the boundary and residual losses, respectively. $\mathrm{MSE}_b\left(\Theta; \{x_b^{(i)}, u_b^{(i)}\}_{i=1}^{N_b}\right)$ and $\mathrm{MSE}_r\left(\Theta; \{x_r^{(i)}\}_{i=1}^{N_r}\right)$ are given by :

$$\begin{aligned}
\mathrm{MSE}_b\left(\Theta; \{x_b^{(i)}, u_b^{(i)}\}_{i=1}^{N_b}\right) &= \frac{1}{N_b}\sum_{i=1}^{N_b}\left|u_b^{(i)} - u_\Theta\left(x_u^{(i)}\right)\right|^2, \\
\mathrm{MSE}_r\left(\Theta; \{x_r^{(i)}\}_{i=1}^{N_r}\right) &= \frac{1}{N_r}\sum_{i=1}^{N_r}\left|D_x(u_\Theta(x_r^{(i)}); \lambda) - f(x_r^{(i)})\right|^2,
\end{aligned} \tag{2.3}$$

where MSE_b measures the data mismatch term, which enforces the boundary conditions as constraints. MSE_r evaluates the PDE residual at a finite set of collocation points. The neural network parameters Θ are trained by minimizing the loss function in (2.2).

The success of PINNs lies in their use of automatic differentiation [97], which provides an efficient and accurate evaluation of the PDE operator. Thus, PINNs transform the task of solving a PDE into a neural network training problem, where the global minimum of the loss function signifies the solution to the PDE. Next, we briefly review the development of PINNs and highlight and some important contributions.

With the rapid development of deep learning, PINNs have emerged as a robust computational framework for solving both forward and inverse PDE problems [26, 98, 99, 100]. Unlike traditional numerical methods such as Finite Element Methods (FEMs), PINNs take advantage of a mesh-free architecture to provide unparalleled flexibility. In addition, PINNs are uniquely capable of incorporating physics-based constraints and empirical data into their loss function, paving the way for a wide range of applications in computational science and engineering. These span multiple domains, including but not limited to nano-optics inverse problems [101], metamaterials [101], fluid dynamics [98], and systems biology parameter estimation [102, 103]. Recent advances have even extended the applicability of PINNs to specialized PDEs such as integro-differential equations [99], fractional PDEs [104], and stochastic PDEs [105].

Despite their remarkable achievements, PINNs are not without limitations. Addressing increasingly complex PDE problems poses both theoretical and practical challenges that require further advances in PINN capabilities for improved predictive accuracy, computational efficiency, and robustness [100]. Several extensions to the basic PINN model have already been proposed, including the introduction of meta-learned loss functions [106] and gradient-augmented PINNs, which incorporate gradient information into the PDE residuals [107]. The issue of effectively balancing multiple loss terms in the total loss has also been addressed through automated weight-tuning methods [108, 109, 110]. Advanced strategies such as spatial domain decomposition and temporally staged training further accelerate PINN training while improving model accuracy [111, 112, 113, 114, 115, 116, 117, 118]. In addition to these general improvements, problem-specific adaptations such as the precise enforcement of Dirichlet or periodic boundary conditions via specialized network architectures have also been developed [119, 120, 121].

2.2.2 . Other Related Topics

Deep Energy Methods (DEMs). DEMs are another cutting-edge deep learning approaches for solving PDEs [122, 123, 124]. A key difference between DEMs and PINNs is the loss design. In PINNs, the loss function is constructed to consider only the general form of PDEs. Conversely, DEMs ingeniously exploit the inherent structure of the problem at hand to design their loss functions. For example, in solid mechanics, the Principle of Minimum Potential Energy (PMPE) dictates that the potential energy of a system reaches a local minimum at equilibrium. This principle synergizes well with the DEM minimization framework, allowing the use of the total potential energy as its loss function.

The innovative loss design has provided DEMs with superior computational performance, especially in solving computational mechanics problems. Applications of DEMs cover a wide range of problems, including Poisson's equation [122], linear elasticity [123, 125], hyperelasticity [123, 125], viscoelasticity [126], piezoelectricity [123], fracture mechanics [123], strain gradient elasticity [123], and topology optimization [127]. In particular, the DEM loss function requires only the first-order gradient of the displacement fields, making it less computationally intensive than the PINNs [26, 128, 129], which require both second-order and first-order spatial gradients.

Deep Operator Network (DeepONet). Another rapidly growing topic is DeepONet. Unlike the aforementioned PINNs and DEMs, which use neural networks to approximate the solutions to PDEs, DeepONet aims to approximate general PDE operators that map one function to another [130]. DeepONet integrates two separate sub-networks : a branch network for encoding the input function and a trunk network for the spatial variables. Subsequent adaptations have introduced convolutional layers to the branch network [131], multiple branch networks for encoding various input source functions [132], and intermediate data fusion strategies before the final dot product [133]. DeepONet has found applications in a wide range of areas, including heat diffusion [134], plastic deformation in dogbone specimens [135], multi-scale analyses [136], crack propagation [137], Darcy flow over complex domains [138], and engine combustion [139].

2.3 . Deep Learning for Solving Optimization Problems

In this section, we present two deep learning methods for solving optimization problems. The first, known as *Voice of Optimization* [140, 141], considers solving mixed integer convex optimization problems. The second, known as *Deep Unfolding* [142, 143, 144], considers solving inverse problems with many applications in image reconstruction and signal recovery. These two methods have connections with our thesis and may inspire further development.

Method	Target Problem	NN architecture	Input Space	Output Space	Loss Type	Problem Receiving	Multiple Problems
[140, 141] (Chapter 2.3.1)	Mixed integer convex optimization	FNN	$\mathbb{R}^p$	$\mathbb{R}^{d+m}$	Empirical Loss	Partial	Yes
[142, 143, 144] (Chapter 2.3.2)	Linear inverse problem	FNN	$\mathbb{R}^m$	$\mathbb{R}^n$	Empirical Loss	Full	Yes
[145] (Chapter 3)	Bimatrix game	CNN	$(\mathbb{R}^{m\times n}, \mathbb{R}^{m\times n})$	$(\mathbb{R}^m, \mathbb{R}^n)$	Empirical Loss	Full	Yes
[146] (Chapter 4)	Nonlinear projection equation	FNN	$\mathbb{R}$	$\mathbb{R}^n$	ODE Loss	Full	No
[147] (Chapter 5)	Nonsmooth convex optimization	FNN	$\mathbb{R}$	$\mathbb{R}^n$	ODE Loss	Full	No
[148] (Chapter 6)	Chance-constraint games	FNN	$\mathbb{R}^2$	$\mathbb{R}^n$	ODE Loss	Full	Yes

Table 2.1 – Summary of the deep learning methods covered in this thesis

Table 2.1 summarizes the six deep learning approaches in this thesis. We will now explain the meaning of each column :

— "Target Problem" refers to the optimization problem to be solved.

— "NN Architecture" refers to the neural network (NN) architecture adopted by the approach.

— "Input Space" and "Output Space" describe the input and output of the NN, respectively.

— "Problem Receiving" refers to whether the NN receives all or part of the information of the target optimization problem. For example, if the NN only receives some problem parameters of the optimization problem, it is considered partial reception.

— "Multiple Problem" indicates whether the NN can solve multiple optimization problems coming from the same probability distribution.

In particular, the "Loss Type" column in Table 2.1 indicates the type of loss function used for training the NN. We categorize the loss into two types : empirical loss and ODE loss. The empirical loss has the following form

$$\ell(w) = \|NN(x; w) - y^*\|, \tag{2.4}$$

where NN represents a NN model with training parameters w. x is the model input. $NN(x; w)$ is the prediction given by the model, and y^* is the corresponding ground truth. Such a loss measures the discrepancy between the prediction and the true value, and it is widely used in numerous deep learning tasks.

The ODE loss has the following form

$$\ell(w) = \left\| \frac{\partial NN(x; w)}{\partial x} - \Phi(NN(x; w)) \right\|, \tag{2.5}$$

where $\frac{\partial NN(x;w)}{\partial x} \in \mathbb{R}^n$ denotes the derivative of the NN output with respect to its input x. $\Phi(NN(x; w)) \in \mathbb{R}^n$ is the expected true derivative, and $\Phi(\cdot)$ is an ODE system. This loss function is a special case of the PINN loss in Eq. (2.2), where we consider the ODE system instead of a PDE. The NN is trained with this loss function to become a solution to the ODE system.

2.3.1 . Voice of Optimization

Bertsimas & Stellato [140] proposed a machine learning approach for mixed integer convex optimization (MICO) problems, namely *Voice of Optimization*. Below, we present how this approach works.

Consider the MICO problem with the following form

$$\begin{aligned} \min_{x} \quad & f(x, \theta) \\ \text{s.t.} \quad & g_1(x, \theta) \leq 0, \\ & \vdots \\ & g_m(x, \theta) \leq 0, \\ & x_{\mathcal{I}} \in \{0, 1\}^d, \end{aligned} \tag{2.6}$$

where $x \in \mathbb{R}^n$ is decision variables, and θ is problem parameters or problem data. $\mathcal{I}$ is the index set for the decision variables constrained to take binary values, with $|\mathcal{I}| = d$. The functions f and $g_j, j \in \{1, 2, \ldots, m\}$ are assumed to be convex w.r.t. x.

The solution approach consists of two phases, Training Phase and Prediction Phase, as detailed below.

Training Phase : This phase generates a training dataset with the following form

$$\{(\theta_i, s(\theta_i))\}_{i=1}^N, \tag{2.7}$$

where N denotes the number of instances. θ_i is a problem data corresponding to an instance of (2.6). $s(\theta_i)$ denotes the true label corresponding to θ_i. The following outlines the process of generating a data point $(\theta_i, s(\theta_i))$.

1. Sample θ_i by a given probability distribution $P(\theta_i)$.
2. Solve the MICO instance associated with θ_i to identify the optimal solution (denoted by $x^*(\theta_i)$) and tight constraints $\tau(\theta_i)$.
3. Combine the optimal value of the binary variables $x^*_{\mathcal{I}}(\theta_i)$ and $\tau(\theta_i)$, i.e., $s(\theta_i) = (x^*_{\mathcal{I}}(\theta_i), \tau(\theta_i))$.

Here, $\tau(\theta_i)$ is the index set of constraints that are equalities at optimality, with the following formal definition,

$$\tau(\theta_i) = \{j \in \{1, 2, \ldots, m\} | g_j(x^*(\theta_i), \theta_i) = 0\}. \tag{2.8}$$

The NN model is represented by

$$NN(\theta_i; w) = \hat{s}(\theta_i), \tag{2.9}$$

where w is trainable parameters. The model inputs θ_i and predicts $\hat{s}(\theta_i)$. The NN model is trained by the dataset $\{(\theta_i, s(\theta_i))\}_{i=1}^N$ with the following loss function defined as

$$\ell(w) = \sum_{i=1}^{N} \|NN(\theta_i; w) - s(\theta_i)\| . \tag{2.10}$$

Prediction Phase : Given a new parameter θ drawn from the same training probability distribution $P(\theta)$, the trained NN predicts $\hat{s}(\theta)$. Using $\hat{s}(\theta)$, the MICO problem (2.6) becomes the following

$$\begin{aligned} \min_{x} \quad & f(\theta, x) \\ \text{s.t.} \quad & g_i(\theta, x) \leq 0, \quad \forall i \in \mathcal{T}(\theta) \\ & x_{\mathcal{I}} = x^{\star}_{\mathcal{I}}(\theta), \end{aligned} \tag{2.11}$$

Solving (2.11) is much easier than (2.6), since redundant constraints are no longer imposed. Bertsimas & Stellato [141] show that large dimensionality MICO problems can be solved in just milliseconds.

We summarize some of the following work on *Voice of Optimization*. As the dimensionality of the problem increases, the number of different strategies in the training set also becomes excessively large, posing a challenge to the classification task. Bertsimas & Stellato [141] proposed a pruning algorithm to reduce this number, albeit at the cost of some performance compromise. Bertsimas & Kim [149] employed an optimal-tree based prescriptive algorithm instead of neural networks to solve the MICO problems. *Voice of Optimization* has demonstrated its applicability in adaptive robust optimization [150] and robot planning [151] problems.

2.3.2 . Deep Unfolding

Deep unfolding or deep unrolling are emerging methods that are typically used to solve inverse problems and have found many applications in the field of signal processing [152]. These methods exploit the strengths of deep learning to accelerate and improve the performance of iterative algorithms. By unrolling iterative schemes into a finite number of layers in a deep NN, they enable the direct learning of algorithmic parameters from data. This transformative approach not only preserves the interpretability of traditional methods but also benefits from the adaptability and generalization power of NNs. Through the lens of these methods, we gain the capacity to address the inherent challenges of inverse problems, such as ill-posedness and underdetermined systems, with newfound efficiency and robustness. Below, we specifically demonstrate how a deep unfolding method known as LISTA [142] solves linear inverse problems.

Consider the linear inverse problem, studied in [153, 154, 142, 143, 144], with the following form

$$b = Ax^* + \epsilon, \tag{2.12}$$

where

- $x^* \in \mathbb{R}^n$ is the sought signal or image.
- $b \in \mathbb{R}^m$ is the observed data.
- $\epsilon \in \mathbb{R}^m$ is an additive Gaussian white noise.
- $A \in \mathbb{R}^{m \times n}$ is the observation operator, which is assumed known.

The objective is to recover x^* given the observation b. The linear operator A is learned from a physical model or prior identification step.

Typically, we have $m \ll n$, and the problem defined in (2.12) is an ill-posed and highly under-determined system. Nevertheless, the problem becomes tractable if x^* is assumed to be sparse ; that is, the cardinality of the support $S = \{i | x_i^* \neq 0\}$ is small relative to n.

The LASSO formulation, a widely adopted method for addressing this problem, is expressed as

$$\min_x \frac{1}{2}\|b - Ax\|_2^2 + \lambda \|x\|_1, \tag{2.13}$$

where λ is a scalar regularization parameter. $\|b - Ax\|_2^2$ is used to measure the fidelity of the reconstructed signal to the observed data, ensuring that the solution

is consistent with the observations, and $\|x\|_1$ is used to encourage the sparsity of x.

The iterative shrinkage thresholding algorithm (ISTA) [153, 154] is a popular choice to solve (2.13), with the following update rule

$$x^{k+1} = \eta_{\lambda/L}\left(x^k + \frac{1}{L}A^T(b - Ax^k)\right), \quad k = 0, 1, \ldots, K-1, \tag{2.14}$$

where K is the maximum number of iterations, and L is typically chosen as the largest eigenvalue of A^TA. $\eta_\theta(\cdot)$ is the soft-thresholding function, defined in an element-wise way

$$\eta_\theta(x) = \text{sign}(x)\max(0, |x| - \theta). \tag{2.15}$$

Gregor & LeCun [142] reinterpreted ISTA as a recurrent neural network (RNN) with parameters $W_1 = \frac{1}{L}A^T$, $W_2 = I - \frac{1}{L}A^TA$, and $\theta = \frac{\lambda}{L}$. Then, (2.14) becomes

$$x^{k+1} = \eta_\theta\left(W_1 b + W_2 x^k\right), \quad k = 0, 1, \ldots, K-1. \tag{2.16}$$

Buidling upon this reinterpretation, they proposed Learned-ISTA (LISTA) that unrolls the RNN, given by

$$x^{k+1} = \eta_{\theta^k}\left(W_1^k b + W_2^k x^k\right), \quad k = 0, 1, \ldots, K-1. \tag{2.17}$$

As a result the model (2.17) is a K-layer feed-forward NN, with $\Theta = \{W_1^k, W_1^k, \theta^k\}_{k=0}^{K-1}$ as trainable parameters.

In ISTA, all parameters are predefined except for the hyperparameter λ to be tuned. In comparison, LISTA treats the weights and thresholds as learnable parameters and optimize them through stochastic gradient descent. The training dataset $\{(x_i^*, b_i)\}_{i=1}^N$ is sampled from a predefined distribution P. The corresponding training objective is formulated as

$$\min_\Theta \mathbb{E}_{(x^*,b)\sim P}\left\|x^* - x^K(\Theta, b, x^0)\right\|_2^2, \tag{2.18}$$

where x^0 is the initial point as a hyperparameter.

After training, the LISTA model (2.17) has the ability to recover a new x from the noisy data b which are from the same training distribution P. In addition, empirical results [142, 143, 155, 156] have demonstrated that a sufficiently trained K-layer LISTA network can match or exceed the generalization capabilities of ISTA, often achieving comparable reconstruction of x' from b with significantly fewer iterations and improved accuracy of the outputs across the layers.

As a follow-up, Chen et. al [143] proposed a variant called LISTA-CPSS (LISTA-coupling weight and support selection). They provided a linear convergence guarantee for the proposed algorithm under certain conditions on the sampling distribution P. Liu et al. [144] analytically characterized optimal network parameters by

imposing mutual incoherence conditions on the network weights. Analytical derivation of the optimal parameters helps reduce the parameter dimensionality to a large extent. Furthermore, the authors demonstrated that a network with analytic parameters can be as effective as a network trained completely from data.

3 - Solving Bimatrix Games with Bi-channel Convolutional Neural Networks

In this chapter, we consider the problem of finding a Nash equilibrium in a bimatrix game. We use a bi-channel convolutional neural network that takes the bimatrix game as an input and generates two mixed strategies as a predicted Nash equilibrium. Experimental results show that our approach is much faster than traditional solution algorithms, including the Lemke-Howson algorithm and enumeration methods, but at the cost of sacrificing some accuracy.

This chapter corresponds to the publication [145].

3.1 . Introduction

This chapter focuses on two-player general-sum games with finite actions, also known as bimatrix games, which are fundamental models in non-cooperative game theory [9]. Bimatrix games have numerous applications in engineering and economics, helping decision makers to analyze and make rational choices in competitive environments.

The Nash equilibrium is a common solution concept for bimatrix games. It represents a state in which no player has an incentive to unilaterally change his strategy to increase his payoff. For two-player zero-sum games, von Neumann [157] established the minimax theorem, which states that such games always have an equilibrium point known as a saddle point. For n-player general-sum games, Nash [10] proved the existence of a Nash equilibrium. Subsequently, various forms of equilibrium in non-cooperative game theory have been developed through fixed point theorems [11, 158].

However, despite significant advances in equilibrium existence theorems in game theory, computing Nash equilibria remains challenging. Two-player zero-sum games can be reformulated as linear programming (LP) problems [2], which can be solved in polynomial time using interior-point methods [159, 160]. However, there is no polynomial time algorithm for bimatrix games, and computing a Nash equilibrium in a bimatrix game falls into the PPAD complexity class [1, 14].

Another research topic covered in this chapter is convolutional neural networks (CNNs). Due to the rapid growth of data availability and computational resources, deep learning has been applied across various domains, such as computer vision [16, 161, 162], natural language processing [19, 163], and recommender systems [164]. CNNs are deep neural networks with many applications in image processing [22, 165]. Following the remarkable success of AlexNet [17] in the ImageNet challenge [18], research on neural network architectures has rapidly advanced to improve computational performance [166, 167, 168, 169, 170]. In addition, specia-

lized network architectures have been developed for various application purposes [171, 172, 173]. For example, by combining with physics-informed neural networks [26] or multigrid methods [174], CNNs have been used to solve partial differential equations [175, 176, 177].

Lemke-Howson Algorithm. The Lemke-Howson algorithm (LH algorithm for short) is a well-established method for finding a Nash equilibrium of a bimatrix game [29]. Despite being in use for several decades, it remains the most efficient method available. In each iteration, the LH algorithm performs an integer pivoting operation until a Nash equilibrium is found. The final Nash equilibrium determined by the method depends on a chosen parameter called the initially dropped label. The algorithm ensures that a Nash equilibrium of a bimatrix game can be computed. However, the complexity of this algorithm implies that the required number of iterations grows exponentially with respect to the game size, even in the best-case scenario [30, 1].

Enumeration Methods. In contrast to the LH algorithm, which finds only one Nash equilibrium, support enumeration and vertex enumeration are two methods that identify all Nash equilibria of a bimatrix game. Support enumeration iterates through all possible supports of mixed strategies, and it solves a linear system in each iteration [9]. Vertex enumeration requires finding all vertices of two best response polytopes, which is equivalent to solving many linear systems [9, 178]. Like the LH algorithm, these enumeration methods require exponential time to find an equilibrium [30].

CNNs for Solving Two-Player Zero-Sum Games. Recent research has explored the potential of using CNNs to solve two-player zero-sum games. In [179], the CNN model first predicts the optimal value of a two-player zero-sum game based on player 1's payoff matrix. Then, the corresponding predicted saddle point is obtained by solving a system of linear inequalities. The advantage of this approach is that the trained CNN can quickly predict the optimal value of the game with a relative loss of accuracy. The disadvantage, however, is that it still requires a standard LP solver to solve the system of linear inequalities to obtain the predicted saddle point. Later research improved this method by using the CNN to predict the saddle point directly, thus eliminating the need for an LP solver to handle linear systems [180].

Motivations. Traditional solution methods, such as the LH algorithm and the enumeration methods, solve a bimatrix game iteratively. Specifically, to reach a Nash equilibrium, the LH algorithm requires numerous integer pivoting iterations, while the enumeration methods must solve many linear systems. Consequently, these traditional methods can be computationally intensive, especially when dealing with numerous bimatrix games. Recently proposed CNN approaches show computational advantages [179, 180]. However, they are only applicable to two-player zero-sum games, which are a special case of bimatrix games.

Contributions. The key contributions of this chapter are as follows.

— In contrast to [179, 180], the proposed method can now address the bimatrix game problem. Specifically, we introduce the bi-channel CNN (BiCNN for short), which takes the two payoff matrices of a bimatrix game as input and predicts its Nash equilibrium.

— We propose a new algorithm to train the BiCNN model using a diverse set of bimatrix games with different game sizes and probability distributions. It is worth noting that in [180], the training algorithm uses only a single combination of game size and probability distribution to train the model.

— Compared to traditional solution methods, such as the LH algorithm and enumeration methods, our proposed BiCNN method can solve a bimatrix game directly without any iterative process. Consequently, the BiCNN model outperforms these traditional methods in terms of computational time, especially when solving multiple instances. For example, the BiCNN model solves a single instance more than 5 times faster than the LH algorithm, and solves 10,000 instances more than 5,000 times faster than the LH algorithm.

The rest of this chapter is organized as follows. Section 3.2 introduces the bimatrix game and discusses traditional solution methods for solving it. Section 3.3 presents the design of the BiCNN model and its loss function. Section 3.4 describes the process of training the BiCNN model, including the generation of training data and the training algorithm. Section 3.5 provides the implementation details of the proposed method and compares the computational performance of the BiCNN model with that of traditional methods. Finally, Section 3.6 gives a summary of this work.

3.2 . Preliminaries

A two-player general-sum game, or a bimatrix game, is represented by a tuple $\left(N, (S_i)_{i\in N}, (u_i)_{i\in N}\right)$, where

— $N = \{1, 2\}$ is the set of players, denoted by player 1 and player 2.

— $S_1 = \{1, \ldots, m\}$ and $S_2 = \{1, \ldots, n\}$ are nonempty finite sets of pure strategies for player 1 and player 2, respectively. m and n represent the number of pure strategies for player 1 and player 2, respectively. Throughout this chapter, we use (m, n) to represent the game size of a bimatrix game.

— Let $S := S_1 \times S_2$ denote the set of pure strategy profiles. $u_1 : S \to \mathbb{R}$ and $u_2 : S \to \mathbb{R}$ are payoff functions for player 1 and player 2, respectively.

The two payoff functions, u_1 and u_2, can be represented by two payoff matrices

$\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{B} \in \mathbb{R}^{m\times n}$, respectively, as follows :

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \tag{3.1}$$

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{bmatrix}. \tag{3.2}$$

If player 1 chooses the i-th pure strategy and player 2 chooses the j-th pure strategy, player 1 and player 2 receive payoffs of a_{ij} and b_{ij}, respectively. These two matrices encapsulate all the information needed for a bimatrix game, including the game size and the payoff functions. Consequently, a bimatrix game can be concisely represented by its two payoff matrices, denoted as $\Gamma := (\mathbf{A}, \mathbf{B})$.

A mixed strategy is a discrete probability distribution over the set of pure strategies. Let $\mathbf{x} \in X$ and $\mathbf{y} \in Y$ represent the mixed strategies of player 1 and 2, respectively. $X = \{\mathbf{x} \in \mathbb{R}^m \mid \mathbf{1}_m^T\mathbf{x} = 1, \mathbf{x} \geq \mathbf{0}\}$ and $Y = \{\mathbf{y} \in \mathbb{R}^n \mid \mathbf{1}_n^T\mathbf{y} = 1, \mathbf{y} \geq \mathbf{0}\}$ are feasible sets, where $\mathbf{1}_m^T = [1, 1, \dots, 1]^T \in \mathbb{R}^m$ and $\mathbf{1}_n^T = [1, 1, \dots, 1]^T \in \mathbb{R}^n$. The support of a mixed strategy consists of the pure strategies with positive probability.

A mixed strategy $\mathbf{x}^*$ of player 1 is considered a best response to the given mixed strategy $\mathbf{y}$ of player 2 if $\mathbf{x}^*$ maximizes player 1's expected payoff, denoted as $\mathbf{x}^* := \arg\max_{\mathbf{x}\in X} \mathbf{x}^T\mathbf{A}\mathbf{y}$. Similarly, a best response $\mathbf{y}^*$ of player 2 to the given $\mathbf{x}$ maximizes player 2's expected payoff, denoted as $\mathbf{y}^* := \arg\max_{\mathbf{y}\in Y} \mathbf{x}^T\mathbf{B}\mathbf{y}$.

Definition 3.1 *Consider a bimatrix game $\Gamma = (\mathbf{A} \in \mathbb{R}^{m\times n}, \mathbf{B} \in \mathbb{R}^{m\times n})$. A mixed strategy profile $(\mathbf{x}^*, \mathbf{y}^*)$, $\mathbf{x}^* \in X$ and $\mathbf{y}^* \in Y$, is called a Nash equilibrium if the following holds*

$$\begin{aligned} (\mathbf{x}^*)^T \mathbf{A}\mathbf{y}^* &\geq \mathbf{x}^T\mathbf{A}\mathbf{y}^* \quad \forall \mathbf{x} \in X, \\ (\mathbf{x}^*)^T \mathbf{B}\mathbf{y}^* &\geq (\mathbf{x}^*)^T\mathbf{B}\mathbf{y} \quad \forall \mathbf{y} \in Y. \end{aligned} \tag{3.3}$$

Theorem 3.1 ([10]) *Any game with a finite set of players and a finite set of pure strategies per player has a Nash equilibrium of mixed strategies.*

According to Theorem 3.1, for any bimatrix game, there always exists at least one Nash equilibrium $(\mathbf{x}^*, \mathbf{y}^*)$. The corresponding optimal values v_1^* and v_2^* for players 1 and 2 are denoted as :

$$\begin{aligned} v_1^* &:= (\mathbf{x}^*)^T\mathbf{A}\mathbf{y}^*, \\ v_2^* &:= (\mathbf{x}^*)^T\mathbf{B}\mathbf{y}^*. \end{aligned} \tag{3.4}$$

Definition 3.2 *A bimatrix game is called nondegenerate if no mixed strategy of support size k has more than k best responses of pure strategies.*

Traditional Methods 1 : Support enumeration Input : a nondegenerate bimatrix game $(\mathbf{A} \in \mathbb{R}^{m\times n}, \mathbf{B} \in \mathbb{R}^{m\times n})$. Output : all Nash equilibria. The solution procedure is as follows.

1. For $k = 1, \ldots, min\{m, n\}$, generate two sets (I, J) which are k-sized subsets of S_1 and S_2, respectively.
2. Solve the following linear system with the variable vectors $(\mathbf{x}, \mathbf{y}, v, u)$,

$$\begin{array}{c} \sum_{i\in I} \mathbf{x}_i b_{ij} = v \quad \forall j \in J, \\ \sum_{j\in J} a_{ij}\mathbf{y}_j = u \quad \forall i \in I, \\ \mathbf{1}_m^T\mathbf{x} = 1, \quad \mathbf{1}_n^T\mathbf{y} = 1, \quad \mathbf{x} \geq \mathbf{0}, \quad \mathbf{y} \geq \mathbf{0}, \end{array} \tag{3.5}$$

   where $\mathbf{x}_i$ and $\mathbf{y}_j$ denote the i-th and j-th element of $\mathbf{x}$ and $\mathbf{y}$, respectively.
3. The solution vector $(\mathbf{x}, \mathbf{y})$ that solves Eq. (3.5) is a Nash equilibrium.

Traditional Methods 2 : Vertex enumeration Input : a nondegenerate bimatrix game $(\mathbf{A} \in \mathbb{R}^{m\times n}, \mathbf{B} \in \mathbb{R}^{m\times n})$. Output : all Nash equilibria. This method is based on two concepts, i.e., completely labeled and best response polytopes. We say $\mathbf{x}$ has label k if one of the following two conditions hold. (a) If $k \in \{1, \ldots, m\}$, $x_k = 0$. (b) If $k \in \{m+1, \ldots, m+n\}$, $(\mathbf{B}^T\mathbf{x})_k = \max\left\{(\mathbf{B}^T\mathbf{x})_k \mid k \in S_2\right\}$. Similarly, $\mathbf{y}$ has label k if one of the following two conditions hold. (a) If $k \in \{1, \ldots, n\}$, $\mathbf{y}_k = 0$. (b) If $k \in \{n+1, \ldots, n+m\}$, $(\mathbf{A}\mathbf{y})_k = \max\{(\mathbf{A}\mathbf{y})_k \mid k \in S_1\}$. A mixed strategy profile $(\mathbf{x}, \mathbf{y})$ is called completely labeled if every label $k \in \{1, \ldots, m+n\}$ appears in either $\mathbf{x}$ or $\mathbf{y}$. The two best response polytopes are denoted as

$$\begin{aligned} P &= \left\{x \in \mathbb{R}^m \mid \mathbf{B}^\top x \leq \mathbf{1}, \quad x \geq \mathbf{0}\right\}, \\ Q &= \left\{y \in \mathbb{R}^n \mid \mathbf{A}y \leq \mathbf{1}, \quad y \geq \mathbf{0}\right\}. \end{aligned} \tag{3.6}$$

The solution procedure is as follows.

1. Obtain the polytopes P and Q of the game according to Eq. (3.6).
2. Find all vertices in P and Q.
3. Check each pair of vertices $(\mathbf{x} \in P, \mathbf{y} \in Q)$, except $(\mathbf{0}, \mathbf{0})$. If $(\mathbf{x}, \mathbf{y})$ is completely labeled, output a Nash equilibrium $\left(\mathbf{x}/\mathbf{1}^\top\mathbf{x}, \mathbf{y}/\mathbf{1}^\top\mathbf{y}\right)$.

Traditional Methods 3 : The LH algorithm Input : a nondegenerate bimatrix game $(\mathbf{A} \in \mathbb{R}^{m\times n}, \mathbf{B} \in \mathbb{R}^{m\times n})$. Output : one Nash equilibrium. The solution procedure is as follows.

1. Obtain the polytopes P and Q of the game according to Eq. (3.6).
2. Start with $(\mathbf{x}, \mathbf{y}) = (\mathbf{0}, \mathbf{0})$, which is completely labeled. Choose an initially dropped label $k \in \{1, 2, \ldots, m+n\}$, and set $l_{\text{old}} = k$.

3. Drop the label l_{old} and move to a new pair of vertices $(\mathbf{x} \in P, \mathbf{y} \in Q)$ of the polytopes. $(\mathbf{x}, \mathbf{y})$ has a new label l_{new}.
4. If $l_{\text{new}} = k$, return the Nash equilibrium $\left(\mathbf{x}/\mathbf{1}^\top\mathbf{x}, \mathbf{y}/\mathbf{1}^\top\mathbf{y}\right)$. Otherwise, set $l_{\text{new}} = l_{\text{old}}$ and repeat step 3.

Note that in step 3, the label replacement corresponds to an integer pivoting operation [2].

For more details on these three traditional solution methods, we refer the reader to [9].

3.3 . BiCNN Model

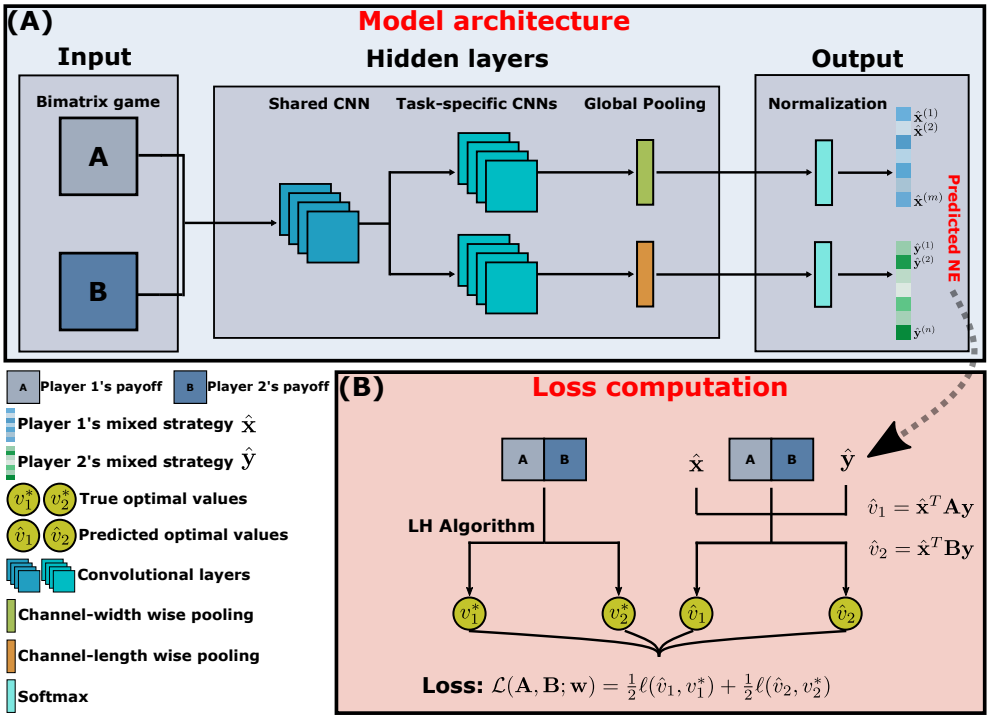


Figure 3.1 – An overview of the proposed BiCNN model. (A) BiCNN model architecture, where the input $(\mathbf{A}, \mathbf{B})$ is a bimatrix game represented by its two payoff matrices, and the output $(\hat{\mathbf{x}}, \hat{\mathbf{y}})$ is a predicted Nash equilibrium. (B) Loss computation, where the predicted optimal values $(\hat{v}_1, \hat{v}_2)$ are calculated using Eq. (3.9), and the true optimal values (v_1^*, v_2^*) are obtained using the Lemke-Howson (LH) algorithm.

Input and Output. We propose a Bi-channel CNN (BiCNN) model that takes a bimatrix game as input and predicts its Nash equilibrium. The BiCNN model receives a two-channel array representing the payoff matrices of player 1

and player 2 in a bimatrix game, respectively. The BiCNN model can be expressed as :

$$h(\mathbf{A}, \mathbf{B}; \mathbf{w}) = (\hat{\mathbf{x}}, \hat{\mathbf{y}}). \tag{3.7}$$

$(\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{B} \in \mathbb{R}^{m \times n})$ is a bimatrix game to solve, where $\mathbf{A}$ is player 1's payoff matrix and $\mathbf{B}$ is player 2's payoff matrix. $h(\mathbf{A}, \mathbf{B}; \mathbf{w})$ represents a neural network with trainable parameters $\mathbf{w}$. $(\hat{\mathbf{x}} \in \mathbb{R}^m, \hat{\mathbf{y}} \in \mathbb{R}^n)$ denotes a predicted Nash equilibrium for the input bimatrix game, where $\hat{\mathbf{x}}$ is the predicted mixed strategy of player 1 and $\hat{\mathbf{y}}$ is the predicted mixed strategy of player 2.

Double Branches Architecture. As shown in Fig. 3.1(A), the model consists of two parts : a shared CNN and two task-specific CNNs. The shared CNN first transforms the input bimatrix game into shared hidden feature maps. The two task-specific CNNs then process the shared feature maps separately to generate the predicted mixed strategies $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$. A key feature of the BiCNN model is its ability to solve bimatrix games of different sizes. The model takes in two matrices $(\mathbf{A}, \mathbf{B})$ with arbitrary shapes $(m \in \mathbb{R}, n \in \mathbb{R})$ and outputs two corresponding vectors with m and n dimensions. This feature is made possible by two important techniques : (a) fully convolutional layer design, and (b) two global pooling techniques.

Spatial Shape Preservation. The BiCNN model consists entirely of convolutional layers, omitting fully connected layers. A convolutional layer with certain settings preserves the spatial shape of the feature map. For example, consider an input feature map with a spatial shape of $(10, 10)$ and a convolutional layer with a kernel size of $3 * 3$, stride of 1, and padding of 1. After processing through this convolutional layer, the feature map retains its spatial shape of $(10, 10)$. Each convolutional layer in the BiCNN model uses such a setup to preserve the spatial shape of a bimatrix game from input to output.

Width-wise and Length-wise Pooling. As shown in the hidden layer of Fig. 3.1(A), the outputs of the two task-specific CNNs are compressed by width-wise and length-wise pooling, respectively. The two final feature maps have the shapes (c_1, m, n) and (c_2, m, n), respectively, where c_1 and c_2 represent the number of channels, and (m, n) corresponds to the game size of the input bimatrix game. Width-wise pooling compresses the channel and width directions, resulting in an m-dimensional vector. Similarly, length-wise pooling compresses the channel and length directions, resulting in an n-dimensional vector. These two vectors are then normalized using the softmax function to become discrete probability distributions, which represent the predicted mixed strategies.

Loss Function. The loss function for the proposed BiCNN model is designed as follows :

$$\mathcal{L}(\mathbf{A}, \mathbf{B}; \mathbf{w}) = \frac{1}{2}\ell(\hat{v}_1, v_1^*) + \frac{1}{2}\ell(\hat{v}_2, v_2^*), \tag{3.8}$$

where

$$\begin{aligned} \hat{v}_1 &= \hat{\mathbf{x}}^T \mathbf{A} \hat{\mathbf{y}}, \\ \hat{v}_2 &= \hat{\mathbf{x}}^T \mathbf{B} \hat{\mathbf{y}}. \end{aligned} \tag{3.9}$$

$\hat{v}_1$ and $\hat{v}_2$ represent the predicted optimal values for player 1 and player 2, respectively, obtained from the predicted Nash equilibrium $(\hat{\mathbf{x}}, \hat{\mathbf{y}})$. v_1^* and v_2^* are the true optimal values, obtained from the true Nash equilibrium and calculated using Eq. (3.3). $\ell(\cdot, \cdot)$ denotes an error metric, such as mean square error.

Motivation for the Loss Design. The loss function measures the error on the predicted optimal values rather than the predicted Nash equilibrium for the following reasons. (a) A bimatrix game typically has multiple Nash equilibria, and fitting to any one of them would be unfair to the others. Using the optimal value circumvents this issue, as it is unique. (b) Training to minimize the gap between two scalars is easier than reducing the gap between two vectors, i.e., approximating $(\hat{v}_1, \hat{v}_2)$ to (v_1^*, v_2^*) is simpler than approximating $(\hat{\mathbf{x}}, \hat{\mathbf{y}})$ to $(\mathbf{x}^*, \mathbf{y}^*)$.

3.4 . Model Training

In this section, we describe the training process for the BiCNN model. Similar to [179, 180], due to the lack of standard and available datasets, we randomly generate bimatrix games for training and testing. The generation of a bimatrix game is determined by two predefined parameters : game size and probability distribution. To improve the ability of the BiCNN model to solve a wide range of bimatrix games, we design an objective function that takes into account different game sizes and probability distributions. We propose a training algorithm that aims to minimize this objective function.

3.4.1 . Bimatrix Game Generation

A Training Sample. A sample used to train the BiCNN model has the form of $(\mathbf{A}_{mn}, \mathbf{B}_{mn}, v_1^*, v_2^*)$, where

— $(\mathbf{A}_{mn}, \mathbf{B}_{mn})$ represents a bimatrix game with game size (GS) of (m, n), where $\mathbf{A}_{mn} \in \mathbb{R}^{m \times n}$ and $\mathbf{B}_{mn} \in \mathbb{R}^{m \times n}$.

— $(\mathbf{A}_{mn}, \mathbf{B}_{mn})$ is generated by a given probability distribution (PD), such as the uniform distribution with interval $[0, 100]$, denoted as $U(0, 100)$. We use the notation $\mathbf{A}_{mn} \sim P$ to indicate that each element within the matrix $\mathbf{A}_{mn}$ is sampled by P. $\mathbf{B}_{mn} \sim P$ has the same meaning.

— (v_1^*, v_2^*) are the optimal values corresponding to the bimatrix game $(\mathbf{A}_{mn}, \mathbf{B}_{mn})$. Traditional solution methods, such as the LH algorithm, are used to solve the bimatrix game and obtain the Nash equilibrium $(\mathbf{x}^*, \mathbf{y}^*)$. Then, the optimal values (v_1^*, v_2^*) are computed using Eq. (3.4).

GS-PD Combination. A GS-PD combination determines how a bimatrix game is generated. It takes the form of GS : (m, n) and PD : P, where (m, n) specifies the game size, and P specifies how the two payoff matrices are sampled. A GS-PD combination is considered trained if it is used to generate training data of bimatrix games, and untrained otherwise.

Robustness of BiCNN. In this chapter, we define the model robustness as its ability to solve bimatrix games generated from untrained GS-PD combinations. To improve model robustness, we train the model with multiple GS-PD combinations, which means that the training bimatrix games have different game sizes and are sampled from different probability distributions. This is in contrast to the previous study [180], where the model is trained with only one GS-PD combination.

3.4.2 . Objective Function with Multiple GS-PDs

Objective Function. The objective function that takes into account multiple training GS-PDs is formulated as follows :

$$E(\mathbf{w}) = \sum_{(m,n)\in\mathbb{G}} \sum_{P\in\mathbb{P}} \mathbb{E}_{\mathbf{A}_{mn}\sim P, \mathbf{B}_{mn}\sim P}[\mathcal{L}(\mathbf{A}_{mn}, \mathbf{B}_{mn}; \mathbf{w})], \tag{3.10}$$

where $\mathbb{G}$ is a set containing multiple GSs. $\mathbb{P}$ is a set containing multiple PDs. $|\mathbb{G}|$ and $|\mathbb{P}|$ denote the sizes of $\mathbb{G}$ and $\mathbb{P}$, respectively. The objective function $E(\mathbf{w})$ has $|\mathbb{G}| * |\mathbb{P}|$ terms, and each term $\mathbb{E}_{\mathbf{A}_{mn}\sim P, \mathbf{B}_{mn}\sim P}[\mathcal{L}(\mathbf{A}_{mn}, \mathbf{B}_{mn}; \mathbf{w})]$ represents an expected risk with respect to a GS-PD combination.

Batch Loss. However, the objective function $E(\mathbf{w})$ is computationally intractable even though the PDs of the expectation are known. Therefore, in practice, the model is trained with a batch loss that serves as an estimate of the objective function (3.10). The batch loss is as follows :

$$\mathcal{L}_{\mathcal{B}}(\mathbf{w}) = \frac{1}{|\mathcal{B}|} \sum_{(\mathbf{A}^{(i)}_{mn}, \mathbf{B}^{(i)}_{mn})\in\mathcal{B}} \mathcal{L}(\mathbf{A}^{(i)}_{mn}, \mathbf{B}^{(i)}_{mn}; \mathbf{w}), \tag{3.11}$$

where $|\mathcal{B}|$ denotes the batch size. $\mathcal{B} = \{(\mathbf{A}^{(1)}_{mn}, \mathbf{B}^{(1)}_{mn}), (\mathbf{A}^{(2)}_{mn}, \mathbf{B}^{(2)}_{mn}), \ldots, (\mathbf{A}^{|\mathcal{B})|}_{mn}, \mathbf{B}^{(|\mathcal{B})|}_{mn})\}$ represents a batch of training bimatrix games, and each training sample $(\mathbf{A}^{(i)}_{mn}, \mathbf{B}^{(i)}_{mn})$ in the batch has the same GS and is sampled from the same PD. The batch loss $\mathcal{L}_{\mathcal{B}}(\mathbf{w})$ represents the average of Eq. (3.8) across all bimatrix games in the batch $\mathcal{B}$.

3.4.3 . Training Algorithm with Multiple GS-PDs

Multi-GS-PD Training Algorithm. Alg. 1 outlines the training process for the BiCNN model, which incorporates multiple GS-PDs and focuses on optimizing the objective function described in Eq. (3.10). The training GSs and PDs are represented by $\mathbb{G}$ and $\mathbb{P}$, respectively, which are traversed by the two *for-loops* in the algorithm. In each iteration, a combination of GS-PD, i.e., $(m, n) \in \mathbb{G}$ and $P \in \mathbb{P}$, is selected to generate a batch of training bimatrix games. The LH algorithm is used to solve these bimatrix games and obtain the true optimal values, which are then used in the loss function according to Eq. (3.8).

Bimatrix Game Generation Rule. Unlike traditional machine learning tasks, Alg. 1 does not rely on a given training dataset. Instead, it relies on a bimatrix game generation rule, specifically the set of GS-PDs, to generate the bimatrix games for

Algorithm 1 BiCNN Training with Multiple GSs and PDs

Require: $\mathbb{G}$: a set of training GSs ; $\mathbb{P}$: a set of training PDs

Initialize a BiCNN model denoted as $h(\cdot\,;\mathbf{w})$
while iter $\leq$ Max iteration **do**
  for $(m,n) \in \mathbb{G}$ **do**
    for $P \in \mathbb{P}$ **do**
      Generate a batch of bimatrix games : $\mathcal{B} \sim P$
      Predict Nash equilbria : $h(\mathcal{B};\mathbf{w})$
      Calculate predicted optimal values by Eq. (3.9)
      Solve the batch $\mathcal{B}$ by the LH algorithm : LH$(\mathcal{B})$
      Compute $\mathcal{L}_{\mathcal{B}}(\mathbf{w})$
      Update $\mathbf{w}$ by $\nabla_{\mathbf{w}}\mathcal{L}_{\mathcal{B}}(\mathbf{w})$
    end for
  end for
end while

training. During each iteration, the model is trained on a batch of bimatrix games generated according to this rule. The algorithm can continue indefinitely without training on the same bimatrix game twice, since the probability of sampling a bimatrix game with the same PD more than once is extremely small.

Learning from the LH Algorithm. In addition to the bimatrix game generation rule, a solution algorithm is needed to solve these games and obtain the true values. Alg. 1 uses the LH algorithm for this purpose. The model learns the LH algorithm in the context of a specific bimatrix game generation rule. The generalization ability of the model depends on the pre-specified generation rule and the quality of its training. Once training is complete, the model can predict the Nash equilibrium for an unseen bimatrix game generated by the same rule used for training, and the BiCNN model makes the prediction without the need for the LH algorithm.

3.4.4 . Error Analysis

Notations Setup. This subsection presents an error analysis for the BiCNN model. For simplicity, we focus on the single GS-PD pair scenario, but note that the analysis can be easily generalized to scenarios involving multiple GS-PD pairs. Before delving into the specifics of the analysis, we first establish some basic definitions to facilitate our discussion.

— $\mathcal{L}_{\mathcal{B}}$: Represents the empirical risk, where $\mathcal{B}$ refers to a training dataset comprised of bimatrix games, and each training game is sampled by the PD of $\mathcal{D}$. $\mathcal{L}_{\mathcal{B}}$ is defined in Eq. (3.11) with only one GS-PD pair.

— $\mathcal{L}_{\mathcal{D}}$: Represents the expected risk associated with the PD of $\mathcal{D}$. Similarly, $\mathcal{L}_{\mathcal{D}}$ is defined in Eq. (3.10) with only one GS-PD pair is considered.

— $\mathcal{H}$: Represents a class of BiCNN models that share a similar architecture as shown in Fig. 3.1-(A). These models must be implementable, i.e., they should contain a certain number of hidden layers and neurons, and their model parameters must be finite.

— $\mathcal{G}$: Represents a more general class of models, which may or may not be implementable, but which necessarily includes $\mathcal{H}$, i.e., $\mathcal{H} \subset \mathcal{G}$. For example, $\mathcal{G}$ could be a network architecture with an infinite number of hidden layers and neurons.

Three Predictors. The empirically optimal predictor $h^*_{\mathcal{B}} \in \mathcal{H}$ w.r.t. the empirical risk $\mathcal{L}_{\mathcal{B}}$ is given by

$$h^*_{\mathcal{B}} \in \arg\min_{h \in \mathcal{H}} \mathcal{L}_{\mathcal{B}}(h). \tag{3.12}$$

The expected optimal predictor $h^* \in \mathcal{H}$ w.r.t. the expected risk $\mathcal{L}_{\mathcal{B}}$ is given by

$$h^* \in \arg\min_{h \in \mathcal{H}} \mathcal{L}_{\mathcal{D}}(h). \tag{3.13}$$

The universal optimal predictor $g^* \in \mathcal{G}$ is given by

$$g^* \in \arg\min_{g \in \mathcal{G}} \mathcal{L}_{\mathcal{D}}(g). \tag{3.14}$$

In terms of the expected risk $\mathcal{L}_{\mathcal{D}}$, the three predictors $h^*_{\mathcal{B}}$, h^*, and g^* have the following relationship,

$$\mathcal{L}_{\mathcal{D}}(g^*) \leq \mathcal{L}_{\mathcal{D}}(h^*) \leq \mathcal{L}_{\mathcal{D}}\left(h^*_{\mathcal{B}}\right). \tag{3.15}$$

The first inequality holds because the model class $\mathcal{G}$ contains $\mathcal{H}$, such that $h^* \in \mathcal{G}$, but g^* may not be in $\mathcal{H}$. The second inequality holds because $h^*_{\mathcal{B}}$ is the minimizer of the empirical risk $\mathcal{L}_{\mathcal{B}}$ for a finite set $\mathcal{B}$, while h^* is the minimizer of the expected risk $\mathcal{L}_{\mathcal{D}}$. Even with $\mathcal{L}_{\mathcal{B}}(h^*_{\mathcal{B}}) = 0$, the inequality still holds, since the predictor $h^*_{\mathcal{B}}$ tends to overfit the training dataset $\mathcal{B}$.

Error Decomposition. Let $\tilde{h} \in \mathcal{H}$ be a BiCNN model that one obtain in practice, e.g. the output of Alg. 1 after 10,000 iterations. As shown in Fig. 3.2-(A), the total error from $\tilde{h}$ to the universal optimal predictor g^* is decomposed into three parts : 1) optimization error, 2) estimation error, and 3) approximation error.

— Optimization error : It measures the gap between the BiCNN predictor $\tilde{h}$ and the empirically optimal predictor $h^*_{\mathcal{B}}$. This error is due to the fact that the empirical risk is highly non-convex and therefore it is difficult to find the global optimum, i.e. $h^*_{\mathcal{B}}$. Minimizing this error has been a persistent problem in machine learning, usually involving many technical tricks and tedious trial-and-error hyperparameter tuning. Gradient descent is typically used to minimize this error, and many variants have been proposed to improve performance.

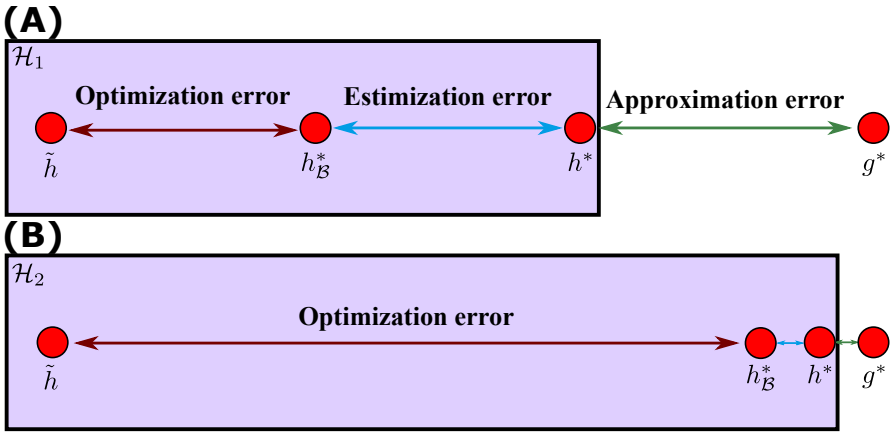


Figure 3.2 – (A) : Illustration of the error decomposition for a BiCNN model, denoted as $\tilde{h}$. (B) : The error decomposition when given an infinite number of iterations and a sufficiently complicated network architecture for Alg. 1.

— Estimation error : This measures the difference between the empirically optimal predictor $h^*_{\mathcal{B}}$ and the expected optimal predictor h^*. This error stems from the use of finite training data of bimatrix games. $h^*_{\mathcal{B}}$ gradually converges to h^* as the provided game dataset $\mathcal{B}$ tends to infinity. In the context of our proposed BiCNN approach, minimizing this error refers to providing more training iterations for Alg. 1, since each iteration generates a new batch of bimatrix games, which is equivalent to expanding $\mathcal{B}$

— Approximation error : This measures the difference between the expected optimal predictor h^* and the universal optimal predictor g^*. This error results from the limited representational capacity of a particular BiCNN model architecture. Reducing this error requires the construction of more complicated network architecture with stronger representational capabilities, such as more hidden layers and neurons.

Infinite Iterations and Complex Model Architecture. Fig. 3.2-(B) shows the scenario where Alg. 1 undergoes infinite iterations and is equipped with a sufficiently complicated network architecture. Under such circumstances, both the estimation error and the approximation error, tend to gradually approach zero. Consequently, the empirically optimal predictor $h^*_{\mathcal{B}}$ converges to the universally optimal predictor g^*. However, this inevitably leads to an increase in the optimization error, which can be difficult to minimize for the BiCNN model. Here, we must emphasize the importance of providing more iterations for Alg. 1. In each iteration, the algorithm not only trains the model, but also generates game data, which reduces both the optimization error and the estimation error.

Open Problems. Our goal is to continuously refine and improve the BiCNN model $\tilde{h}$ to approximate the universally optimal predictor g^*. Despite our progress,

several important problems still remain open in the study of game theory. First, given a model class $\mathcal{H}$, what is the value of $\min_{h\in\mathcal{H}} \mathcal{L}_{\mathcal{D}}(h)$? Second, what is the lower bound of $\min_{g\in\mathcal{G}} \mathcal{L}_{\mathcal{D}}(g)$ for any arbitrary hypothesis class $\mathcal{G}$? Third, is there a hypothesis class $\mathcal{G}$ for which the minimum expected risk, $\min_{g\in\mathcal{G}} \mathcal{L}_{\mathcal{D}}(g)$, is zero ? Answering these questions could potentially reshape our understanding of the limits of model performance. Their answers, once revealed, will undoubtedly deepen our understanding and contribute significantly to the advancement of our BiCNN research.

3.5 . Numerical Results

Experimental Setup. We run our experiments on the Google Colab Pro+ platform with an A100-40GB GPU. The BiCNN model is implemented using PyTorch 1.12.1 with CUDA 11.2 support [181]. Our proposed method is benchmarked against the LH algorithm, support enumeration, and vertex enumeration methods, all of which are implemented in the *Nashpy* library [182].

Training and Testing Data. As described in Section 3.4, we generate bimatrix games for training and testing purposes. A bimatrix game is generated according to a GS-PD combination. For example, a bimatrix game $(\mathbf{A}, \mathbf{B})$ is considered generated by GS : (20, 20) and PD : U(0, 100) if (a) the game has 20 pure strategies for both player 1 and player 2, i.e., $\mathbf{A} \in \mathbb{R}^{20\times 20}$ and $\mathbf{B} \in \mathbb{R}^{20\times 20}$, and (b) each element within the payoff matrices $\mathbf{A}$ and $\mathbf{B}$ is sampled from the uniform distribution $U(0, 100)$. In practice, we generate such a bimatrix game with this GS-PD combination using the *numpy.random.uniform(0, 100, (20, 20))* function in the *NumPy* library [183].

In Section 3.5.1, we present the training process for a BiCNN model, detailing the model architecture, hyperparameter settings, and training loss. Section 3.5.2 and Section 3.5.3 evaluate the performance of the trained BiCNN model in terms of computation time and prediction accuracy, respectively. In Section 3.5.4, we perform an ablation study to evaluate the robustness training property of Alg. 1. In Section 3.5.5, we combine the BiCNN model with the traditional solvers for efficient and accurate solutions. In Section 3.5.6, we discuss the advantages and limitations of our proposed approach.

3.5.1 . Model Training

Model Details and Hyperparameters. We initialize a BiCNN model with the architecture shown in Table 3.1. The training hyperparameters are as follows : the maximum number of iterations is set to 15,000, and the ADAM optimizer [27] is used. The learning rate is set to 0.001 before the 10,000th iteration and to 0.0001 thereafter. The batch size is fixed at 32. We choose the training GS set as $\mathbb{G} = \{(20, 20), (30, 30), (40, 40)\}$ and the training PD set as $\mathbb{P} = \{U(0, 100), N(50, 25)\}$. $U(0, 100)$ denotes the uniform distribution on the interval $[0, 100]$, and $N(50, 25)$ denotes the normal distribution with a mean of 50

Table 3.1 – Details of the BiCNN model. Given a bimatrix game with game size (m, n) as input, the model predicts two mixed strategies : an m-dimensional vector $\hat{x}$ for player 1 and an n-dimensional vector $\hat{y}$ for player 2. Each hidden layer uses leaky relu as the activation function, as well as batch normalization.

Group	Layer	Operator	Details	Input size	Output size
Shared CNN	1	Conv 3*3	8 filters	(2, m, n)	(8, m, n)
	2	Conv 3*3	16 filters	(8, m, n)	(16, m, n)
	3	Conv 3*3	32 filters	(16, m, n)	(32, m, n)
	4	Conv 3*3	64 filters	(32, m, n)	(64, m, n)
Task-specific CNN for Player 1	1	Conv 3*3	32 filters	(64, m, n)	(32, m, n)
	2	Conv 3*3	16 filters	(32, m, n)	(16, m, n)
	3	Conv 3*3	16 filters	(16, m, n)	(16, m, n)
	4	Conv 3*3	8 filters	(16, m, n)	(8, m, n)
	5	Pooling	Channel-width wise global pooling	(8, m, n)	(m,)
Task-specific CNN for Player 2	1	Conv 3*3	32 filters	(64, m, n)	(32, m, n)
	2	Conv 3*3	16 filters	(32, m, n)	(16, m, n)
	3	Conv 3*3	16 filters	(16, m, n)	(16, m, n)
	4	Conv 3*3	8 filters	(16, m, n)	(8, m, n)
	5	Pooling	Channel-length wise global pooling	(8, m, n)	(n,)

and a variance of 25.

Training of the BiCNN Model. We train the BiCNN model using Alg. 1. In each iteration, the algorithm first selects a GS and a PD from $\mathbb{G}$ and $\mathbb{P}$, respectively. The BiCNN model is then trained on the bimatrix games that are generated based on the selected GS-PD combination. In each iteration, the model updates its parameters six times, corresponding to the six different combinations in $\mathbb{G}$ and $\mathbb{P}$. Figs. 3.3 (B-C) shows the losses for the six combinations of GS-PD, while Fig. 3.3 (A) shows the average of these six losses.

Loss Analysis. Due to the characteristics of our training algorithm, our BiCNN model is not explicitly trained on any particular bimatrix game dataset, thus avoiding overfitting to any particular dataset. Notably, the model is constantly trained on fresh, unseen data, and the loss shown in Fig. 3.3 refers to the test loss. Within a span of 15,000 iterations, the loss value shows a significant reduction, converging from an initial value of 1600 to a relatively minimal value of about 250. As explained in Section 3.4.4, this final loss value of 250 is composed of three components : optimization error, estimation error, and approximation error. This final loss value of 250 serves as a measure of the deviation between the performance of the BiCNN model and an exact method, such as the LH algorithm, which can solve the bimatrix game with zero error.

3.5.2 . Computational Efficiency

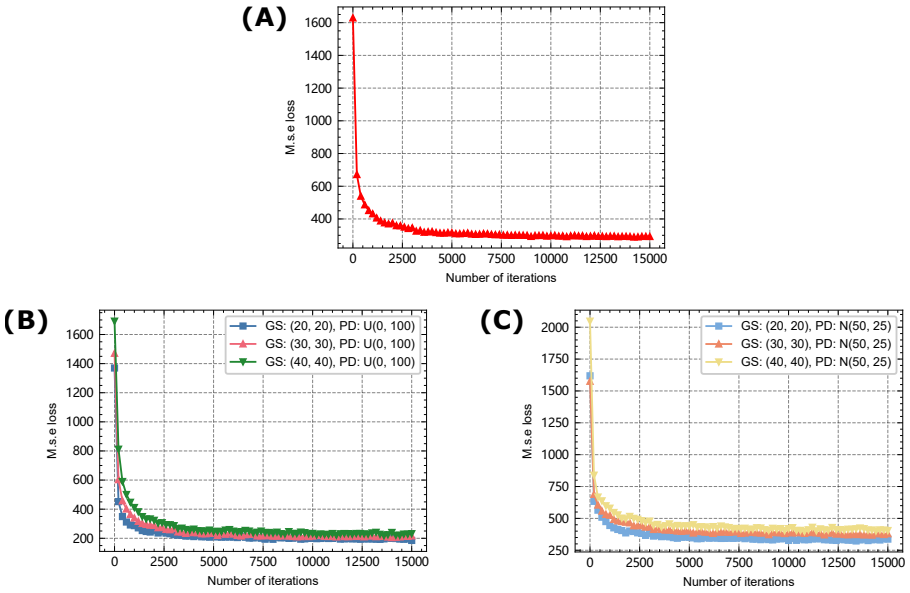


Figure 3.3 – M.S.E. loss versus the number of iterations, where M.S.E. refers to mean square error. In each iteration, there are six updates of the model parameters, corresponding to six loss values. (A) Average of the six losses. (B) The three losses associated with the uniform distribution $U(0, 100)$. (C) The three losses associated with the normal distribution $N(50, 25)$.

Solving a Single Instance. The main advantage of our proposed BiCNN method is that it is much faster than traditional solvers. Fig. 3.4(A) compares the computation time of the BiCNN model with the LH algorithm, support enumeration (SE), and vertex enumeration (VE) for solving a bimatrix game of GS $(20, 20)$ generated by PD $U(0, 100)$. The BiCNN model significantly outperforms these three solvers. On average, the LH algorithm, the most efficient traditional solver, takes 15.1 ms to solve a bimatrix game, while the BiCNN model solves it in only 2.8 ms (5x faster). SE and VE take even more than 1.5 hours to find a Nash equilibrium. In addition, the solution time of BiCNN is more stable than that of the LH algorithm, as indicated by the lower standard deviation. This is because the LH algorithm typically performs a varying number of pivoting operations depending on the instance to reach a Nash equilibrium. In contrast, the solution time of the BiCNN model is independent of the specific instance.

Solving Multiple Instances. When it comes to solving multiple instances of bimatrix games, the computational efficiency of our BiCNN method outperforms the LH algorithm by more than an order of magnitude. We test the computational time of the two methods on solving different numbers of game instances, where each game instance has a GS of $(20, 20)$ and is generated by $U(0, 100)$. As shown in Fig. 3.4(B), the LH algorithm requires a total of 145,213 ms to solve 10,000

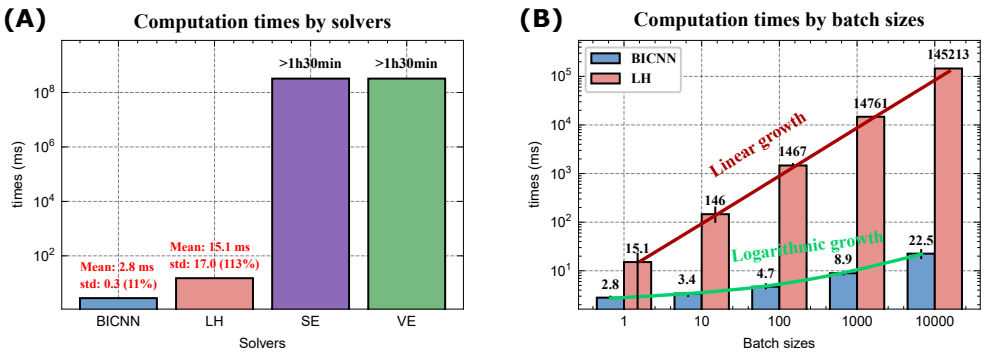


Figure 3.4 – Computational efficiency comparison. Time is measured in milliseconds (ms). (A) Computational times (ms) for solving a single bimatrix game using four solvers : the BiCNN model (our approach), Lemke-Howson algorithm (LH), support enumeration (SE), and vertex enumeration (VE). (B) Computational times (ms) for solving multiple bimatrix games using the BiCNN model and LH algorithm.

instances, while the BiCNN model completes the predictions in only 22.5 ms (6454x faster). Furthermore, the solution time of the LH algorithm increases linearly with the number of instances to be solved. For example, the LH algorithm takes ten times longer to solve 1,000 instances than it does to solve 100 instances. Conversely, the BiCNN model breaks this linear growth, requiring only twice the computational time to solve 1,000 instances compared to 100 instances.

CUDA Support. To the best of our knowledge, traditional algorithms for solving bimatrix games, including the LH algorithm and the enumeration methods, operate on CPUs and do not have CUDA support. This lack of CUDA utilization in the traditional solution algorithms is mainly due to their iterative nature, where each step involves solving sub-optimization problems, making it difficult to utilize CUDA to improve performance. In contrast, our proposed BiCNN model works in a fundamentally different way. The solution process of the BiCNN model for a bimatrix game is equivalent to a single forward propagation of neural networks, which inherently depends on matrix operations. As a result, the BiCNN model is capable to take advantage of CUDA to greatly improve computational efficiency. From the perspective of CUDA support, BiCNN can be seen as a novel method that effectively exploits the power of CUDA, in stark contrast to traditional CPU-based methods.

3.5.3 . Prediction Accuracy

Evaluation Setup. The accuracy of the trained BiCNN model is evaluated by comparing its performance with both the naive method and the LH algorithm,

Table 3.2 – Performance comparison of the BiCNN model, naive method, and LH algorithm on trained GS-PDs. Time is measured in milliseconds. The error metric MAPE is calculated by Eq. (3.16), and the MOV and SOV values are calculated by Eq. (3.17). Each test batch contains 1,000 instances.

Method	GS : (20, 20), PD : U(0, 100)					GS : (20, 20), PD : N(50, 25)				
	Time (ms) ↓	MAPE ↓		MOV(SOV)		Time (ms) ↓	MAPE ↓		MOV(SOV)	
		v_1	v_2	v_1	v_2		v_1	v_2	v_1	v_2
Naive	0	71%±3.5%	72%±3.7%	50.14(1.56)	50.08(1.57)	0	65%±3.2%	66%±3.1%	49.98(1.31)	49.95(1.31)
BiCNN	**8.9±0.9**	**13%±2.4%**	**13%±2.5%**	84.76(8.83)	84.61(9.30)	**9.2±0.8**	**18%±2.7%**	**17%±2.7%**	81.49(10.66)	81.12(9.66)
LH	14000±542	0%	0%	85.91(10.26)	86.18(10.21)	15000±583	0%	0%	82.41(15.75)	82.93(15.33)

Method	GS : (30, 30), PD : U(0, 100)					GS : (30, 30), PD : N(50, 25)				
	Time (ms) ↓	MAPE ↓		MOV(SOV)		Time (ms) ↓	MAPE ↓		MOV(SOV)	
		v_1	v_2	v_1	v_2		v_1	v_2	v_1	v_2
Naive	0	76%±3.5%	76%±3.2%	50.02(1.00)	49.96(1.01)	0	71%±2.8%	72%±2.9%	50.01(0.89)	49.96(0.93)
BiCNN	**22±2.1**	**14%±2.1%**	**13%±2.3%**	86.11(10.01)	86.43(9.78)	**21.9±1.9**	**19%±2.8%**	**19%±2.7%**	83.86(13.08)	83.04(12.44)
LH	36100±1371	0%	0%	87.97(9.42)	87.80(9.30)	38800±1453	0%	0%	85.51(15.33)	85.76(15.65)

Method	GS : (40, 40), PD : U(0, 100)					GS : (40, 40), PD : N(50, 25)				
	Time (ms) ↓	MAPE ↓		MOV(SOV)		Time (ms) ↓	MAPE ↓		MOV(SOV)	
		v_1	v_2	v_1	v_2		v_1	v_2	v_1	v_2
Naive	0	78%±3.2%	79%±3.0%	49.96(0.77)	49.97(0.75)	0	75%±2.9%	75%±3.1%	49.96(0.67)	50.01(0.67)
BiCNN	**39.1±4.6**	**14% ±2.3%**	**15%±2.5%**	86.57(10.40)	86.69(11.67)	**39.1±4.3**	**19%±2.6%**	**18%±2.8%**	85.97(13.20)	84.52(12.29)
LH	80000±2314	0%	0%	88.96(8.65)	89.39(8.59)	78000±1899	0%	0%	87.51(14.82)	87.41(14.65)

as shown in Table 3.2. The test bimatrix games are generated using the same generation rule applied during training, with $(m, n) \in \mathbb{G} = \{(20, 20), (30, 30), (40, 40)\}$ and $P \in \mathbb{P} = \{U(0, 100), N(50, 25)\}$. While $\mathbb{G}$ and $\mathbb{P}$ participated in the training process, the game instances for testing are untrained data.

Naive Method. In this context, the naive method generates two random mixed strategies as a predicted Nash equilibrium. For example, we first generate two vectors by the uniform distribution $U(0, 1)$, which are then transformed into two probability distributions by the softmax function. These two probability distributions constitute a mixed strategy profile that serves as a predicted Nash equilibrium. Note that the naive method is equivalent to an untrained BiCNN model. The computational time of the naive method is considered negligible, since the generation of two vectors requires minimal machine time.

Evaluation Metric. The Mean Absolute Percentage Error (MAPE) is used to evaluate prediction accuracy. The MAPE is defined as :

$$\text{MAPE} = \frac{1}{BS} \sum_{i=1}^{BS} \left| \frac{v_i^* - \hat{v}_i}{\hat{v}_i} \right| * 100\%, \tag{3.16}$$

where BS is the number of instances in a test batch. $\hat{v}_i, i = 1, 2$ and $v_i^*, i = 1, 2$ represent the predicted and true optimal values for both players, respectively. Since the LH algorithm generates an exact Nash equilibrium, its MAPE is considered to be zero.

The Mean Optimal Value (MOV) and Standard Deviation of Optimal Values (SOV) are employed to provide a simple description of a test batch, denoted as :

$$\begin{aligned} \text{MOV} &= \tfrac{1}{BS} \textstyle\sum_{i=1}^{BS} v_i, \\ \text{SOV} &= \sqrt{\tfrac{1}{BS} \textstyle\sum_{i=1}^{BS} (\text{MOV} - v_i)^2}, \end{aligned} \tag{3.17}$$

where v_i refers to either $\hat{v}_i$ or v_i^*.

Comparative Results. The experimental results presented in Table 3.2 provide the following insights :

(a) In all scenarios, the errors of the BiCNN model are significantly lower than those of the naive method. This confirms the effectiveness of our training in Section 3.5.1, since the naive method is equivalent to the untrained BiCNN model. The BiCNN model after training can be considered as an improved version of the naive method, providing a significant improvement in accuracy with only a small increase in computational time.

(b) Although the LH algorithm can produce an exact Nash equilibrium, it requires more computation time. As discussed in Section 3.5.2, our approach drastically reduces the computational cost of the LH algorithm. Furthermore, we note that the BiCNN model predicts GS (40, 40) in 4 times more time than GS (20, 20), while the time of the LH algorithm increases by about 5.5 times. The MOV and SOV results computed by BiCNN are already close to

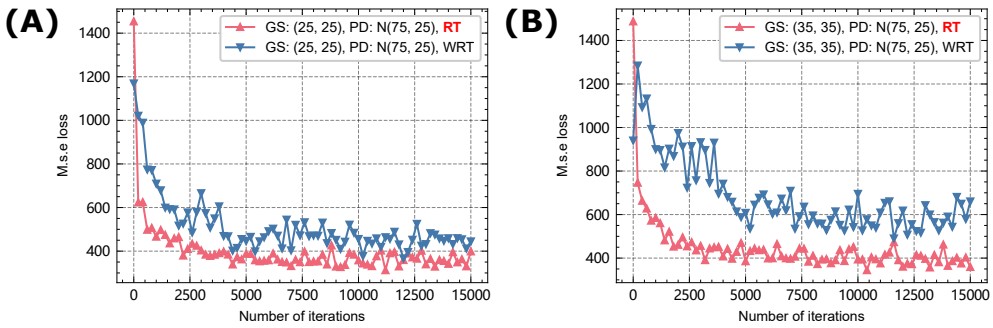


Figure 3.5 – M.S.E. loss values of untrained GS-PD combinations. RT and WRT refer to robust training and non-robust training, respectively. (A) Loss values for GS : $(25, 25)$ and PD : $N(75, 25)$. (B) Loss values for GS : $(35, 35)$ and PD : $N(75, 25)$.

the results obtained by the LH algorithm, indicating a higher efficiency in predicting outcomes.

(c) The BiCNN method strikes a balance between the naive method and the LH algorithm. While the naive method requires minimal computation time, it lacks prediction accuracy. Conversely, BiCNN significantly improves the accuracy of the naive method while significantly reducing the computational time required by the LH algorithm, making it an ideal compromise between the two approaches.

3.5.4 . Ablation Study

Motivation for the Ablation Study. In previous studies [180], the model therein is trained with a single GS-PD combination, resulting in suboptimal performance on untrained GS-PDs. Our proposed Alg. 1 incorporates multiple GS-PDs for training to improve the robustness of the model over a wide range of untrained GS-PDs. We refer to training with only one GS-PD as without robust training (WRT) and training with multiple GS-PDs as robust training (RT). This ablation study aims to investigate the importance of RT in Alg. 1.

RT and WRT Setup. We initialize two BiCNN models and train them with RT and WRT, respectively. The other training settings, including hyperparameters and model architecture, remain the same for both models. The first model (BiCNN), trained with RT, is the same as the one obtained in Section 3.5.1. The second model (BiCNN WRT) is trained only with the GS $(20, 20)$ and PD $U(0, 100)$. All test bimatrix games in this subsection are generated from untrained GS-PD combinations. Fig. 3.5 shows the loss reduction for RT and WRT on untrained GS-PD combinations, specifically $(25, 25) - N(75, 25)$ and $(35, 35) - N(75, 25)$. Table 3.3 shows additional results for the two models after training.

RT and WRT Result Comparison. As shown in Fig. 3.5, RT converges

Table 3.3 – Performance of BiCNN, naive method, and LH algorithm on untrained GS-PD combinations. Each test batch contains 1,000 instances.

Method	GS : (25, 25), PD : U(10, 90)					GS : (35, 35), PD : U(10, 90)				
	Time (ms) ↓	MAPE ↓		MOV(SOV)		Time (ms) ↓	MAPE ↓		MOV(SOV)	
		v_1	v_2	v_1	v_2		v_1	v_2	v_1	v_2
Naive	0	59%±3.0%	60%3.2%	50.02(1.01)	49.99(0.98)	0	62%±3.2%	62%±3.3%	49.97(0.70)	50.01(0.69)
LH	24100±824	0%	0%	79.61(7.81)	79.80(7.80)	48900±1565	0%	0%	81.00(7.11)	81.14(7.04)
BiCNN WRT	16.1±1.2	12%±2.5%	12%±2.2%	76.36(6.48)	76.44(6.35)	34.5±3.3	17%±2.7%	17%±2.8%	71.84(7.49)	71.36(7.47)
BiCNN	**16.1±1.3**	**12%±2.2%**	**12%±2.3%**	78.17(7.96)	78.09(8.09)	**34.3±3.6**	**12%±2.1%**	**12%±2.0%**	78.74(8.56)	78.92(8.71)

Method	GS : (25, 25), PD : N(75, 25)					GS : (35, 35), PD : N(75, 25)				
	Time (ms) ↓	MAPE ↓		MOV(SOV)		Time (ms) ↓	MAPE ↓		MOV(SOV)	
		v_1	v_2	v_1	v_2		v_1	v_2	v_1	v_2
Naive	0	45%±2.9%	46%±3.1%	75.02(1.07)	75.03(1.08)	0	49%±3.1%	50%±2.8%	75.03(0.78)	74.97(0.79)
LH	26000±855	0%	0%	108.76(14.72)	109.65(15.68)	59300±1487	0%	0%	111.88(14.84)	112.25(14.72)
BiCNN WRT	16.1±1.2	15%±2.8%	16%±2.9%	106.26(11.39)	105.74(10.54)	34.5±3.4	18%±3.0%	19%±2.9%	100.40(12.58)	100.16(13.07)
BiCNN	**16.1±1.5**	**14%±2.4%**	**14%±2.6%**	107.28(14.31)	107.07(14.47)	**34.5±3.8**	**15%±2.6%**	**14%±2.6%**	109.62(13.31)	108.43(12.76)

Method	GS : (25, 25), PD : Poi(50, 10)					GS : (35, 35), PD : Poi(50, 10)				
	Time (ms) ↓	MAPE ↓		MOV(SOV)		Time (ms) ↓	MAPE ↓		MOV(SOV)	
		v_1	v_2	v_1	v_2		v_1	v_2	v_1	v_2
Naive	0	41%±2.9%	40%±2.8%	50.01(0.63)	49.96(0.60)	0	44%±3.1%	45%±3.2%	50.01(0.43)	49.98(0.43)
LH	21500±644	0%	0%	70.32(12.07)	69.75(12.11)	48800±1389	0%	0%	72.18(12.65)	72.37(12.92)
BiCNN WRT	16.1±1.4	21%±2.8%	21%±3.1%	67.51(9.98)	66.08(8.77)	34.4±3.5	22%±3.1%	23%±3.0%	64.49(10.38)	65.29(11.16)
BiCNN	**16.1±1.1**	**16%±1.9%**	**16% ±2.3%**	70.45(13.50)	70.26(13.51)	**34.3±3.1**	**17%±2.2%**	**18%±2.4%**	69.31(10.28)	67.92(8.78)

faster and achieves a lower loss compared to WRT. The results in Table 3.3 reveal the following insights :

(a) The computation times for BiCNN and BiCNN WRT are identical, since they share the same model structure. The different training methods do not affect this aspect.

(b) In all cases, BiCNN outperforms BiCNN WRT, as shown by the lower MAPE and the MOV that is closer to the true MOV.

(c) The accuracy gap between the two models on GS $(35, 35)$ is more significant because BiCNN includes training GSs of $(30, 30)$ and $(40, 40)$, ensuring accurate performance on $(35, 35)$. In contrast, BiCNN WRT only has a training GS of $(20, 20)$, which is far from $(35, 35)$.

3.5.5 . Enhancing Traditional Solvers with BiCNN

TS and DFM Solvers. In this subsection, we explore the combination of BiCNN with two traditional solvers :

— *TS algorithm [184]* is a polynomial-time approximation algorithm for solving bimatrix games. It computes ϵ-approximate Nash equilibrium, with $\epsilon = 0.3393$.

— *DFM algorithm[185]* is a new refinement of the TS algorithm, and yields a 1/3 approximate Nash equilibrium, which is slightly better than the TS algorithm.

The difference between TS and DFM with the LH algorithm and enumeration methods is that TS and DFM are polynomial-time approximation algorithms. While the LH algorithm and enumeration methods are exact methods that cannot solve problems in polynomial time.

BICNN+TS/DFM. We use the BiCNN prediction as an initial strategy to warm start both TS and DFM. The BiCNN model first provides a fast prediction, which is then refined by the traditional solvers, TS and DFM, to improve accuracy. By combining with these traditional solvers, our goal is to provide a theoretical guarantee for the BiCNN prediction. However, we did not include BiCNN to speed up the LH algorithm and the enumeration methods. The reason for this is that both the LH algorithm and the enumeration methods work independently of the initial strategy. For example, the LH algorithm is dependent on the initial drop label, and the enumeration methods have similar dependencies, making them impossible to take advantage of the BiCNN prediction.

Accelerating TS/DFM with BiCNN. Fig. 3.6-(A) clearly shows that using the BiCNN model as a warm start for traditional solvers leads to a dramatic increase in computational efficiency. These solvers require nearly 100 iterations to complete the computation when using a random strategy as the initial strategy. In contrast, using the BiCNN prediction as the initial strategy, they complete the computation in just a few iterations, speeding up the process by more than 20 times. We also

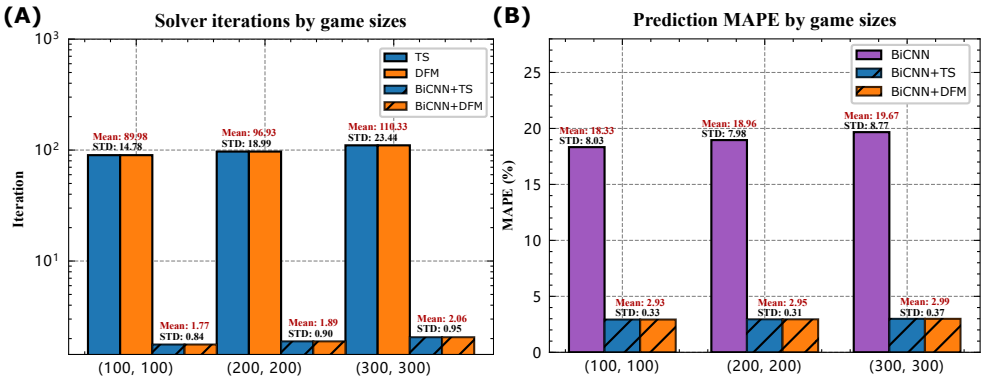


Figure 3.6 – (A) : Average iterations of standalone TS/DFM and BiCNN+TS/DFM. (B) : Prediction accuracy of standalone BiCNN and BiCNN+TS/DFM. The game sizes under consideration are $(100, 100)$, $(200, 200)$, $(300, 300)$ with the probability distribution of $U(0, 100)$. Each test batch contains 100 game instances.

observed that the results of TS and DFM are very close, since they differ only in the final output step, and thus the number of iterations is almost the same. We have not shown the computation time in the figure because the computation of both solvers depends on the linear programming solver used, which could introduce significant variance. Therefore, to maintain fairness in our comparison, we have focused only on the iteration counts.

Improving BiCNN Prediction with TS/DFM. As shown in Fig. 3.6-(B), traditional solvers significantly improve the prediction accuracy for the BiCNN model. Originally, the BiCNN prediction has a relatively high MAPE, especially since the considered game sizes are large and untrained. With the help of the TS and DFM solvers, the MAPE of the BiCNN model drops to nearly 3%, which means that the predicted strategy after the improvement is much closer to the true Nash equilibrium. This significant improvement in accuracy is due to the fact that the traditional solvers have a strict error bound. Therefore, this combination not only improves the computational efficiency of traditional solvers, but also improves the prediction accuracy of BiCNN, illustrating the potential of our hybrid approach in tackling complex games.

3.5.6 . Discussion

Why BiCNN is Much Faster. As discussed in Section 3.5.2, BiCNN significantly outperforms traditional solvers in terms of solution speed. This superior performance is due to the non-iterative solution procedure in BiCNN, as opposed to the iterative procedure used by traditional solvers. Essentially, the BiCNN model is a function that directly maps bimatrix games to its predicted Nash equilibrium, while traditional solvers require multiple iteration steps to determine the Nash

equilibrium. For example, the LH algorithm requires continuous label replacement until a Nash equilibrium is reached, and the enumeration methods require solving a system of inequalities at each solution step. In contrast to these traditional methods, BiCNN requires only one forward pass of the neural network to obtain the predicted Nash equilibrium, which requires very little computational time.

Computational Complexity. Consider a BiCNN model with fixed structure. The computational time complexity is linear with respect to the size of the game to be solved, denoted as $O(d)$, where d is the size of the bimatrix game. This is because as the size of the game increases, the convolution operations within the neuralnetwork scale linearly. Even with multiple convolution kernels or deeper networks, the basic linear growth remains, affected only by a constant multiplier. In comparison, the LH algorithm, recognized as the best exact method, has an exponential growth even in the best case [30], underscoring the computational efficiency of our BiCNN approach.

BiCNN Limitations. Despite the promising results, there is still room for improvement in the BiCNN model, especially in terms of prediction accuracy. While traditional solution methods can provide an exact Nash equilibrium, our approach can only provide an approximation. Compared to previous research [180, 179] that solves matrix games (a special case of bimatrix games), the MAPE error in this study is higher. The increase in error is primarily due to the inherent complexity of bimatrix games compared to matrix games. Specifically, matrix games can be transformed into linear programming problems, while bimatrix games lead to relatively more complex linear complementarity problems. In terms of robustness, the performance of the model is highly dependent on the selected training GS-PDs. Although the robust training of Alg. 1 improves the accuracy on some untrained GS-PDs, as shown in Table 3.3, it is not guaranteed for a wider range of untrained GS-PDs.

Future Directions for Improving BiCNN Accuracy : To improve the accuracy and robustness of the BiCNN model, we can explore several future directions :

— *Running more iterations in Alg. 1 :* Our proposed algorithm ensures that the BiCNN model avoids training twice on the same bimatrix game and overfitting on any specific dataset. Thus, providing more training iterations could potentially refine the accuracy of the model without any negative impact.

— *Designing efficient training algorithms :* Although Alg. 1 prevents overfitting, it introduces a higher computational overhead than traditional deep learning training because it requires the generation of a new batch of bimatrix games at each iteration. This data generation process is time-consuming, which can be improved by recycling the generated bimatrix games for training, or by using approximation algorithms such as TS or DFM to speed up the generation of bimatrix games.

— *Fine-tuning the model architecture and training details :* To facilitate un-

derstanding of the proposed approach, the model architecture and training details in this chapter are intentionally kept simple. There are many opportunities for further improvement, including adjusting hyperparameters, incorporating advanced model architectures, and implementing state-of-the-art training techniques.

— *Expanding training GS-PD combinations :* In Section 3.5.1, we use only six combinations of training GS-PDs. We could introduce a greater variety of combinations to improve the BiCNN model's ability to solve more untrained GS-PDs, although this may also increase the training load.

3.6 . Conclusion

In this chapter, we introduced BiCNN, a CNN-based approach for predicting Nash equilibria in bimatrix games. We provided a comprehensive description of the methodology, including the design of the BiCNN model, the loss function, and the robust training algorithm with multiple GS-PDs. In the numerical experiments section, we compared BiCNN with traditional solution methods in terms of computational efficiency and accuracy, and discussed both the advantages and limitations of our proposed method.

By connecting the bimatrix game problem to the expanding fields of machine learning and deep learning, this work contributes to the ongoing exploration of Nash equilibria from a machine learning perspective. Given the rapid growth of these research areas, we believe that this proposed approach will continue to be relevant and valuable to the advancement of computational game theory and its applications.

4 - Solving Nonlinear Projection Equations with Neurodynamic Optimization and PINNs

In this chapter, we consider Nonlinear projection equations (NPEs), which is a unified framework for various nonlinear optimization and engineering problems [31]. Unlike the bimatrix games in Chapter 3, NPEs cannot be directly used as input for neural networks. Therefore, we need some advanced tools, namely neurodynamic optimization and PINNs, both of which are discussed in detail in Chapter 2.

To address these challenges, our approach consists of three main steps. First, we transform the NPE into an Ordinary Differential Equation (ODE) system via neurodynamic optimization, making the ODE system our new objective to solve. Second, we use a Physics-Informed Neural Networks (PINNs)-like model to serve as an approximate solution to this ODE system. Finally, we train the neural network using a specialized algorithm that is designed to optimize both the ODE system and the original NPE problem. By following this sequence of steps, we are able to bridge the gap between NPEs and deep learning. The effectiveness of our proposed framework is validated on a variety of classical problems, including variational inequalities and complementarity problems.

This chapter corresponds to the publication [146].

4.1 . Introduction

Nonlinear optimization problems are prevalent across a wide range of fields, such as engineering, physics, and economics. Solving these problems requires finding an optimal solution that satisfies a defined set of constraints while optimizing the objective function [186]. NPEs are a useful tool for formulating these problems and serve as a unifying framework for treating various nonlinear optimization problems, including nonlinear complementarity problems, variational inequalities, and equilibrium point problems [31, 187].

The NPEs are typically addressed by neurodynamic optimization, which models the problem as a system of ODEs [32, 33, 34]. The constructed ODE system must be shown to have the global convergence property, i.e., the state solution of the system converges to the optimal solution of the NPE, regardless of the initial point. The NPE problem is then transformed into solving the state solution of the ODE system. However, the ODE system is typically highly nonlinear and has no analytical solutions. Therefore, numerical integration methods such as Runge-Kutta (RK) methods are commonly used to solve the state solution [188].

Despite the usefulness of numerical integration methods, they are not efficient enough for solving the NPE problems. This inefficiency is due to the fact that the state solution of the ODE system only provides a solution to the NPE at

the end state. To reach the end state, numerical integration methods require the computation of all intermediate states starting from the initial state, making the process computationally intensive. Therefore, there is a need for a more efficient algorithm to solve the NPE problem.

Neurodynamic Optimization. Over the past few decades, numerous neurodynamic approaches have been developed to solve various constrained optimization problems, including linear and quadratic programming [42, 189], general convex programming [46, 69, 190], biconvex optimization [86], non-smooth optimization [191], and pseudoconvex optimization problems [192]. In particular, a projection neurodynamic model for solving NPEs was introduced and found to have global convergence to the exact solution under mild conditions [32]. The model also showed both asymptotic and exponential stability without the need for a smooth nonlinear mapping. To further improve the performance, a bi-projection neurodynamic model was developed to efficiently solve quadratic optimization problems [33]. In addition, a collaborative approach combining the projection neurodynamic model with particle swarm optimization was introduced for global optimization problems [34].

Physics-Informed Neural Networks. Another avenue of research explored in this chapter is the use of deep learning to solve differential equations. In the 1990s, Lagaris et al. used neural networks (NNs) to serve as solutions to both ODEs and PDEs, embedding boundary conditions directly into the network architecture [193, 194]. The advent of deep learning has reinvigorated interest in using such methods to tackle high-dimensional, nonlinear PDEs [195, 196]. A key contribution in this area is the development of PINNs [26], which integrate differential equations and data errors into the loss function. The versatile architecture and efficient training algorithms of PINNs have enabled numerous successes in a wide range of computational challenges in physics and engineering [123, 197, 198, 199, 200]. This expanding research landscape is driven by a combination strategy that adapts PINNs to exploit the structural properties of the target problem. As a result, a variety of PINN variants have emerged to address different problem scenarios [119, 201] and improve computational efficiency [107, 202, 203, 128]. Apart from collocation-based PINN approaches, many studies use deep energy methods to solve PDEs, where the energy of the system is used as a loss function to train the NN model [204, 123, 122]. Many packages have been developed to support the use of deep learning for solving PDEs [99, 205].

Contributions. This chapter presents several key contributions :

— We propose a deep learning approach to NPEs that combines neurodynamic optimization with PINNs. Our approach reformulates an NPE problem as an NN training problem, thereby eliminating the need for numerical methods to solve NPEs.

— To improve the performance of our proposed approach, we design a specia-

lized training algorithm that focuses on the end state of the ODE system. In each training iteration, the algorithm uses an evaluation metric based on the NPE error to assess the predictive accuracy of the NN model and retain the optimal result. Therefore, the NN model is not only trained to solve the ODE system, but also to solve the NPE problem.

— In our experiments, we demonstrate the effectiveness of the proposed approach in solving large-scale NPE problems. We compare our approach with the traditional RK method and PINN to validate its performance. In addition, we perform a hyperparameter sensitivity analysis to investigate the impact of different hyperparameter configurations on the performance of our proposed approach.

The remaining sections are organized as follows : Section 4.2 provides the necessary background, including an introduction to the NPE problem, neurodynamic optimization, and the RK method. In Section 4.3, we present our NN model for the NPE problem. Section 4.4 details the design of the loss function and the training algorithm for the proposed NN model. Section 4.5 presents the experimental results, and we compare our approach with the RK method and PINN. Finally, Section 4.6 summarizes the main results of the chapter and outlines possible directions for future research.

4.2 . Preliminaries

4.2.1 . NPE Problems

Definition 4.1 (NPE) *Consider a nonlinear mapping $G : \mathbb{R}^n \to \mathbb{R}^n$ and a feasible set $\Omega \subset \mathbb{R}^n$. The projection function $P_\Omega : \mathbb{R}^n \to \Omega$ maps a point $z \in \mathbb{R}^n$ onto Ω, such that :*

$$P_\Omega(z) = \arg\min_{x \in \Omega} \|z - x\| \tag{4.1}$$

where $\|\cdot\|$ denotes the Euclidean norm.

The NPE problem, denoted by $NPE(\Omega, G)$, is to find a vector $x^* \in \Omega$ satisfying :

$$P_\Omega(x^* - G(x^*)) = x^* \tag{4.2}$$

Assumption 4.1

— *The function $G(\cdot)$ is locally Lipschitz continuous.*

— *The feasible set Ω is a box-constrained set, defined as $\Omega = \{x = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n \mid l_i \leq x_i \leq h_i\}$, where l_i and h_i denote the lower and upper bounds of x_i respectively. In this case, the projection function $P_\Omega(\cdot)$ in equation* (4.1) *is reduced as follows :*

$$P_\Omega(x) = \left(P_\Omega^1(x_1), P_\Omega^2(x_2), \ldots, P_\Omega^n(x_n)\right)^T, \tag{4.3}$$

where $P_{\Omega}^{i}(x_i)$, $i \in \{1, 2, \ldots, n\}$ *is defined as :*

$$P_{\Omega}^{i}(x_i) = \begin{cases} l_i & \text{if } x_i < l_i, \\ x_i & \text{if } l_i \leq x_i \leq h_i, \\ h_i & \text{if } x_i > h_i. \end{cases} \tag{4.4}$$

— *The Jacobian* $\nabla G(x)$ *for* $x \in \Omega$ *is positive semi-definite.*

In this chapter, we only consider the NPE problem under Assumption 4.1, namely that the feasible set is box-constrained and the nonlinear mapping is locally Lipschitz continuous. In the following, we show two nonlinear optimization problems, namely the nonlinear complementarity problem (NCP) and the variational inequality (VI), both of which can be reformulated as NPEs.

Definition 4.2 (NCP) *Consider a nonlinear mapping* $G : \mathbb{R}^n \to \mathbb{R}^n$*. The nonlinear complementarity problem, denoted by* $NCP(G)$*, is to find a vector* $x^* \in \mathbb{R}^n$ *satisfying :*

$$G(x^*) \geq 0, \quad x^* \geq 0, \quad G(x^*)^T x^* = 0. \tag{4.5}$$

Definition 4.3 (VI) *Consider a nonlinear mapping* $G : \mathbb{R}^n \to \mathbb{R}^n$ *and a feasible set* $\Omega \subset \mathbb{R}^n$*. The variational inequality problem, denoted by* $VI(\Omega, G)$*, is to find a vector* $x^* \in \Omega$ *satisfying :*

$$(x - x^*)^T G(x^*) \geq 0, \quad x \in \Omega. \tag{4.6}$$

Proposition 4.1 (Harker & Pang [31]) *Let* $\Omega \subset \mathbb{R}^n$ *be a nonempty closed convex set. Then* x^* *solves the problem* $NCP(G)$ *if and only if* x^* *solves* $NPE(\mathbb{R}^n_+, G)$*, where* $\mathbb{R}^n_+ = \{x \in \mathbb{R}^n | x \geq \mathbf{0}\}$ *represents the set of non-negative real vectors.*

Proposition 4.2 (Harker & Pang [31]) *Let* $\Omega \subset \mathbb{R}^n$ *be a nonempty closed convex set. Then* x^* *solves the problem* $VI(\Omega, G)$ *if and only if* x^* *solves* $NPE(\Omega, G)$*.*

4.2.2 . Neurodynamic Optimization

In the following, we show how to model an NPE as an ODE system. Let $y : \mathbb{R} \to \mathbb{R}^n$ be a time-dependent function, where $y(t)$ represents the state at time t. The aim of neurodynamic optimization is to design a first-order ODE system, $\frac{dy}{dt}$, to govern $y(t)$. In this chapter, we employ the projection neurodynamic model introduced by [32] to model the NPE, where the ODE system is defined as follows :

$$\frac{dy}{dt} = -G\left(P_{\Omega}(y)\right) + P_{\Omega}(y) - y. \tag{4.7}$$

To simplify the discussion, we define :

$$\Phi(y) = -G\left(P_\Omega(y)\right) + P_\Omega(y) - y. \tag{4.8}$$

Thus, the ODE system (4.7) can be written as $\frac{dy}{dt} = \Phi(y)$.

Definition 4.4 (State solution) *Consider an ODE system $\frac{dy}{dt} = \Phi(y)$, where $\Phi : \mathbb{R}^n \to \mathbb{R}^n$ and an initial condition $y(t_0) = y_0$. A vector value function $y : \mathbb{R} \to \mathbb{R}^n$ is the state solution, if it satisfies the ODE system $\frac{dy}{dt} = \Phi(y)$ and the initial condition $y(0) = y_0$.*

$y(t)$ is called the state at time t. Given a time range $[t_0, T]$, $y(T)$ is called the end state on the interval.

Theorem 4.3 (Xia & Feng [32]) *Consider an NPE problem, $NPE(\Omega, G)$, and let Assumption 4.1 hold. Given any initial condition, $y(t_0) = y_0$, the state solution of the ODE system of Eq. (4.7) converges to the optimal solution of $NPE(\Omega, G)$ as time t goes to infinity, i.e,*

$$\lim_{t\to\infty} y(t) = x^*, \tag{4.9}$$

where x^* is a satisfied point of $NPE(\Omega, G)$.

In particular, if $NPE(\Omega, G)$ contains only one satisfied point x^*, then the ODE system is globally asymptotically stable at x^*.

Initial Value Problem (IVP) Construction. In practice, in order to use the neurodynamic approach to solve the NPE, we need to construct an IVP consisting of three components : 1) the ODE system of Eq. (4.7), 2) an initial condition $y(t_0) = y_0$, and 3) a time range $t \in [t_0, T]$. $y(t)$ for $t \in [t_0, T]$ represents the state solution of this IVP over the time range $[t_0, T]$, where the end state, $y(T)$, is considered to be the predicted solution to the NPE. According to Theorem 4.3, the larger the time range $[t_0, T]$, the closer $y(T)$ is to the optimal solution x^* of the NPE.

4.2.3 . Runge-Kutta Method

The fourth-order Runge-Kutta method (often referred to as the RK method) is a commonly used method for solving ODE systems. The RK method takes as input an ODE system $\frac{dy}{dt} = \Phi(y)$, an initial condition $y(t_0) = y_0$, and a time range $[t_0, T]$. The method sets N collocation points that are equally spaced on the time range $[t_0, T]$. The RK method returns an approximate state solution, denoted by $\bar{y}(t)$, where t is a time point within the finite set $\{t_0, t_1, \ldots, t_N, t_T\}$. The RK algorithm is stated as follows :

— Step 1 : Initialize the step size $h = \dfrac{(T - t_0)}{N + 1}$, the initial time $t = t_0$, and set the initial state $\bar{y}(t_0) = y_0$.

— Step 2 : For $i = 1, 2, \ldots, N+1$ do Steps 3 and 4.

— Step 3 : Set

$$\begin{aligned} K_1 &= h\Phi\left(\bar{y}(t)\right), \\ K_2 &= h\Phi\left(\bar{y}(t) + \frac{K_1}{2}\right), \\ K_3 &= h\Phi\left(\bar{y}(t) + \frac{K_2}{2}\right), \\ K_4 &= h\Phi\left(\bar{y}(t) + K_3\right). \end{aligned} \tag{4.10}$$

— Step 4 : Set

$$\bar{y}(t+h) = \bar{y}(t) + \frac{K_1 + 2K_2 + 2K_3 + K_4}{6}, \tag{4.11}$$

and $t = t + h$.

When applying the RK method to the NPE problem, the approximate end state $\bar{y}(T)$ is considered as the predicted solution of the NPE problem. Specifically, we have :

$$\bar{y}(T) \approx y(T) \approx x^*, \tag{4.12}$$

where $\bar{y}(T) \approx y(T)$ indicates that the end state obtained by the RK method is an approximation of the true end state, and $y(T) \approx x^*$ indicates that the true end state is the predicted solution of the NPE problem according to Theorem 4.3.

4.3 . Modified PINN

Model description. We propose a neural network (NN) model to solve the NPE problem. Our model can be expressed by the following equation :

$$\hat{y}(t; w) = y_0 + \left(1 - e^{-(t-t_0)}\right) \mathrm{N}(t; w), \quad t \in [t_0, T], \tag{4.13}$$

where $\mathrm{N}(t; w)$ represents a fully connected NN with trainable parameters w, and $[t_0, T]$ is a given time range. We use the Lagaris method to incorporate the initial conditions of the ODE system into the NN model [193]. In particular, the auxiliary function $\left(1 - e^{-(t-t_0)}\right)$ ensures that the NN model always satisfies the initial condition (t_0, y_0), i.e., $\hat{y}(t = t_0; w) = y_0$, regardless of the model parameters w. The exponential form in the auxiliary function has been demonstrated to improve the convergence of the model [206].

Approximate state solution of the ODE. As shown in Fig. 4.1 (Left), the proposed model itself is an approximate state solution to the ODE system of Eq. (4.7) on the time range $[t_0, T]$, i.e.,

$$\hat{y}(t; w) \approx y(t), \quad t \in [t_0, T], \tag{4.14}$$

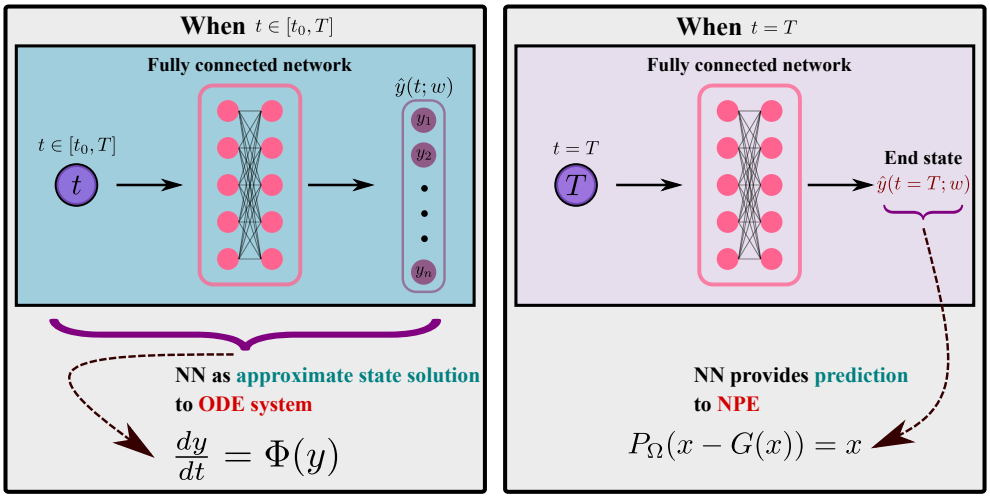


Figure 4.1 – Neural network (NN) solution for the ODE system and the NPE. Left : When $t \in [t_0, T]$, the model $\hat{y}(t; w)$ itself is considered to be an approximate state solution of the ODE system. Right : When $t = T$, the end state of the model, $\hat{y}(t = T; w)$, is a predicted solution to the NPE.

where $y(t)$ represents the true state solution of the ODE system. Although the input time t of the model $\hat{y}(t; w)$ can be any real number, we only consider $\hat{y}(t; w)$ as the solution of the ODE for the time range $[t_0, T]$. Therefore, we restrict the input to $t \in [t_0, T]$.

Predicted solution of the NPE. The end state of the proposed model, i.e., $\hat{y}(t = T; w)$, is used as a predicted solution to the NPE of Eq. (4.2), as shown in Fig. 4.1 (Right). The following equation shows how $\hat{y}(t = T; w)$ approximate the optimal solution x^* of the NPE problem :

$$\hat{y}(t = T; w) \approx y(T) \approx x^*, \tag{4.15}$$

where $\hat{y}(t = T; w) \approx y(T)$ indicates that the end state of our model approximates the true end state, and $y(T) \approx y^*$ comes from Theorem 4.3, indicating that $y(T)$ solves the NPE.

Unlike most PINN models, which aim to solve for the entire input space $[t_0, T]$, our NN model focuses on the end state of the ODE system, since it represents the predicted solution to the NPE problem. In the following section, we will show how to train the NN model with an emphasis on improving the prediction accuracy of the end state.

4.4 . Model training

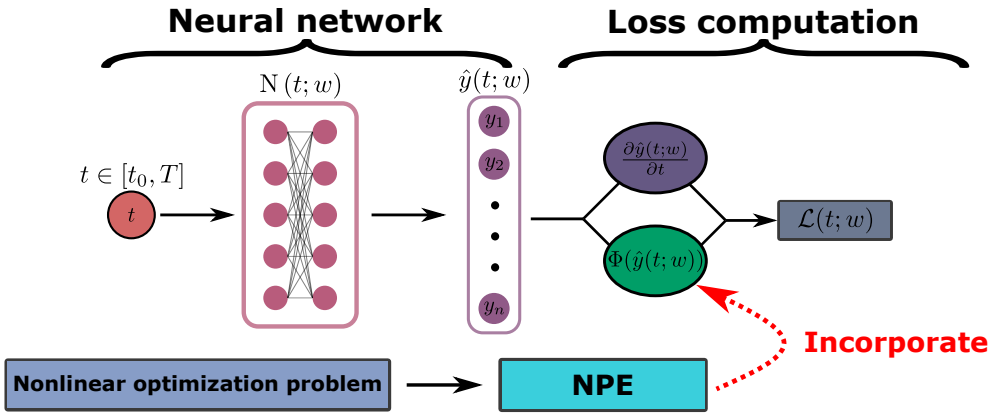


Figure 4.2 – Integrating an NPE into loss computation for NN training.

Section 4.4.1 provides a definition of the loss function and objective function for the proposed NN model. Section 4.4.2 presents a training algorithm for the NN model. Section 4.4.3 compares our proposed NN approach with the RK method.

4.4.1 . Training Objective

Loss Function. The loss function of the proposed NN model is defined as :

$$\mathcal{L}(t, w) = \left\| \frac{\partial \hat{y}(t; w)}{\partial t} - \Phi(\hat{y}(t; w)) \right\|, \tag{4.16}$$

where $\Phi(\cdot)$ refers to the ODE system, corresponding to the NPE problem to be solved. $\Phi(\hat{y}(t; w))$ represents the expected derivative according to the ODE system, and $\frac{\partial \hat{y}(t;w)}{\partial t}$ represents the actual derivative of the NN model. $\frac{\partial \hat{y}(t;w)}{\partial t}$ can be computed using automatic differentiation tools such as PyTorch or JAX [181, 207]. $\mathcal{L}(t, w)$ represents the difference between the two at time t and with network parameters w. As shown in Fig. 4.2, the NPE is first reformulated as an ODE system via neurodynamic optimization. The ODE system is then incorporated into the loss computational process.

Objective Function. The objective function of the NN model is defined as :

$$J(w) = \int_{t_0}^{T} \mathcal{L}(t, w) dt, \tag{4.17}$$

which is the integral of the loss function over the time range $[t_0, T]$. The loss value $\mathcal{L}(t, w)$ represents the error of the model at time t, while the objective function $J(w)$ represents the total error of the model over the time range $[t_0, T]$.

Batch Loss. However, the objective function $J(w)$ is computationally intractable due to its integral part. Therefore, in practice, we cannot directly use $J(w)$

to train the NN model. Instead, we train the model by minimizing the following batch loss :

$$\mathcal{L}(\mathbb{T}, w) = \frac{1}{|\mathbb{T}|} \sum_{t \in \mathbb{T}} \mathcal{L}(t, w), \tag{4.18}$$

where $\mathbb{T}$ is a set of time points uniformly sampled from the interval $[t_0, T]$, and $|\mathbb{T}|$ denotes the size of the set. The batch loss $\mathcal{L}(\mathbb{T}, w)$ approximates the objective function $J(w)$ by a sum of loss values over a set of sampled time points. By minimizing the batch loss, we can effectively train the model to solve the NPE.

4.4.2 . Algorithm Design

NPE Error. We introduce an evaluation metric, called NPE error, to measure how well a prediction x_{pred} solves the NPE problem. The metric is defined as :

$$\text{NE}(x_{\text{pred}}) = \left\| P_\Omega(x_{\text{pred}} - G(x_{\text{pred}})) - x_{\text{pred}} \right\|_\infty, \tag{4.19}$$

where $\|\cdot\|_\infty$ represents the infinity norm.

Algorithm 2 Deep learning solver for NPE based on neurodynamic optimization

Input : $NPE(\Omega, G)$ as defined in Eq. (4.2); Time range $[t_0, T]$; Initial condition (t_0, y_0).

Output : x_{best}, the NN prediction to $NPE(\Omega, G)$.

function solver($NPE(\Omega, G)$, $[t_0, T]$, y_0)
  Derive the ODE system, $\Phi(\cdot)$, according to Eq. (4.7).
  Initialize a NN model $\hat{y}(t; w)$.
  $\text{NE}_{\text{best}} = \text{NE}(\hat{y}(t = T; w))$
  while iteration $\leq$ maximum iteration **do**
    $\mathbb{T} \sim U(t_0, T)$ ▷ Sample collocation points $\mathbb{T}$
    $\mathcal{L}(\mathbb{T}, w)$ ▷ Forward propagation
    $w \leftarrow \nabla_w \mathcal{L}(\mathbb{T}, w)$ ▷ Backward propagation
    $x_{\text{curr}} = \hat{y}(t = T; w)$
    $x_{\text{curr}} = P_\Omega(x_{\text{curr}})$ ▷ Project x_{curr} onto the feasible set Ω.
    $\text{NE}_{\text{curr}} = \text{NE}(x_{\text{curr}})$
    if $\text{NE}_{\text{curr}} < \text{NE}_{\text{best}}$ **then**
      $\text{NE}_{\text{best}} = \text{NE}_{\text{curr}}$
      $x_{\text{best}} = x_{\text{curr}}$
    end if
  end while
  return x_{best}
end function

Pipeline. Alg. 2 summarizes how to use our proposed approach to solve an NPE problem. First, we need to specify an initial condition (t_0, y_0) and a time range $[t_0, T]$ to construct an IVP for the NPE problem. Then, we initialize the

proposed NN model (4.13), which serves as an approximate state solution for this IVP. The model is trained by performing gradient descent on the batch loss of Eq. (4.18) to improve the approximation. Note that our solver is completely based on the deep learning infrastructure and does not require any standard optimization solver or numerical integration solver.

Optimal Result Retention (ORR) Mechanism. A key to Alg. 2 is that we use an ORR mechanism based on the evaluation metric of Eq. (4.19). Specifically, in each iteration, the algorithm compares the NPE error of the current iteration, denoted as $\mathrm{NE}_{\mathrm{curr}}$, with the lowest NPE error found so far, denoted as $\mathrm{NE}_{\mathrm{best}}$. Correspondingly, x_{curr} and x_{best} represent the current prediction and the best predictions found so far, respectively. If $\mathrm{NE}_{\mathrm{curr}}$ is less than $\mathrm{NE}_{\mathrm{best}}$, it means that the model found a better prediction in the current iteration. The algorithm then updates $\mathrm{NE}_{\mathrm{best}}$ to equal $\mathrm{NE}_{\mathrm{curr}}$ and stores the best prediction as $x_{\mathrm{best}} = x_{\mathrm{curr}}$. This mechanism ensures that the best prediction obtained by the model is maintained throughout the training process, improving the overall performance of the algorithm.

4.4.3 . Comparison with the RK Method

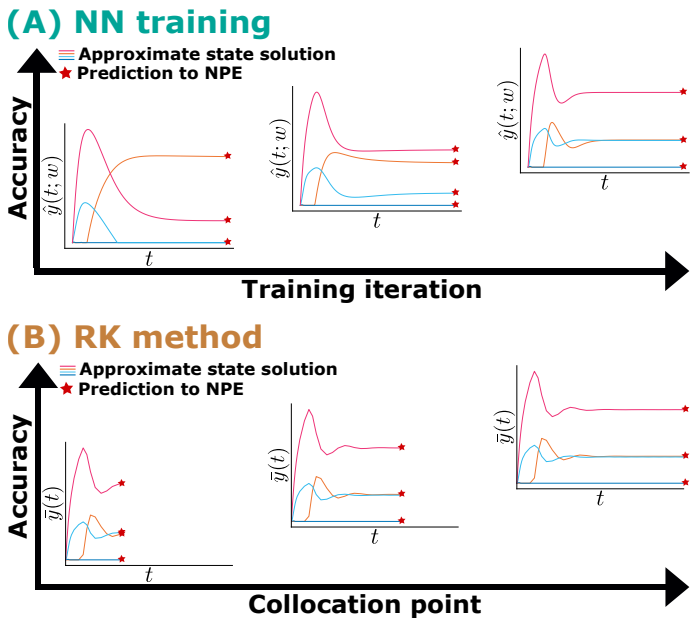


Figure 4.3 – Comparison of the solution procedures between the NN approach and RK method.

Fig. 4.3 compares the solution procedures between our proposed NN approach and the RK method, both of which solve the NPE problem by solving the IVP. Both approaches use the end state as the predicted solution for the NPE problem. However, they differ in how they enhance the accuracy of the end state.

NN Solution Procedure. The NN approach employs gradient descent on the batch loss of Eq. (4.18) to improve the NN prediction at each iteration. The evolution of the NN model is represented by $\hat{y}(t; w_1), \hat{y}(t; w_2), \ldots, \hat{y}(t; w_M)$, where w_i and $\hat{y}(t; w_i)$ denote the model parameters and the approximate state solution at the i-th iteration, respectively. The predicted end states are $\hat{y}(t = T; w_1), \hat{y}(t = T; w_2), \ldots, \hat{y}(t = T; w_M)$, where $\hat{y}(t = T; w_i)$ represents the NPE prediction at the i-th iteration.

RK solution procedure. In contrast, the RK method computes discrete collocation points iteratively. The method progresses by solving Eq. (4.10) and Eq. (4.11) to obtain $\bar{y}_j$ and t_j, which represent the solved state values and collocation point at the j-th iteration, respectively. The state value $\bar{y}_j$ incorporates all previously solved state values. At the end of the j-th iteration, $\bar{y}_j$ is used as the prediction for the NPE.

4.5 . Numerical Results

Section 4.5.1 delineates the application of the proposed NN approach for solving various types of NPE problems. Section 4.5.2 contrasts our approach with the PINN. Section 4.5.3 investigates the performance of our NN approach over different network architectures and hyperparameter configurations. Section 4.5.4 demonstrates the effectiveness of our NN approach in solving large-scale NPE problems. Finally, Section 4.5.5 discusses the distinctive features and limitations of our proposed NN approach, while also outlining possible avenues for future research.

4.5.1 . Three Examples

Experimental Setup of Our NN Approach. We used PyTorch 1.12.1 [181] to implement the proposed NN model and JAX 0.4.1 [207] to implement the ODE system. The NN model consists of a single fully connected layer with 100 neurons, and the activation function is Tanh. For training, we used the Adam optimizer with a learning rate of 0.001, a batch size of 128.

Experimental setup of the RK method. We used the RK method [208] for comparison and called it via the Scipy library [209]. We set the number of collocation points to 50,000, evenly distributed over the time range.

Linear Complementarity Problem

Example 1 : Consider the following linear complementarity problem :

$$x^T(Mx + q) = 0, \quad x \geq \mathbf{0}, \quad Mx + q \geq \mathbf{0}, \tag{4.20}$$

where

$$M = \begin{bmatrix} 50 & -8 & -6 & -9 & 12 \\ -8 & 33 & -1 & -25 & 3 \\ -6 & -1 & 38 & 10 & -4 \\ -9 & -25 & 10 & 55 & -24 \\ 12 & 3 & -4 & -24 & 20 \end{bmatrix}, \quad q = \begin{bmatrix} -2 \\ -20 \\ -16 \\ -12 \\ -14 \end{bmatrix}. \tag{4.21}$$

The goal is to find an optimal solution $x^* \in \mathbb{R}^5$ that solves Eq. (4.20). Example 1 is reformulated as $NPE(\mathbb{R}^+, G)$ by Proposition 4.1, where $G(x) = Mx + q$. Then, $NPE(\mathbb{R}^+, G)$ is modeled by the ODE system of Eq. (4.7). We establish an IVP by specifying the initial condition as $y(0) = \mathbf{0}$ and the time range as $[0, 10]$. We use the NN model proposed in Eq. (4.13), denoted as $\hat{y}(t; w)$, to serve as an approximate state solution for this IVP. The end state, $\hat{y}(t = 10; w)$, serves as the predicted solution for both $NPE(\mathbb{R}^+, G)$ and Example 1. The NN model is trained using Alg. 2 to improve accuracy.

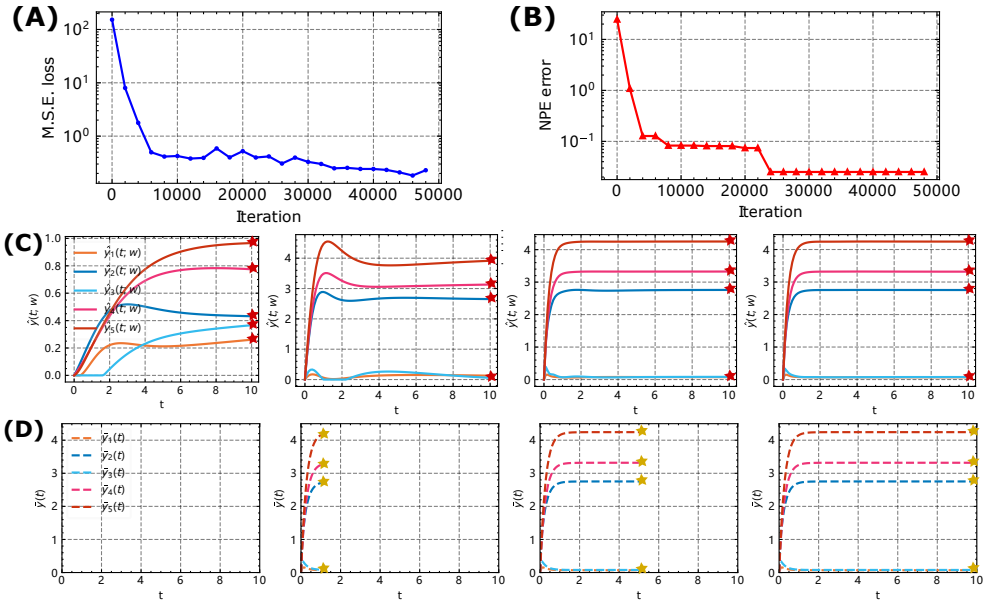


Figure 4.4 – Solving Example 1 : (A) Mean square error (MSE) loss versus training iterations. (B) NPE error versus training iterations, where the NPE error is defined in Eq. (4.2). (C) Evolution of the NN solution, $\hat{y}(t; w)$. (D) Evolution of the RK solution, $\bar{y}(t)$.

Fig. 4.4(A) and Fig. 4.4(B) show the decreasing loss and NPE error, respectively, where the loss drops from 152.59 to 0.23 and the NPE error drops from 24.69 to 0.03. Fig. 4.4(C) shows the NN model $\hat{y}(t; w)$ at the 0th, 1,000th, 10,000th, and 50,000th training iterations, from left to right, and the predicted solutions for Example 1 are marked with red stars. Fig. 4.4(D) shows the results of the RK

Table 4.1 – Comparison of predicted solutions for Example 1 between our approach and the RK method

Our NN approach			The RK method		
Iteration	Prediction	NPE error	Collocation point	Prediction	NPE error
0	[0.26, 0.43, 0.37, 0.77, 0.97]	**24.67**	0	[0.00, 0.00, 0.00, 0.00, 0.00]	20.00
100	[0.17, 3.07, 0.00, 3.64, 4.50]	**2.32**	100	[0.05, 0.34, 0.22, 0.24, 0.28]	14.60
1000	[0.13, 2.65, 0.06, 3.13, 3.92]	**1.62**	1000	[0.15, 1.70, 0.26, 1.82, 2.20]	7.86
5000	[0.08, 2.76, 0.13, 3.32, 4.25]	**0.13**	5000	[0.07, 2.71, 0.09, 3.25, 4.15]	0.37
10000	[0.07, 2.76, 0.08, 3.33, 4.25]	**0.08**	10000	[0.07, 2.75, 0.08, 3.32, 4.24]	0.03
30000	[0.07, 2.75, 0.08, 3.32, 4.24]	**0.03**	30000	[0.07, 2.75, 0.08, 3.32, 4.24]	0.03
50000	[0.07, 2.75, 0.08, 3.32, 4.24]	**0.03**	50000	[0.07, 2.75, 0.08, 3.32, 4.24]	0.03

method after accumulating 0, 5,000, 25,000, and 50,000 collocation points, and the predicted solutions are marked with yellow stars.

Table 4.1 shows the predictions of our NN approach and the RK method for Example 1 at different iterations. The results suggest that our approach is comparably accurate to the RK method given the same initial condition and time range. After 10,000 iterations, the NPE error of our prediction is reduced to less than 0.1. The final solution from our NN approach is $[0.07, 2.75, 0.08, 3.32, 4.24]$, with an NPE error of 0.03.

Nonlinear Complementarity Problem

Example 2 : Consider the following nonlinear complementarity problem :

$$x^T F(x) = 0, \quad x \geq \mathbf{0}, \quad F(x) \geq \mathbf{0}, \tag{4.22}$$

where

$$F(x) = \begin{pmatrix} x_1 e^{(x_1^2+(x_2-1)^2)} + x_2^2 + x_3 - 10 \\ (x_2-1)e^{x_1^2+(x_2-1)^2} + 4x_1 + x_2 x_3 + 2x_3^2 + e^{x_4-2} \\ x_1 + 8x_2 + 3x_3 - 3 \\ x_4 - 4 \end{pmatrix}. \tag{4.23}$$

We reformulate Example 2 as $NPE(\mathbb{R}^+, F)$ by Proposition 4.1. Then, $NPE(\mathbb{R}^+, F)$ is modeled by the ODE system of Eq. (4.7). We establish an IVP by specifying the initial point as $y(0) = \mathbf{0}$ and the time range as $[0, 10]$. We use the NN model, $\hat{y}(t; w)$, to serve as an approximate state solution for this IVP, where the end state $\hat{y}(t = 10; w)$ is the predicted solution for Example 2.

Fig. 4.5(A) and Fig. 4.5(B) illustrate the decrease in loss and NPE error, respectively. Specifically, the loss decreases from 17.78 to 0.05, while the NPE error decreases from 5.04 to a value close to zero. Notably, the most substantial reduction in both loss and NPE error occurs within the first 10,000 iterations. In addition,

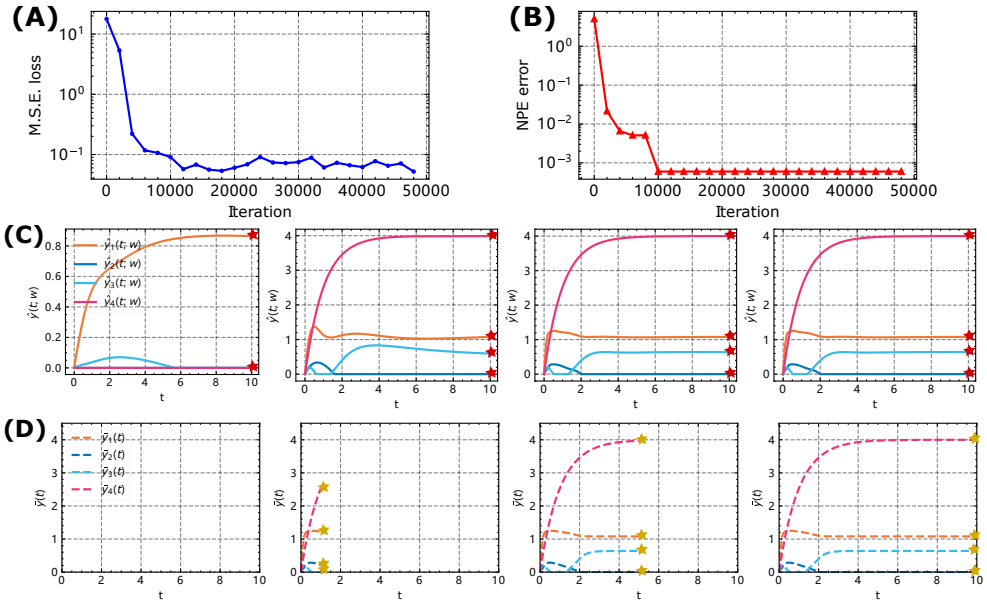


Figure 4.5 – Solving Example 2 : (A) MSE loss versus training iterations. (B) NPE error versus training iterations. (C) Evolution of the NN solution, $\hat{y}(t; w)$. (D) Evolution of the RK solution, $\bar{y}(t)$.

Table 4.2 – Comparison of predicted solutions for Example 2 between our approach and the RK method

Our NN approach			The RK method		
Iteration	Prediction	NPE error	Collocation point	Prediction	NPE error
0	[0.86, 0.00, 0.00, 0.00]	**5.04**	0	[0.00, 0.00, 0.00, 0.00]	10.00
100	[1.30, 0.94, 0.00, 2.01]	**2.00**	100	[0.19, 0.04, 0.05, 0.08]	9.44
1000	[1.08, 0.00, 0.59, 3.99]	**0.14**	1000	[1.16, 0.19, 0.20, 0.73]	3.27
5000	[1.08, 0.00, 0.64, 3.99]	**0.01**	5000	[1.22, 0.22, 0.00, 2.53]	1.47
10000	[1.08, 0.00, 0.64, 4.00]	**0.00**	10000	[1.10, 0.01, 0.40, 3.46]	0.62
30000	[1.08, 0.00, 0.64, 4.00]	**0.00**	30000	[1.08, 0.00, 0.64, 3.99]	0.08
50000	[1.08, 0.00, 0.64, 4.00]	**0.00**	50000	[1.08, 0.00, 0.64, 4.00]	0.09

the implementation of the ORR mechanism, as described in Alg. 2, ensures that the NPE error decreases consistently, despite occasional small increases in the loss values.

Fig. 4.5(C) shows the NN model $\hat{y}(t; w)$ at the 0th, 1,000th, 10,000th, and 50,000th training iterations, from left to right, and the predicted solution for Example 2 are marked with red stars. As shown in the figure, the NN model at the 1,000th iteration is already very close to the final result at the 50,000th iteration. Fig. 4.5(D) shows the results of the RK method after accumulating 0, 5,000, 25,000, and 50,000 collocation points, and the predicted solutions are marked with yellow stars.

Table 4.2 shows the predictions and their NPE errors of our NN approach and the RK method for Example 2 at different iterations. Thanks to the adoption of the projection function in Alg. 2, our NN approach has a lower initial NPE error compared to the RK method. By the 1,000th iteration, the NPE error associated with our NN approach has been reduced to 0.14. By the 5,000th iteration, the NPE error has further diminished to 0.01. Upon completion of 10,000 iterations, our NN approach yields an optimal solution of $[1.08, 0.00, 0.64, 4.00]$, which accurately resolves Example 2.

Variational Inequality

Example 3 : Consider the following variational inequality :

$$(x - x^*)^T G(x^*) \geq 0, \quad x \in \Omega, \tag{4.24}$$

where

$$G(x) = \begin{bmatrix} x_1 - \frac{2}{x_1+0.8} + 5x_2 - 13 \\ 1.2x_1 + 7x_2 \\ 3x_3 + 8x_4 \\ 1x_3 + 2x_4 - \frac{4}{x_4+2} - 12 \end{bmatrix}, \quad \Omega = \{x \in \mathbb{R}^4 \,|1 \leq x_1 \leq 100, -3 \leq x_2 \leq 100, \\ -10 \leq x_3 \leq 100, 1 \leq x_4 \leq 100\}. \tag{4.25}$$

We reformulate Example 3 as $NPE(\Omega, G)$ by Proposition 4.2. Then, $NPE(\Omega, G)$ is modeled by the ODE system of Eq. (4.7). We establish an IVP by specifying initial point as $y(0) = \mathbf{0}$ and the time interval as $[0, 10]$. We use the NN model, $\hat{y}(t; w)$, to serve as an approximate state solution for this IVP, where the end state $\hat{y}(t = 10; w)$ is the predicted solution for Example 3.

Fig. 4.6(A) and Fig. 4.6(B) illustrate the decrease in loss and NPE error, respectively. Specifically, the loss decreases from 126.99 to 0.10, while the NPE error decreases from 15.42 to 0.00. Note that there are small fluctuations in the loss value around the 18,000th iteration, but the NPE error remains unaffected because the training algorithm retains the best prediction from its training history.

Fig. 4.6(C) shows the NN model $\hat{y}(t; w)$ at the 0th, 1,000th, 10,000th, and 50,000th training iterations, from left to right, and the predicted solution for

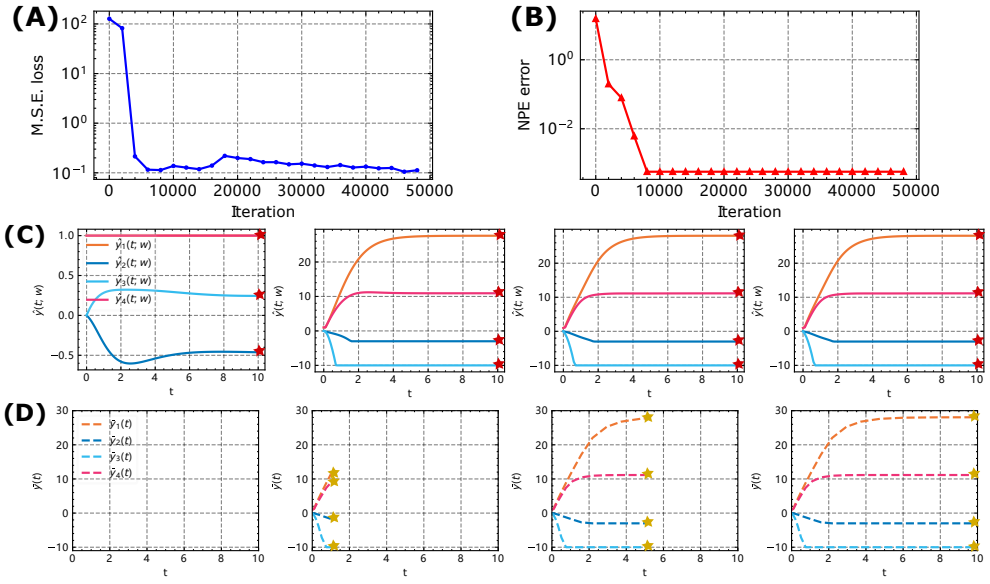


Figure 4.6 – Solving Example 3 : (A) MSE loss versus training iterations. (B) NPE error versus training iterations. (C) Evolution of the NN solution, $\hat{y}(t; w)$. (D) Evolution of the the RK solution, $\bar{y}(t)$.

Table 4.3 – Comparison of predicted solutions for Example 3 between our approach and the RK method

Our NN approach			The RK method		
Iteration	Prediction	NPE error	Collocation point	Prediction	NPE error
0	[1.00, -0.46, 0.24, 1.00]	**15.42**	0	[1.00, 0.00, 0.00, 1.00]	13.11
100	[1.77, 0.12, -1.57, 1.66]	**11.43**	100	[1.00, -0.02, -0.16, 1.00]	13.22
1000	[27.62, -3.00, -10.00, 10.92]	**0.46**	1000	[2.61, -0.22, -1.75, 2.28]	12.09
5000	[28.15, -3.00, -10.00, 11.18]	**0.08**	5000	[11.38, -1.69, -10.00, 8.88]	10.26
10000	[28.07, -3.00, -10.00, 11.15]	**0.00**	10000	[20.78, -3.00, -10.00, 10.87]	7.32
30000	[28.07, -3.00, -10.00, 11.15]	**0.00**	30000	[27.96, -3.00, -10.00, 11.15]	0.11
49999	[28.07, -3.00, -10.00, 11.15]	**0.00**	49999	[28.07, -3.00, -10.00, 11.15]	0.00

Example 3 are marked with red stars. In particular, at the 1,000th iteration, i.e., the first subplot on the left in Fig. 4.6(C), the NN model is already very close to the final result of the 50,000th iteration. Fig. 4.6(D) shows the results of the RK method after accumulating 0, 5,000, 25,000, and 50,000 collocation points, and the predicted solutions are marked with yellow stars.

Table 4.3 shows the predictions for Example 3 provided by our NN approach and the RK method. Remarkably, after only 1,000 iterations, our NN approach yields an acceptable prediction with an NPE error of 0.46. After 5,000 iterations, our NN approach refines this prediction further to an NPE error of less than 0.1. After 10,000 iterations, our approach converges to the solution of $[28.07, -3.00, -10.00, 11.15]$, which is an optimal solution of Example 3.

4.5.2 . Comparision with PINN

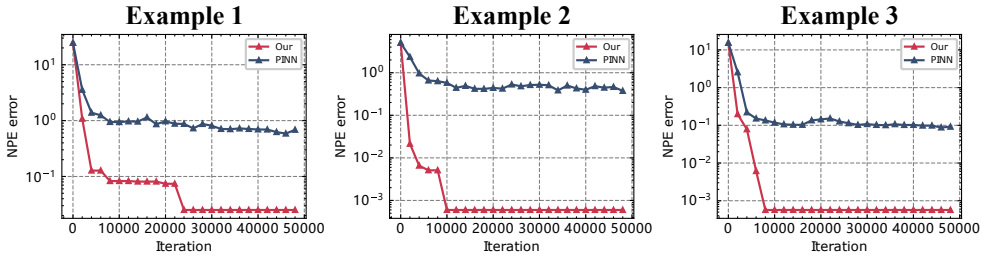


Figure 4.7 – Comparison of NPE errors between our proposed approach and the PINN for the three examples given in Section 4.5.1.

In this subsection, we compare our approach with PINNs in terms of accuracy. Specifically, we have chosen the basic version of PINN [26] for comparison. Despite the existence of more advanced models such as CPINNs [210] or XPINNs [113], however, we find that the basic version of PINN is sufficiently effective for the NPE problem, obviating the need for more complex variants. Regarding the setting for the PINN, we use the same network architecture and hyperparameters as in Section 4.5.1.

Our approach can be viewed as a modification of the PINN, specifically designed for the NPE problem to improve computational performance. The core distinction between our approach and the PINN lies in our focus on the end state of the NN model. Building on this, we employ the ORR mechanism in Alg. 2, which continuously monitors the NPE error of the end state throughout the training process. Therefore, the NN model optimizes simultaneously for both the ODE system and the NPE problem at hand, consistently maintaining the best result throughout the solution process. This unique focus results in improved performance, as demonstrated below.

Fig. 4.7 compares our approach with the PINN for solving the three examples given in Section 4.5.1. As shown in the figure, both approaches start with similar initial errors, but our approach is significantly outperforms the PINN as training progresses. Specifically, in Example 1, the NPE error converges to 0.03 with our approach, while the PINN converges to 0.7. In Example 2, our approach converges to less than 0.001, while the PINN could only converge to 0.4. In Example 3, our approach again converges to less than 0.001, while the PINN only converges to 0.1. These experimental results show that our approach provides superior solutions for solving these NPE problems. Moreover, this validates the effectiveness of the key designs in Alg. 2, such as the use of the ORR mechanism and the projection function.

4.5.3 . Hyperparameter Study

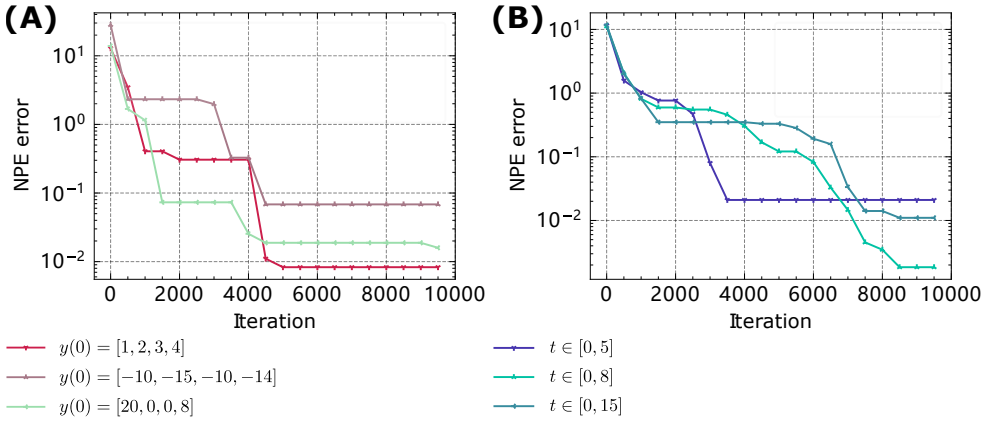


Figure 4.8 – (A) NPE errors versus training iterations for three different initial points. (B) NPE errors versus training iterations for three different time ranges.

Table 4.4 – Predictions and their NPE errors at different training iterations for three different initial points

	Initial point : $y(0) = [1, 2, 3, 4]$		Initial point : $y(0) = [-10, -15, -10, -14]$		Initial point : $y(0) = [20, 0, 0, 8]$	
Iteration	Prediction	NPE error	Prediction	NPE error	Prediction	NPE error
0	[1.00, 2.79, 2.86, 3.18]	12.86	[1.00, -3.00, -10.00, 1.00]	28.11	[20.10, 1.38, 0.23, 7.94]	13.89
100	[4.37, -0.06, -0.55, 5.96]	9.45	[1.00, -3.00, -10.00, 1.00]	28.11	[17.92, -2.25, -3.62, 6.54]	6.43
300	[23.89, -3.00, -10.00, 13.22]	4.19	[11.23, 5.10, 0.28, 6.14]	10.28	[26.40, -3.00, -10.00, 11.98]	1.68
500	[25.77, -3.00, -10.00, 12.87]	3.48	[14.21, -0.67, -10.00, 10.50]	2.33	[26.40, -3.00, -10.00, 11.98]	1.68
1000	[28.47, -3.00, -10.00, 11.28]	0.41	[14.21, -0.67, -10.00, 10.50]	2.33	[27.88, -3.00, -10.00, 11.72]	1.15
3000	[28.37, -3.00, -10.00, 11.30]	0.31	[30.04, -3.00, -10.00, 12.00]	1.97	[28.14, -3.00, -10.00, 11.14]	0.07
5000	[28.06, -3.00, -10.00, 11.16]	0.01	[28.01, -3.00, -10.00, 11.19]	0.07	[28.09, -3.00, -10.00, 11.15]	0.02
10000	[28.06, -3.00, -10.00, 11.16]	0.01	[28.01, -3.00, -10.00, 11.19]	0.07	[28.06, -3.00, -10.00, 11.16]	0.01

In this subsection, we explore the influence of various hyperparameters on the computational performance of our NN model. Specifically, we focus on Example

Table 4.5 – Prediction and their NPE errors at different training iterations for three different time ranges.

	Time range : $t \in [0, 5]$		Time range : $t \in [0, 8]$		Time range : $t \in [0, 15]$	
Iteration	Predition	NPE error	Predition	NPE error	Predition	NPE error
0	[1.00, 1.23, -0.22, 1.00]	11.56	[1.00, 0.63, 0.15, 1.00]	11.183	[1.00, 0.90, -0.40, 1.00]	11.74
100	[5.47, -0.52, -6.10, 5.25]	10.44	[5.53, -0.56, -5.13, 4.76]	10.59	[4.30 -0.236 -4.644 4.08]	10.27
300	[22.82, -3.00, -10.00, 14.27]	6.28	[23.81, -3.00, -10.00, 14.69]	7.14	[22.12, -3.00, -10.00, 14.18]	6.12
500	[27.04, -3.00, -10.00, 11.93]	1.57	[27.73, -3.00, -10.00, 12.17]	2.06	[27.14, -3.00, -10.00, 12.14]	2.00
1000	[27.04, -3.00, -10.00, 11.66]	1.03	[27.26, -3.00, -10.00, 11.55]	0.81	[27.25, -3.00, -10.00, 11.55]	0.82
3000	[28.08, -3.00, -10.00, 11.11]	0.08	[28.62, -3.00, -10.00, 10.91]	0.55	[27.77, -3.00, -10.00, 10.98]	0.35
5000	[28.05, -3.00, -10.00, 11.16]	0.02	[28.18, -3.00, -10.00, 11.21]	0.12	[28.40, -3.00, -10.00, 11.32]	0.33
9999	[28.05, -3.00, -10.00, 11.16]	0.02	[28.07, -3.00, -10.00, 11.15]	0.00	[28.08, -3.00, -10.00, 11.15]	0.01

3, as given in Section 4.5.1. The hyperparameters under investigation include the initial point, the time range, the number of hidden layers, the number of neurons, and the activation function used.

Initial point. Fig. 4.8(A) and Table 4.4 show the results of different initial point configurations with the same time range of $[0, 10]$. The results suggest the following :

— All initial points converge to the same optimal solution, as supported by Theorem 4.3.

— The convergence rates of different initial points vary, with initial points closer to the optimal solution converging faster.

— The initial point $y(0) = [1, 2, 3, 4]$ is closest to the optimal solution and achieves the fastest convergence with the smallest initial NPE error.

— The initial point $y(0) = [-10, -15, -10, -14]$ is the furthest away from the optimal solution and still converges, but has a higher initial error and slower convergence rate.

Time Range. Fig. 4.8(B) and Table 4.5 show the results of different time range configurations with the same initial points of $y(0) = [0, 0, 0, 0]$. The results suggest the following :

— Shorter time ranges lead to faster convergence but may result in less accurate predictions. As shown in the table, the shortest time range of $[0, 5]$ converges very fast, but its NPE error does not decrease much after 3,000 iterations.

— Longer time ranges provide better predictions, but require more training iterations. The longest range of $[0, 15]$ converges slowly, but with more training it can give better results than the other two ranges.

— The choice of time ranges represents a trade-off. Longer ranges may enhance accuracy but require more training, whereas shorter ranges are easier to train but may yield less satisfactory predictions. As shown in Table 4.5, considering a fixed maximum number of iterations at 10,000, the time range of $[0, 8]$ achieves the optimal performance.

Table 4.6 – Comparison of NPE errors for NNs with different model sizes. The top half of the table presents the results for single-layer networks with different numbers of neurons, while the bottom half presents the results for multi-layer networks with 500 neurons per layer.

Iteration	1 layer, 100 neurons	1 layer, 500 neurons	1 layer, 1000 neurons	1 layer, 1500 neurons	1 layer, 2000 neurons
100	11.43	0.33	0.23	0.78	2.32
300	7.08	0.33	0.23	0.75	0.71
500	2.07	0.33	0.23	0.52	0.71
1000	0.46	0.33	0.23	0.16	0.29
3000	0.10	0.00	0.03	0.02	0.02
5000	0.08	0.00	0.00	0.01	0.01
10000	0.00	0.00	0.00	0.00	0.01

Iteration	1 layer each, 500 neurons	2 layer each, 500 neurons	3 layer each, 500 neurons	4 layer 500, each neurons	5 layer each, 500 neurons
100	0.33	1.73	2.22	6.43	6.88
300	0.33	0.10	0.76	1.82	6.70
500	0.33	0.07	0.27	0.57	0.52
1000	0.33	0.01	0.03	0.23	0.34
3000	0.00	0.01	0.01	0.03	0.03
5000	0.00	0.00	0.00	0.03	0.03
10000	0.00	0.00	0.00	0.02	0.03

Number of Layers and Neurons. Table 4.6 shows the NPE errors of different model sizes, specifically different numbers of hidden layers and neurons. As shown in the table, all models of different sizes converge to solutions with NPE errors below 0.03 by the 10,000th iteration. However, the model size has a significant impact on the speed of convergence. Models that are too small or too large perform worse than the others. For example, the 100-neuron single-layer model has higher NPE errors in the first 1000 iterations. The 5-layer model with 500 neurons per layer has similar results.

Therefore, selecting an appropriately-sized model is importance to achieve optimal performance. Small model sizes have limited capacity, while large model sizes require a lot of training to converge, which may not be necessary. Among the model sizes considered in Table 4.6, the two-layer model with 500 neurons each achieves the best performance. It converges to an NPE error of 0.1 within 300 iterations and further reduces to 0.01 within 1000 iterations, outperforming other model architectures.

Table 4.7 – Comparison of NPE errors for different activation functions. The considered NN model is a two-layer network, with 500 neurons in each layer.

Iteration	2 layers, each 500 neurons				
	tanh	sinx	sigmoid	relu	leaky relu
100	1.73	2.18	3.01	2.62	2.32
300	0.10	0.73	0.28	1.13	0.71
500	0.07	0.15	0.05	1.13	0.71
1000	0.01	0.03	0.01	1.13	0.29
3000	0.01	0.03	0.01	0.01	0.01
5000	0.00	0.03	0.01	0.01	0.01
10000	0.00	0.03	0.00	0.01	0.00

Activation Function. Table 4.7 shows the NPE errors for different activation functions on the same NN model. The data show that regardless of the activation function chosen, all NN models converge to solutions with NPE errors less than 0.03. Notably, the influence of the activation function on model performance is relatively minor when compared to the effects of model size. Among the tested activation functions, ReLU and Leaky ReLU were found to be slightly less effective than the others. Specifically, the tanh activation function stands out as the most efficient, which corroborates its widespread adoption in research related to PINNs [211, 212, 213].

4.5.4 . Large Scale NPE

Consider the following NPE problem : For $i = 1, 2, \ldots, 1000$,

$$\left(x_i^* - \frac{1}{2\sqrt{(Mx^*)_i + q_i}} \right)^+ = x_i^*. \tag{4.26}$$

The objective is to find an optimal solution $x^* = [x_1^*, x_2^*, \ldots, x_{1000}^*] \in \mathbb{R}^{1000}$ that solves Eq. (4.26). The problem data $M \in \mathbb{R}^{1000\times1000}$ is partitioned as

$$M = \begin{bmatrix} M_1 & M_2 \\ M_3 & M_4 \end{bmatrix}, \tag{4.27}$$

where $M_1 \in \mathbb{R}^{500\times500}$, $M_2 \in \mathbb{R}^{500\times500}$, $M_3 \in \mathbb{R}^{500\times500}$, and $M_4 \in \mathbb{R}^{500\times500}$ are given by

$$\mathsf{M}_1 = \begin{bmatrix} 1.2 & 0.6 & \ldots & 0.6 \\ 0.6 & 1.2 & \ldots & 0.6 \\ \vdots & \vdots & \ddots & \vdots \\ 0.6 & 0.6 & \ldots & 1.2 \end{bmatrix}, M_2 = \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{bmatrix}, M_3 = \begin{bmatrix} -1 & 0 & \ldots & 0 \\ 0 & -1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & -1 \end{bmatrix}, M_4 = \begin{bmatrix} 0 & 0 & \ldots & 0 \\ 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 0 \end{bmatrix}. \tag{4.28}$$

For the problem data $q \in \mathbb{R}^{1000}$, the first half is sampled uniformly from the interval $[-31, -28]$, while the second half is sampled uniformly from the interval $[3, 10]$. We create a problem set by generating ten different q, resulting in ten different NPE problems. All the sampled q can be accessed through the link [1]. Solving these NPE problems is non-trivial due to their high dimensionality with 1,000 variables and the presence of multiple nonlinear operations. Therefore, the RK method may encounter computational inefficiencies when applied to these NPE problems.

Table 4.8 – Performance of the NN approach for solving the set of NPE problems. STD stands for standard deviation. CPU time is measured in seconds.

Iteration	M.S.E. loss (Mean ± STD)	CPU time (Mean ± STD)	NPE error (Mean ± STD)	NPE error (50%-quantile)	NPE error (75%-quantile)	NPE error (95%-quantile)
0	470.29 ± 92.68	0.00 ± 0.00	4839.33 ± 673.10	4720.42	4911.21	6008.41
100	45.85 ± 39.83	2.36 ± 0.60	15.34 ± 16.46	6.65	19.98	46.38
500	32.10 ± 26.70	11.32 ± 0.68	0.67 ± 0.89	0.37	0.46	2.30
1000	26.81 ± 11.53	22.57 ± 1.16	0.31 ± 0.18	0.27	0.36	0.60
3000	15.20 ± 11.23	67.82 ± 1.40	0.19 ± 0.06	0.18	0.24	0.27
5000	12.33 ± 10.44	112.91 ± 2.20	0.13 ± 0.07	0.11	0.17	0.24
7000	8.16 ± 8.13	157.98 ± 2.86	0.10 ± 0.06	0.07	0.14	0.19
10000	2.93 ± 3.38	225.47 ± 3.82	0.05 ± 0.04	0.04	0.07	0.12
30000	0.73 ± 1.18	670.38 ± 5.82	0.01 ± 0.01	0.01	0.01	0.02

We use the proposed NN approach to solve the ten large-scale NPE problems. With respect to the experimental setup, the employed network architecture consisted of a three-layer fully connected network, each layer having 500 neurons and utilizing the tanh activation function. The time range chosen for the experiments

1. https://github.com/wuwudawen/IJNME_data_2023/blob/main/Q.npy

Table 4.9 – Performance of the RK method for solving the set of NPE problems STD stands for standard deviation. CPU time is measured in seconds.

Time range	CPU time (Mean $\pm$ STD)	NPE error (Mean $\pm$ STD)	NPE error (50%-quantile)	NPE error (75%-quantile)	NPE error (95%-quantile)
$[0, 2]$	369.33 ± 44.27	1.87 ± 0.16	1.86	2.01	2.08
$[0, 4]$	547.93 ± 74.83	0.48 ± 0.06	0.50	0.51	0.55
$[0, 6]$	739.74 ± 88.02	0.14 ± 0.02	0.15	0.15	0.16
$[0, 8]$	1048.92 ± 88.20	0.04 ± 0.01	0.04	0.05	0.05
$[0, 10]$	1542.07 ± 205.43	0.01 ± 0.01	0.01	0.01	0.02

is $[0, 10]$, and the initial point is set to a zero vector. All other hyperparameter settings are kept consistent with those described in Section 4.5.1. Tables 4.8 and 4.9 present the experimental results of our NN approach compared to the RK method, focusing on both accuracy and computational time. In order to offer a comprehensive understanding of the experimental results, we provide statistical descriptors of the outcomes, including the mean, standard deviation (STD), and various percentiles.

In the following, we discuss the differences between our NN approach and the RK method in terms of computational efficiency, stability and convergence.

— Efficiency : As shown in the tables, our NN approach outperforms the RK method in terms of computational efficiency when solving large scale NPE problems. In particular, our NN approach requires less CPU time than the RK method for the same level of accuracy. For example, our approach achieves an NPE error of 0.01, requiring an average of 30,000 iterations and consuming only 670.38 seconds. On the contrary, the RK method requires spanning a time range of $[0, 10]$, which requires a higher average time of 1542.07 seconds. Moreover, the efficiency gap becomes more significant as the required accuracy is relaxed. For example, for an NPE error threshold of less than 0.50, our NN approach takes on average only 500 iterations and 11.32 seconds, making it about 48 times faster than the RK method, which requires 547.93 seconds.

— Stability : Our NN approach demonstrates not only computational efficiency, but also greater stability, reflected in a much lower STD of CPU times - only 5.82 at the 30,000th iteration. This indicates consistent and reliable performance across different NPE problem instances. On the other hand, the RK method exhibits higher variability with an STD of 205.43 for the time range of $[0, 10]$, signifying greater sensitivity to the specific NPE problem at hand.

— Convergence : While there are notable differences in computational efficiency and stability between our NN approach and the RK method, both techniques demonstrate comparable accuracy in the long run, each achieving an NPE

error as low as 0.01. As discussed in Section 4.4.3, the convergence mechanisms of the two approaches are fundamentally different. Our NN approach operates within a fixed time range of $[0, 10]$ and refines its predictive accuracy through iterative training. In contrast, the RK method improves its accuracy by progressively extending the time range.

4.5.5 . Discussion

Below, we summarize the key features of our proposed NN approach :

— Our NN approach reliably converges to the optimal solutions of the NPE problems. This is supported by Theorem 4.3 in neurodynamic optimization and the universal approximation theorem of NNs. Experimentally, we have shown that the NN approach successfully found optimal solutions for the three types of NPE problems in Section 4.5.1, as well as ten large-scale NPE problems in Section 4.5.4.

— The proposed NN approach outperforms the PINN in solving NPE problems. This improvement is due to some modifications made to the basic PINN approach that allow it to exploit the problem structure of NPEs for better performance. In particular, during each training iteration, the NN model evaluates the accuracy of its end state against the target NPE problem and retains the best performing solution. Additionally, we employ the projection function of Eq. (4.1) to further boost accuracy.

— The computational performance of our NN approach is significantly affected by the hyperparameter settings. As discussed in Section 4.5.3, choices regarding the time range and initial point have a strong impact on the convergence result. For example, smaller time ranges may prevent the NN model from converging to an optimal solution, regardless of the training time. Moreover, in our empirical observations, the network architecture and activation functions influence the model convergence rate.

— Our NN approach excels in solving large-scale NPE problems. As presented in Section 4.5.4, our NN approach outperforms the RK method in terms of computational efficiency. For the same level of accuracy, the CPU time required by our approach is less than that required by the RK method. In addition, our approach exhibits greater stability, with its solution time remaining consistent across different NPE problems.

However, we must acknowledge some limitations of our proposed NN approach. The most notable limitation is that, unlike traditional RK methods, the NN approach lacks rigorous theoretical underpinnings to guarantee convergence, primarily due to the black-box nature of NNs. Furthermore, as elaborated in Section 4.5.3, the performance of the model is highly sensitive to hyperparameter choices and architectural decisions, requiring careful tuning. Numerical integration methods, which are typically simpler, avoid these complexities.

To address these limitations, future research should focus on improving network design and training methods. This could include developing more effective hyperparameter search strategies or exploring advanced network architectures, such as attention-based models or transformers. In addition, investigating how to incorporate domain-specific knowledge into the network structure may improve solution accuracy and reduce training time.

4.6 . Conclusion

In this chapter, we presented an innovative deep learning-based approach for solving NPEs based on neurodynamic optimization and PINNs. We showed how our approach can efficiently solve NPEs and highlighted its advantages over PINNs and the RK method. The proposed approach transforms NPE problems into NN training problems, allowing the use of the latest advances in machine learning and deep learning to solve NPEs. We also identified areas for future research, such as exploring better methods for selecting initial points and time ranges, experimenting with different network architectures, and investigating advanced neurodynamic optimization techniques, among others.

In summary, our proposed framework shows considerable promise for improving computational efficiency in solving NPEs. With ongoing research and development, we expect to further strengthen the robustness of our approach and position it as a valuable computational tool for addressing a wide range of nonlinear optimization challenges in diverse applications.

5 - Solving Nonsmooth Convex Optimization Problems with Neurodynamic Optimization and PINNs

In this chapter, we consider Nonsmooth Convex Optimization Problems (NCOPs), which are a more general class than smooth nonlinear convex optimization. The NCOPs under consideration naturally encompass various types of nonlinear convex optimization problems, including linear programming, quadratic programming, and second-order cone programming, among others. The approach presented in this chapter is similar to the one introduced in Chapter 4, and their solving algorithms are both based on neurodynamic optimization and Physics-Informed Neural Networks (PINNs).

The main differences between the approach in this chapter and that in Chapter 4 are :

— We use a different neurodynamic method to model the NCOP.

— We use a projection function to handle the equality constraints in NCOPs.

— We use a different evaluation metric specifically the NCOP to improve performance.

The main contribution of this chapter is to present a new paradigm for solving classical nonlinear programming problems. The proposed solution algorithm is based on deep learning and can be implemented entirely on PyTorch or Tensorflow.

This chapter corresponds to the reference [147].

5.1 . Introduction

Constrained nonlinear optimization problems involve finding the best solution among a set of possible solutions by minimizing or maximizing an objective function. These problems are prevalent in various fields such as engineering, physics, finance, and management, with a wide range of applications. They can be divided into two groups based on the nature of the objective or constraint functions : convex and nonconvex optimization problems. Convex optimization problems, which include linear programming and quadratic programming, are a special class of nonconvex optimization problems and have been studied extensively. Methods such as the primal-dual interior point method have been developed to solve them efficiently [35, 4]. Nonconvex optimization problems, however, are more complex and commonly solved through gradient descent-based algorithms [214, 215]. Consider solving smooth and deterministic nonconvex problems using gradient descent. Carmon et al. [216] showed that $\Omega(\epsilon^{-1})$ gradient evaluations are necessary to find a

ϵ-stationary point. For smooth and stochastic settings, Arjevani et al. [217] showed that $\Omega(\epsilon^{-2})$ noisy gradient evaluations are required to find a ϵ-stationary point.

In this chapter, we focus on Nonsmooth Convex Optimization Problems, which have a non-differentiable objective function or constraint function. While traditional optimization algorithms have proven effective in solving smooth convex optimization problems, they encounter limitations when applied to NCOPs. This limitation arises because most traditional algorithms are gradient-based, and gradient information is not available in the context of NCOPs. Therefore, algorithms such as subgradient methods and bundle methods have been developed specifically to overcome the challenges associated with the nondifferentiable functions [6, 218]. Among the many solution methods, we explore neurodynamic approaches for solving NCOPs [219, 220, 221, 51, 222]. This approach entails the deployment of a circuit-based neurodynamic model capable of real-time optimization problem-solving. Such methods hold promise for a wide array of applications, including but not limited to resource allocation [191], feature selection [223], and the coordination of multi-manipulator systems [224].

In this chapter, we propose a deep learning approach, based on neurodynamic optimization [221] and PINNs [26], for solving NCOPs. The main steps are outlined below :

— First, we use the neurodynamic method proposed by [221] to model the NCOP as an Ordinary Differential Equation (ODE) system.

— Second, we adapt the PINN model [26] to solve the ODE system. In particular, the end state of the PINN represents a prediction for the NCOP.

— Third, we train the PINN to solve the NCOP. The training algorithm focuses on improving the end state of the PINN.

The remaining sections are organized as follows : Section 5.2 illustrates the neurodynamic approach used to model NCOPs. Section 5.3 describes our proposed approach, including model introduction, loss definition, and model training. Section 5.4 presents the experimental results of solving some NCOP instances using the proposed deep learning approach. Finally, Section 5.5 summarizes the main results of this chapter and outlines possible directions for future research.

5.2 . Neurodynamic Approach to Model NCOPs

NCOP. We consider the following optimization problem :

$$\begin{cases} \min\limits_{x} f(x) \\ \text{s.t.} \\ \quad g(x) \leq \mathbf{0} \\ \quad Ax = b, \end{cases} \tag{5.1}$$

where $x = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n$ is the decision variables, and $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function. $g(x) = (g_1(x), g_2(x), \ldots, g_m(x))^T : \mathbb{R}^n \to \mathbb{R}^m$ represents the inequality constraints, and $u = (u_1, u_2, \ldots, u_m)$ represents the dual variables of the inequality constraints. $Ax = b$ represents the equality constraints with $A \in \mathbb{R}^{k\times n}$ and $b \in \mathbb{R}^k$. n, m, and k denote the number of decision variables, inequality constraints, and equality constraints, respectively.

In this chapter, we consider the case where $f(x)$ and $g(x)$ are convex but not necessarily smooth, and A is of full row rank. We denote x^* and u^* as the optimal primal and dual solutions, respectively.

Definition 5.1 (Subgradient and subdifferential) *A vector $l \in \mathbb{R}^n$ is a subgradient of $f : \mathbb{R}^n \to \mathbb{R}$ at $x \in \operatorname{dom} f$ if the following holds*

$$f(z) \geq f(x) + l^T(z - x), \quad \forall z \in \operatorname{dom} f. \tag{5.2}$$

The set of all subgradients of f at x is called the subdifferential of f at x and is denoted by $\partial f(x)$.

Neurodynamic Approach. Now, let $x : \mathbb{R} \to \mathbb{R}^n$ and $u : \mathbb{R} \to \mathbb{R}^m$ be some time dependent functions. The aim of a neurodynamic approach is to construct a first-order ODE system to govern $x(t)$ and $u(t)$, such that they will settle down to the optimal primal and dual solutions of the NCOP (5.1). In this chapter, the two-layer neurodynamic approach in [221] is adopted, where the ODE system is described as follows :

$$\begin{aligned} \frac{dx}{dt} \in& -(I - U)\left[\partial f(x) + \partial g(x)^T(u + g(x))^+\right] - A^T\rho(Ax - b), \\ \frac{du}{dt} =& \frac{1}{2}\left(-u + (u + g(x))^+\right), \end{aligned} \tag{5.3}$$

where $U = A^T\left(AA^T\right)^{-1}A$, $I \in \mathbb{R}^{n\times n}$ is the identity matrix, $\rho(s) = \left(\tilde{\rho}\left(s_1\right), \tilde{\rho}\left(s_2\right), \ldots, \tilde{\rho}\left(s_k\right)\right)^T$, and for $i = 1, 2, \ldots, k$,

$$\tilde{\rho}\left(s_i\right) = \begin{cases} 1 & \text{if } s_i > 0, \\ [-1, 1] & \text{if } s_i = 0, \\ -1 & \text{if } s_i < 0. \end{cases} \tag{5.4}$$

To simplify the discussion, we denote $y(t) = (x(t)^T, u(t)^T)^T$ and define :

$$\Phi(y) = \begin{bmatrix} -(I - U)\left[\partial f(x) + \partial g(x)^T(U + g(x))^+\right] - A^T\rho(Ax - b) \\ \frac{1}{2}\left(-U + (U + g(x))^+\right) \end{bmatrix}. \tag{5.5}$$

Thus, the ODE system (5.3) can be written as $\frac{dy}{dt} \in \Phi(y)$.

Definition 5.2 (State solution) *Consider an ODE system $\frac{dy}{dt} \in \Phi(y)$, where $\Phi : \mathbb{R}^n \to \mathbb{R}^n$. Given $(t_0 \in \mathbb{R}, y_0 \in \mathbb{R}^n)$, a vector value function $y : \mathbb{R} \to \mathbb{R}^n$ is called*

a state solution, if it satisfies the ODE system $\frac{dy}{dt} \in \Phi(y)$ *and the initial condition* $y(t_0) = y_0$.

In particular, we call $y(t)$ *the state at time* t. *Given a time interval* $[t_0, T]$, *we call* $y(T)$ *the end state on that time interval.*

Theorem 5.1 ([221]) *Consider a NCOP* (5.1) *and its derived ODE system* (5.3). *Given any initial condition* $y(t_0) = y_0$, *the state solution* $y(t)$ *of the ODE system converges to an optimal solution* $y*$ *as time* t *approaches infinity, i.e.*

$$\lim_{t \to \infty} y(t) = y^*, \tag{5.6}$$

where $y^* = (x^{*T}, u^{*T})^T$, x^* *and* u^* *are the optimal primal and dual solutions to the NCOP.*

In particular, if the NCOP contains only one optimal solution x^*, *then the ODE system is called globally asymptotically stable at* y^*.

Initial Value Problem (IVP) Construction. In practice, in order to use the neurodynamic approach to solve the NCOP, we need to construct an IVP consisting of three components : 1) the ODE system (5.3), 2) an initial condition $y(t_0) = y_0$, and 3) a time range $t \in [t_0, T]$. $y(t)$ for $t \in [t_0, T]$ represents the state solution of this IVP problem over the time range $[t_0, T]$, where the end state, $y(T)$, is considered to be the predicted solution to the NCOP. According to Theorem 5.1, the larger the time range $[t_0, T]$, the closer $y(T)$ is to the optimal solution y^* of the NCOP.

5.3 . Methodology

5.3.1 . Modified PINN

Model Description. We propose a modified PINN model to solve the NCOP. Our model can be expressed by the following equation :

$$\hat{y}(t; w) = y_0 + \left(1 - e^{-(t-t_0)}\right) \mathrm{N}(t; w), \quad t \in [t_0, T], \tag{5.7}$$

where $\mathrm{N}(t; w)$ is a fully connected neural network with trainable parameters w. y_0 is a given initial point for the ODE system. $[t_0, T]$ is a given time range. The auxiliary function $\left(1 - e^{-(t-t_0)}\right)$ ensures that the neural network always satisfies the initial condition $\hat{y}(t = t_0; w) = y_0$ regardless of w.

Approximate State Solution to the ODE. As shown in Figure 5.1 (Left), the proposed model (5.7) itself is an approximate state solution of the ODE system (5.3) on the time range $[t_0, T]$, i.e,

$$\hat{y}(t; w) \approx y(t), \quad t \in [t_0, T], \tag{5.8}$$

where $y(t)$ is the true state solution of the ODE system. While the input time t of the model $\hat{y}(t; w)$ can be take any real number, we specifically use $\hat{y}(t; w)$ as the

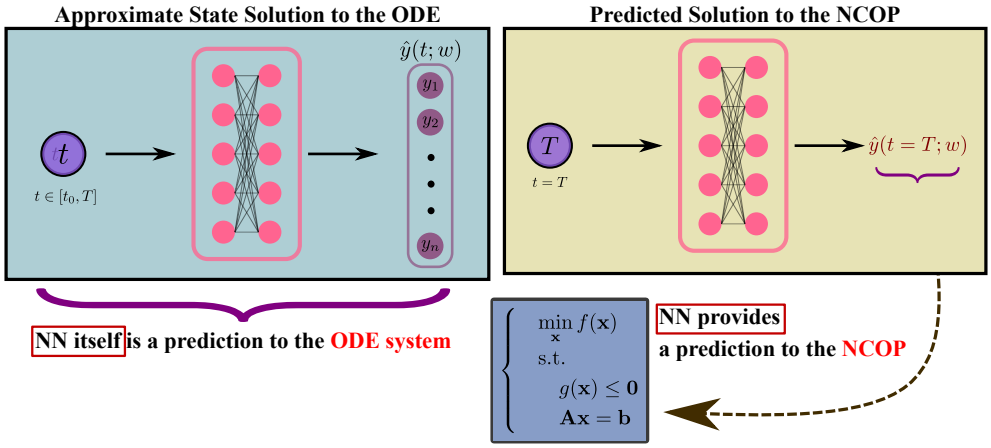


Figure 5.1 – Neural network solution for the ODE system and the NCOP. Left : When $t \in [t_0, T]$, the model $\hat{y}(t; w)$ itself is considered to be an approximate state solution of the ODE system. Right : When $t = T$, the model returns the prediction, $(\hat{x}, \hat{u}) = \hat{y}(t = T; w)$, where $\hat{x}$ and $\hat{u}$ represent the primal and dual predictions for the NCOP, respectively.

solution of the ODE over the time range $[t_0, T]$. As a result, we restrict the input of $\hat{y}(t; w)$ to the time range $t \in [t_0, T]$.

Predicted Solution to the NCOP. The end state of the proposed model, i.e., $\hat{y}(t = T; w)$, is used as the predicted solution to the NCOP (5.1), as shown in Figure 5.1 (Right). The following equation shows how $\hat{y}(t = T; w)$ approximates the optimal solution y^* :

$$\hat{y}(t = T; w) \approx y(T) \approx y^*. \tag{5.9}$$

Here, $\hat{y}(t = T; w) \approx y(T)$ indicates that the end state of our model approximates the true end state, and $y(T) \approx y^*$ comes from Theorem 5.1, indicating that the true end state is the predicted solution of the NCOP.

5.3.2 . Training Objective

Loss Function. We define the loss function of the proposed model (5.7) as follows :

$$\mathcal{L}(t, w) = \left\| \frac{\partial \hat{y}(t; w)}{\partial t} - \Phi(\hat{y}(t; w)) \right\|, \tag{5.10}$$

where $\Phi(\cdot)$ refers to the ODE system (5.3), which corresponds to the NCOP to be solved. $\|\cdot\|$ is the Euclidean norm. $\Phi(\hat{y}(t; w))$ is the expected derivative according to the ODE system. $\frac{\partial \hat{y}(t;w)}{\partial t}$ is the actual derivative of the model, which can be computed using automatic differentiation tools such as PyTorch or JAX [181, 207]. $\mathcal{L}(t, w)$ represents the difference between the two at time t and with network parameters w.

Embedding the NCOP into the Loss Function. The NCOP is integrated into the loss computation process through the ODE system rather than as a component of the neural network. A neural network is created as an empty framework without a specific goal to solve a particular NCOP. Instead, by reformulating the NCOP as an ODE system and embedding it into the loss function, the neural network is trained toward solving the NCOP.

Objective Function. The goal of training the proposed model is to minimize the following objective function :

$$J(w) = \int_{t_0}^{T} \mathcal{L}(t, w)dt, \tag{5.11}$$

which is the integral of the loss function over the time range $[t_0, T]$. The loss value $\mathcal{L}(t, w)$ represents the error of the model at time t, while the objective function $J(w)$ represents the total error of the model over the time range $[t_0, T]$.

Batch Loss. However, the objective function $J(w)$ is computationally intractable to compute due to its integral part. Therefore, in practice, we train the model by minimizing the following batch loss :

$$\mathcal{L}(\mathbb{T}, w) = \frac{1}{|\mathbb{T}|} \sum_{t \in \mathbb{T}} \mathcal{L}(t, w), \tag{5.12}$$

where $\mathbb{T}$ is a set of randomly sampled time points from the interval $[t_0, T]$, and $|\mathbb{T}|$ denotes the size of this set. In this way, we can approximate the integral in the objective function $J(w)$ by a sum of loss values over the set of sampled times. By minimizing the batch loss, we can effectively train the model to solve the NCOP.

5.3.3 . Algorithm Design

Objective value under Constraints (OuC) Metric. We introduce an evaluation metric called OuC to measure how well a predicted solution, x_{pred}, solves the NCOP :

$$\text{OuC}(x_{\text{pred}}) = \begin{cases} f(x_{\text{pred}}) & \text{if } x_{\text{pred}} \in \Omega, \\ +\infty & \text{otherwise ,} \end{cases} \tag{5.13}$$

where Ω is the feasible set defined as $\Omega = \{x | x \leq g(x), Ax = b\}$. If the predicted solution is not feasible, the OuC is set to positive infinity ; if it is feasible, the OuC is set to the objective value.

Projection Mapping onto Equality Constraints. To increase the likelihood of $\text{OuC}(x_{\text{pred}})$ being a real value instead of infinite, we employ the following projection function to map x_{pred} to the set of equality constraints,

$$P_{eq}(x_{\text{pred}}) = x_{\text{pred}} - A^T \left(AA^T\right)^{-1} (Ax_{\text{pred}} - b). \tag{5.14}$$

By definition, the evaluation metric $\text{OuC}(x_{\text{pred}})$ attains a real value only when x_{pred} lies within the feasible set Ω. As illustrated in Figure 5.2, there are two circumstances when P_{eq} aids in projecting x_{pred} onto the feasible set :

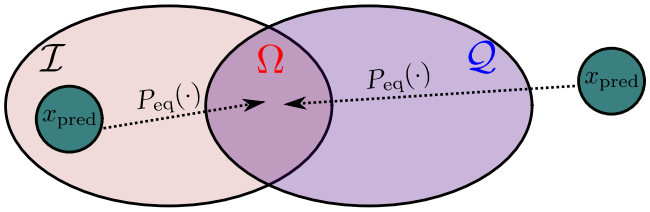


Figure 5.2 – $\mathcal{I}$ and $\mathcal{Q}$ denote the feasible set of inequality constraints and equality constraints, respectively. $\Omega = \mathcal{I} \cap \mathcal{Q}$ denotes the general feasible set of the problem. P_{eq} is a projection function that maps x_{pred} onto $\mathcal{Q}$.

— x_{pred} satisfies the inequality constraints but fails to meet the equality constraints, i.e., $x_{\text{pred}} \in \mathcal{I} - \mathcal{Q}$.

— x_{pred} does not satisfy both the inequality and equality constraints, i.e., $x_{\text{pred}} \notin \mathcal{I} \cup \mathcal{Q}$.

In both of the above scenarios, P_{eq} has a reasonable chance of mapping x_{pred} onto the feasible set Ω, resulting in the value of OuC being a real number rather than infinite.

Pipeline. Algorithm 3 summarizes how to use our proposed method to solve the NCOP. First, we need to specify an initial condition $y(t_0) = y_0$ and a time range $[t_0, T]$ to construct the IVP. Then, we instantiate the proposed model (5.7), which serves as an approximate state solution for this IVP. The model is trained by performing gradient descent on the batch loss (5.12) to improve the approximation. Note that our solver is completely based on the deep learning infrastructure and does not require any standard optimization solver or numerical integration solver.

Optimal Result Retention (ORR) Mechanism. A key to Algorithm 3 is that we use an ORR mechanism based on the OuC metric (5.13). Specifically, at each iteration, the algorithm compares the OuC value at the current iteration, denoted as OuC_{curr}, with the best OuC value found so far, denoted as OuC_{best}. $(\hat{x}_{\text{curr}}, \hat{u}_{\text{curr}})$ and $(\hat{x}_{\text{best}}, \hat{u}_{\text{best}})$ represent the current prediction and the best prediction found so far, respectively. If OuC_{curr} is less than OuC_{best}, it means that the model found a better prediction in this iteration. The algorithm updates OuC_{best} to equal OuC_{curr} and stores the best prediction as $(\hat{x}_{\text{best}}, \hat{u}_{\text{best}}) = (\hat{x}_{\text{curr}}, \hat{u}_{\text{curr}})$. This mechanism ensures that the best prediction obtained by the model is maintained throughout the training process, improving the overall performance of the algorithm.

Neural Network Solution Procedure. Our approach employs gradient descent on the batch loss (5.12) to improve the neural network prediction at each iteration. Assume that the maximum number of training iterations is M. The evolution of the neural network is represented by $\hat{y}(t; w_1), \hat{y}(t; w_2), \ldots, \hat{y}(t; w_M)$, where w_i and $\hat{y}(t; w_i)$ denote the network parameters and the approximate state solution at the i-th iteration, respectively. The predicted end states are $\hat{y}(t = T; w_1), \hat{y}(t =$

Algorithm 3 Deep learning solver for NCOPs based on neurodynamic optimization

Input : A NCOP as defined in (5.1); A time range $[t_0, T]$; An initial condition $y(t_0) = y_0$; Learning rate α.

Output : The predicted primal and dual solution to the NCOP, denoted as $\hat{x}_{\text{best}}$ and $\hat{u}_{\text{best}}$, respectively.

function
  Derive the ODE system, $\Phi(\cdot)$ by Eq. (5.3).
  Instantiate the proposed model $\hat{y}(t; w)$ of Eq. (5.7).
  $(\hat{x}_{\text{best}}, \hat{u}_{\text{best}}) = \hat{y}(t = T; w)$
  $\text{OuC}_{\text{best}} = \text{OuC}(\hat{x}_{\text{best}})$
  while iter $\leq$ maximum iteration **do**
    $\mathbb{T} \sim U(t_0, T)$ ▷ Sample collocation points $\mathbb{T}$.
    $\mathcal{L}(\mathbb{T}, w)$ ▷ Forward propagation.
    $w = w - \alpha \nabla_w \mathcal{L}(\mathbb{T}, w)$ ▷ Backward propagation.
    $(\hat{x}_{\text{curr}}, \hat{u}_{\text{curr}}) = \hat{y}(t = T; w)$ ▷ Neural network prediction
    $\hat{x}_{\text{curr}} = P_{eq}(\hat{x}_{\text{curr}})$ ▷ Project $\hat{x}_{\text{curr}}$ to the feasible set.
    $\text{OuC}_{\text{curr}} = \text{OuC}(\hat{x}_{\text{curr}})$ ▷ Calculate the OuC value of $\hat{x}_{\text{curr}}$.
    if $\text{OuC}_{\text{curr}} < \text{OuC}_{\text{best}}$ **then**
      $\text{OuC}_{\text{best}} = \text{OuC}_{\text{curr}}$
      $(\hat{x}_{\text{best}}, \hat{u}_{\text{best}}) = (\hat{x}_{\text{curr}}, \hat{u}_{\text{curr}})$
    end if
  end while
  return $(\hat{x}_{\text{best}}, \hat{u}_{\text{best}})$
end function

$T; w_2), \ldots, \hat{y}(t = T; w_M)$, where $\hat{y}(t = T; w_i)$ represents the prediction to the NCOP at the i-th iteration.

5.4 . Numerical Results

Neural Network Setup. To implement our proposed model and the ODE system, we used PyTorch 1.12.1 with CUDA 11.2 [181] and JAX 0.4.1 [207]. Our neural network architecture consisted of a single layer fully connected network with 100 neurons and a Tanh activation function. For training, we used the Adam optimizer [27] with a learning rate of 0.001, and a batch size of 128.

Evaluation metrics. We employed two metrics to assess model performance. The first is computation time, which measures computational efficiency. The second is the OuC metric, defined in equation (5.13), which measures accuracy. The OuC metric categorises a prediction into two scenarios. In the first scenario, if the prediction falls outside the feasible set, it returns 'inf'. In the second scenario, if the prediction is within the feasible set, the OuC metric returns the objective value of the optimization problem ; in this case, a lower OuC value indicates better performance. The rationale behind using this metric is its convenience in measuring

how swiftly the proposed method can find a feasible solution and the extent of improvement after finding one.

5.4.1 . Comparisons with Numerical Integration Methods

In this subsection, we compare our proposed method with six classical numerical integration methods : Runge-Kutta 45 (RK45), Runge-Kutta 23 (RK23), Dormand-Prince 853 (DOP853), Backward Differentiation Formula (BDF), Radau, and LSODA. All of these methods are available in the Scipy library [209]. The RK45, RK23, and DOP853 are explicit Runge-Kutta methods, while the BDF and Radau are implicit methods. LSODA is an adaptive method that automatically switches between explicit and implicit methods depending on the stiffness of the ODE system.

Example 1 : We aim to solve the following NCOP :

$$\begin{aligned} &\min_x f(x) = 10(x_1 + x_2)^2 + (x_1 - 2)^2 + 20|x_3 - 3| + e^{x_3} \\ &\text{s.t.} \\ &\quad g(x) = (x_1 + 3)^2 + x_2 - 36 \leq 0 \\ &\quad h(x) = 2x_1 + 3x_2 + 5x_3 - 7 = 0. \end{aligned} \tag{5.15}$$

The feasible set of this problem is convex, and the objective function is convex but non-smooth due to its inclusion of absolute values.

Construction of IVPs. We model the problem (5.15) by the ODE system (5.3) and set the time range as $[t_0, T] = [0, 10]$. We choose three initial points to study, namely $[0, 0, 0, 0]$, $[1, 0, -2, 3]$, and $[-1, 1, -1, 1]$, which result in three IVPs. Based on these three initial points, we construct each of the three proposed neural networks (5.7) as approximate state solutions to the IVPs.

Neural Network Solution. Figure 5.3 shows the solution process of our approach. Each horizontal row represents the evolution of a neural network trained by Algorithm 3. Each sub-figure shows $\hat{y}(t; w)$ for $t \in [0, 10]$, which represents the approximate state solution of the IVP at a given training iteration. Here, we must emphasize that $\hat{y}(t; w)$ is implemented by one neural network with four output units, and $\hat{y}(t; w) = (\hat{y}_1(t; w), \hat{y}_2(t; w), \hat{y}_3(t; w), \hat{y}_4(t; w))$. In particular, the end state has

$$\hat{y}(t = 10; w) = \left(\underbrace{\hat{y}_1(t = 10; w), \hat{y}_2(t = 10; w), \hat{y}_3(t = 10; w)}_{=\hat{x}}, \underbrace{\hat{y}_4(t = 10; w)}_{=\hat{u}} \right), \tag{5.16}$$

where $\hat{x}$ and $\hat{u}$ represent the predicted primal and dual solutions, respectively, of the problem (5.15).

Evolution of the Predictions. As discussed in Section 5.3.3, the end state prediction $\hat{y}(t = 10; w)$ is improved by training on the entire approximate state solution. At training iteration 0, the approximate state solution is far from the true state solution, resulting in a high OuC value of the endpoint prediction. After

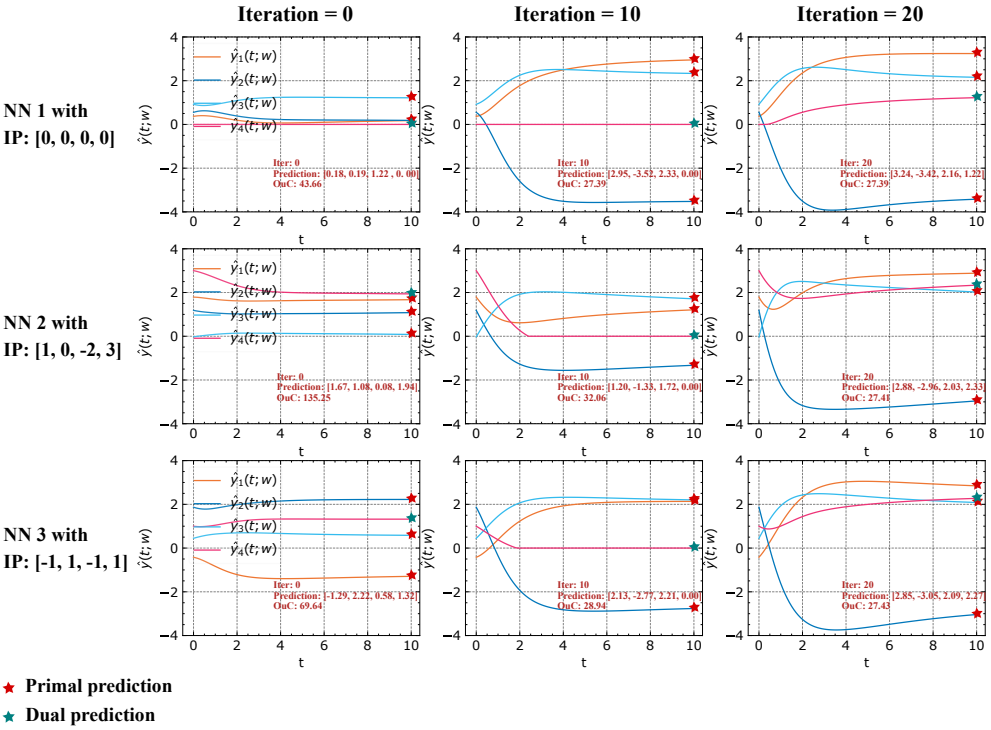


Figure 5.3 – Neural network solutions to problem (5.15). The three neural networks are initialized with three different initial points (IPs). Each row shows a neural network as a function at selected training iterations.

20 training iterations, the approximate state solution gets closer to the true state solution, and the endpoint prediction improves significantly. Notably, as these networks are constructed with different initial conditions, they have varying initial OuC values. Nonetheless, our proposed algorithm ensures that all networks converge to a prediction with an OuC value of less than 28 after 20 training iterations.

OuC Performance. Figure 5.4 presents a comparative analysis of the OuC drop rates for our proposed method and six traditional numerical integration methods over three different initial points. The following observations can be made :

— Our method outperforms traditional numerical integration methods in terms of OuC reduction, as evidenced by the lower OuC values achieved in fewer iterations. For example, for the initial point (IP) $[0, 0, 0, 0]$, our method reduces OuC from 43.66 to 28 in just five iterations, whereas the best-performing numerical integration methods, namely RK45, DOP853, and Radau, require 20 iterations to achieve comparable results. Similar results are observed for the other two initial points.

— The speed of convergence varies for different initial points, with our method

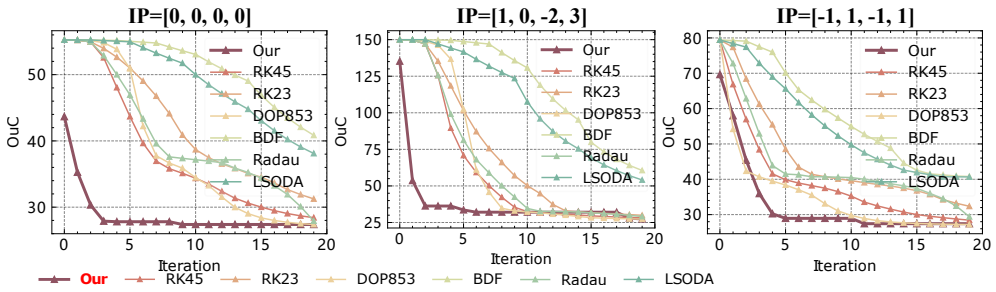


Figure 5.4 – Comparison of our proposed method with the numerical integration methods on the OuC metric. The OuC metric is defined in Equation (5.13).

showing greater robustness to different initial settings compared to numerical integration methods. While most numerical integration methods converge faster for the IP $[1, 0, -2, 3]$ and slower for $[0, 0, 0, 0]$, our method converges at approximately the same rate for both.

— The starting OuC values differ for various initial points, with the IP $[1, 0, -2, 3]$ having the highest starting OuC value of around 135, and $[0, 0, 0, 0]$ having the lowest starting OuC of approximately 43. However, the results demonstrate that the OuC values do not significantly affect the speed of convergence.

Why Our Approach has Better OuC than RK45. It is important to emphasize that our approach has no advantage over classical numerical integrators such as RK45 when it comes to solving for the full solution on the IVP, i.e. the function y. As shown in Figure 5.4, our method outperforms these numerical integrators in the OuC metric because our approach focuses on improving the end point $\hat{y}(t = T; w)$, rather than the entirety of the function $\hat{y}$. It is also worth noting that the OuC performance metric only considers the endpoint $\hat{y}(t = T; w)$, not the entire function $\hat{y}$. The methodology we propose, described in sections 5.3, is deliberately designed with this specific goal in mind.

Computational Time Performance. Figure 5.5 shows the time needed by different solution methods to obtain an acceptable solution (i.e., OuC ≤ 28) to Problem (5.15). Our proposed method outperforms all considered numerical integration methods in terms of computational efficiency. The most efficient numerical integration method, BDF, still requires 10 times more computational time than our method to obtain a satisfactory solution. Moreover, the computational time of our method is less affected by different IP settings, requiring about 1.78 seconds for all three initial points. In contrast, some numerical integration methods, such as Radau and LSODA, show significant variations in computation time at different initial points.

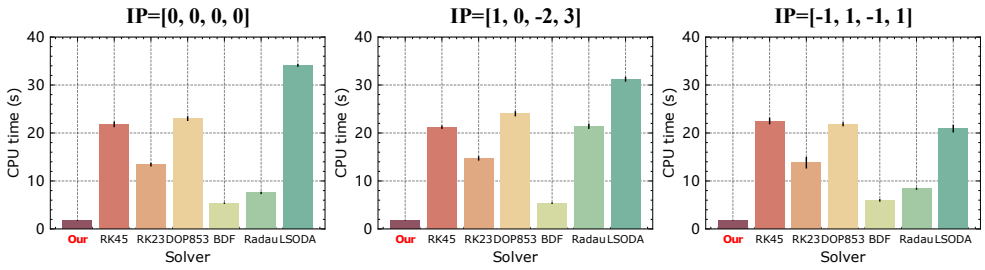


Figure 5.5 – Comparison of our method with the numerical integration methods in terms of computational efficiency. Time is measured in seconds.

5.4.2 . Comparisons with PINN

In this subsection, we perform an ablation study comparing the proposed method with PINN [26] and Lagaris method [193] to validate the effectiveness of our method.

Example 2 :

$$\min_x f(x) = |2.3x_1 + x_3 - 3.5| + |x_2 + 2x_3 - 1.8| + |1.3x_1 + x_2 + x_3 + 3|$$

s.t.

$$\begin{aligned} g(x) &= x_1^2 - x_2 + x_3 + 3 \leq 0 \\ h_1(x) &= x_1 + x_2 + x_3 = 0, \\ h_2(x) &= 2x_2 + x_3 = 2. \end{aligned} \tag{5.17}$$

We aim to solve the NCOP (5.17). The problem has a convex feasible set and a convex but non-smooth objective function due to its inclusion of absolute values.

Construction of IVPs. We model problem (5.17) as an ODE system (5.3) and set the time range as $[t_0, T] = [0, 10]$. To construct three IVPs, we choose three initial points, namely $[1, -1, 0, 3]$, $[2, 3, -2, 1]$, and $[2, -2, 1, -2]$. Based on these initial points, we instantiate three proposed neural networks (5.7) as approximate state solutions.

Experimental Setup. We compare our proposed approach with two methods : vanilla PINN and PINN with Lagaris construction method (PINN+Lagaris). Our approach can be regarded as the PINN+Lagaris method enhanced by the ORR mechanism (PINN+Lagaris+ORR). The hyperparameters and training details are the same as those in Section 5.4.1.

Table 5.1 shows the performance of the three methods over the first one hundred iterations, while Figure 5.6 shows their convergence behavior over the first one thousand iterations. The results reveal the following key observations :

— Our proposed approach yields an excellent predicted solution within the first 20 training iterations, consistent with the results presented in Section

Table 5.1 – Comparison of PINN, PINN+Lagaris, and PINN+Lagaris+ORR (Our approach) for three different initial points with predicted solutions and corresponding OuC values. inf indicates that the predicted solution is not in the feasible set.

Initial point : [1, -1, 0, 3]						
Iteration	PINN		PINN+Lagaris		PINN+Lagaris+ORR	
	Predicted solution	OuC	Predicted solution	OuC	Predicted solution	OuC
0	[-0.64 1.36 -0.71 0.47]	inf	[-1.00 1.00 -0.00 3.09]	inf	[-1.00 1.00 -0.00 3.09]	**inf**
5	[-0.40 1.60 -1.19 1.10]	inf	[-1.42 0.58 0.84 7.02]	inf	[0.05 2.05 -2.09 4.85]	**12.44**
10	[-0.31 1.69 -1.37 2.17]	inf	[0.16 2.16 -2.31 9.50]	12.77	[-0.18 1.82 -1.64 9.25]	**11.76**
20	[0.81 2.81 -3.63 3.53]	14.74	[1.43 3.43 -4.86 6.63]	16.60	[-0.18 1.82 -1.64 9.25]	**11.76**
40	[0.72 2.72 -3.44 5.03]	14.45	[0.29 2.29 -2.58 1.91]	13.17	[-0.18 1.82 -1.64 9.25]	**11.76**
60	[0.74 2.74 -3.48 4.80]	14.52	[1.12 3.12 -4.25 2.18]	15.67	[-0.18 1.82 -1.64 9.25]	**11.76**
80	[0.84 2.84 -3.68 4.35]	14.82	[0.45 2.45 -2.90 1.48]	13.65	[-0.18 1.82 -1.64 9.25]	**11.76**
100	[0.82 2.82 -3.64 3.96]	14.76	[0.19 2.19 -2.39 0.72]	12.88	[-0.18 1.82 -1.64 9.25]	**11.76**

Initial point : [2, 3, -2, 1]						
Iteration	PINN		PINN+Lagaris		PINN+Lagaris+ORR	
	Predicted solution	OuC	Predicted solution	OuC	Predicted solution	OuC
0	[-1.00 1.00 0.00 0.00]	inf	[0.41 2.41 -2.82 0.23]	13.54	[0.41 2.41 -2.82 0.23]	**13.54**
5	[-1.07 0.93 0.13 0.64]	inf	[-0.77 1.23 -0.46 1.52]	inf	[-0.29 1.71 -1.41 1.77]	**11.42**
10	[-1.42 0.58 0.85 2.04]	inf	[0.04 2.04 -2.08 1.39]	12.42	[-0.29 1.71 -1.41 1.77]	**11.42**
20	[-1.44 0.56 0.87 4.04]	inf	[0.85 2.85 -3.69 2.26]	14.84	[-0.29 1.71 -1.41 1.77]	**11.42**
40	[0.89 2.89 -3.77 3.24]	14.96	[0.26 2.26 -2.53 1.21]	13.09	[-0.29 1.71 -1.41 1.77]	**11.42**
60	[0.56 2.56 -3.11 1.69]	13.97	[-0.29 1.71 -1.43 0.25]	11.44	[-0.29 1.71 -1.41 1.77]	**11.42**
80	[0.44 2.44 -2.88 1.79]	13.63	[-0.62 1.38 -0.75 0.71]	inf	[-0.30 1.70 -1.40 1.19]	**11.40**
100	[0.01 2.01 -2.02 1.05]	12.33	[-0.09 1.91 -1.83 0.88]	12.04	[-0.30 1.70 -1.40 1.19]	**11.40**

Initial point : [2, -2, 1, -2]						
Iteration	PINN		PINN+Lagaris		PINN+Lagaris+ORR	
	Predicted solution	OuC	Predicted solution	OuC	Predicted solution	OuC
0	[-0.65 1.35 -0.7 0.]	inf	[-1.05 0.95 0.1 0.]	inf	[-1.05 0.95 0.1 0.]	**inf**
5	[-0.68 1.32 -0.65 0.44]	inf	[-2.78 -0.78 3.56 1.67]	inf	[-0.10 1.90 -1.80 6.38]	**12.01**
10	[-1.53 0.47 1.05 2.08]	inf	[-1.21 0.79 0.41 5.23]	inf	[-0.10 1.90 -1.80 6.38]	**12.01**
20	[-1.66 0.34 1.32 5.54]	inf	[2.03 4.03 -6.06 9.]	18.39	[-0.23 1.77 -1.54 6.37]	**11.62**
40	[0.42 2.42 -2.84 6.84]	13.56	[0.53 2.53 -3.05 3.46]	13.88	[-0.23 1.77 -1.54 6.37]	**11.62**
60	[1.24 3.24 -4.47 6.27]	16.01	[0.67 2.67 -3.34 5.02]	14.31	[-0.23 1.77 -1.54 6.37]	**11.62**
80	[1.3 3.3 -4.6 5.42]	16.2	[0.63 2.63 -3.25 3.59]	14.18	[-0.23 1.77 -1.54 6.37]	**11.62**
100	[1.27 3.27 -4.54 4.92]	16.11	[0.59 2.59 -3.19 3.69]	14.08	[-0.23 1.77 -1.54 6.37]	**11.62**

5.4.1. In contrast, even after 1000 iterations, neither the PINN nor the PINN+Lagaris methods achieves a predicted solution that compares favorably to that of our approach.

— Our method has a higher probability of obtaining feasible solutions. As shown in Table 5.1, the PINN method returns 'inf' 11 times for the three IP configurations, while the Lagaris method reduces this occurrence to 7. In contrast, our method returns 'inf' only twice, both times in the first round, indicating that it can reach a feasible solution more quickly.

— Neither the PINN method nor the PINN+Lagaris method maintains an optimal solution during the optimization process. As shown in Table 5.1, the

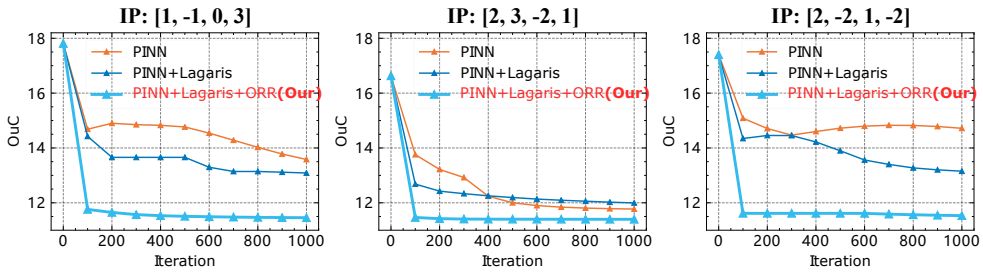


Figure 5.6 – Comparison of our proposed method with PINN and the PINN+Lagaris method on the OuC metric. The experiment is conducted on Problem (5.17)

PINN method and the PINN+Lagaris method achieve good OuC values of 12.33 and 12.04, respectively, for the IP $[2, 3, -2, 1]$ at the 100th iteration. However, neither method maintains this level of performance, and their OuC values increase in subsequent iterations.

— The Lagaris method can improve the performance of vanilla PINN, as shown in Figure 5.6. However, this improvement is not substantial and varies depending on the IP configuration. For example, the improvement is significant for the first and third initial points but negligible for the second IP.

5.4.3 . Hyperparameter Study

In this subsection, we perform a hyperparameter study on the following NCOP.

Example 3 :

$$\begin{aligned} &\min_x f(x) = \|Cx - d\|_1 \\ &\text{s.t.} \\ &\quad g_1(x) = x_1^2 - x_2 + x_3 + x_5 - x_8 - 10 \leq 0, \\ &\quad g_2(x) = |x1 - x3 + x4 + x7| - 4.8 \leq 0, \\ &\quad h(x) = x_1 + x_3 + x_5 + x_7 - 1 = 0, \end{aligned} \tag{5.18}$$

where $\|\cdot\|_1$ denotes the L^1 norm, and

$$C = \begin{pmatrix} 1 & 4 & 2 & 2 & 1.3 & 4 & 2 & 1 \\ 2.8 & 2 & 1.6 & 3.2 & 0 & 2 & 1 & 1 \\ 1 & 4 & 2.3 & 2 & 2.5 & 0 & 5 & 1 \\ 1 & 1 & 1 & 3.1 & 2.3 & 0 & 0.8 & 1 \end{pmatrix}, \quad d = \begin{pmatrix} 1.5 \\ -3.8 \\ 6.2 \\ 7.5 \end{pmatrix}. \tag{5.19}$$

Example 3 involves a nonsmooth objective function and a nonsmooth inequality constraint $g_2(x)$.

To set up the algorithm, we choose the IP as an all-ones vector and the time range as $[0, 10]$. In the following, we discuss the computational performance for different neural network sizes and different learning rates.

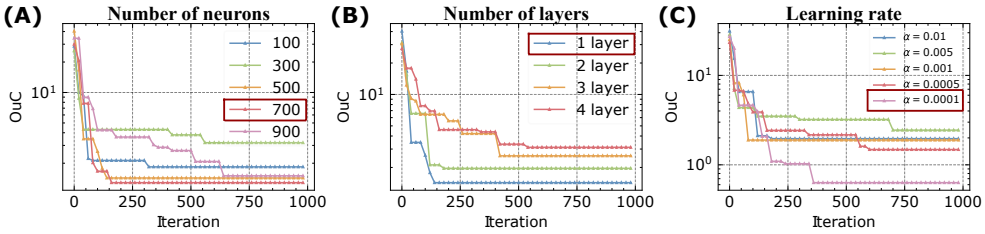


Figure 5.7 – Hyperparameter study. (A) : OuC performance on different numbers of neurons in a single layer neural network with a learning rate of 0.01. (B) : OuC performance of different layers in a multilayer neural network with 500 neurons per layer and a learning rate of 0.01. (C) : OuC performance at different learning rates in a two-layer neural network with 500 neurons per layer.

Model Size. In Figures 5.7 (A) and (B), we investigate the computational performance of neural networks with various widths and depths, and the optimal result is obtained with a 700-neuron-wide, single-layer structure. In (A), with a maximum of 1000 training iterations, networks with fewer neurons (such as 100, 300, and 500) underperform due to their model capacity, limiting further improvements in OuC even with more training. Conversely, a network with more neurons (such as 900) shows underperformance, likely due to insufficient model training, and its OuC would potentially improve with additional training. In (B), a single layer neural network is shown to outperform other configurations. Taken together, these results underscore the need to find the most appropriate network structure for a given NCOP problem. An overly complex network would require an excessive amount of computing resources for optimization that may not be necessary. Conversely, a network that is too simple would not find the appropriate solution, regardless of the amount of training. Thus, the size of the neural network should be determined by factors related to the NCOP being solved. These include the number of decision variables, the constraints, and the complexity of both the objective and constraint functions.

Learning Rate. Figure 5.7(C) shows the performance of neural networks trained with different learning rates. At iteration 1000, the optimal performance is observed at the learning rate of $\alpha = 0.0001$. It is important to note that if we zoom into the first 100 iterations, a larger learning rate $\alpha = 0.001$ is more effective. This suggests that the choice of learning rate should depend on the actual preferences of the user. The advantage of a large learning rate is that it can find a better prediction for the NCOP in a short time, while the disadvantage is that it performs poorly in the long run. In contrast, a small learning rate finds better solutions in the long run. Overall, the choice of learning rate should be determined by the user's specific requirements for speed and accuracy.

5.4.4 . L^1 Norm Minimization Problem

Consider the following NCOP problem :

Example 4 :

$$\min_x f(x) = \|x\|_1$$

s.t.

$$g_i(x) = x^2_{10*(i-1)+1} + x^2_{10*(i-1)+2} + \cdots + x^2_{10*(i-1)+10} - 20 \leq 0, \quad \text{for } i = 1, 2, \ldots, 300, \tag{5.20}$$

$$h(x) = Ax - b = 0,$$

where $x \in \mathbb{R}^{3000}$, $A \in \mathbb{R}^{1\times 3000}$, with the first half entries of A being 1 and the rest 3, and $b = 16$.

Table 5.2 – Description of IPs and their OuC performance at different algorithm iterations. Columns 2 to 6 describe the IPs and the their initial information, and columns 7 to 10 describe their OuC values at different training iterations.

	Description	Initial $f(x)$	Initial $\max_{i=1,2,\ldots,300}(g_i(x))$	Initial $h(x)$	Initial OuC	OuC at iteration 100	OuC at iteration 1000	OuC at iteration 3000	OuC at iteration 10000
IP1	All-ones vector : $(1, 1, \ldots, 1)$	3000.0	-10.0	5984.	inf	133.92	7.08	6.31	5.81
IP2	All-threes vector : $(3, 3, \ldots, 3)$	9000.0	70	17984	inf	202	118	25	6.00
IP3	All-negative ones vector : $(-1, -1, \ldots, -1)$	3000.0	-10.0	-6016	inf	166.10	8.65	6.23	6.23
IP4	Alternating sequence of 2 and -2 : $(2, -2, \ldots, 2, -2)$	6000.0	20	-16	inf	643	16.62	16.62	16.62
IP5	First half entries are 1 and the rest are 3 : $(1, 1, \ldots, 3)$	5700.0	70	14084	inf	946	380	344	95

IPs Description. We examine five different IPs, which are listed in Table 5.2. The IPs are used to configure Algorithm 3 to solve problem (5.20). As shown in the table, all the five IPs have large initial objective values, with IP2, IP4, and IP5 failing to satisfy the inequality constraint, i.e., $\max_{i=1,2,\ldots,300}(g_i(x)) \geq 0$, and all IPs failing to meet the equality constraint, i.e., $h(x) \neq 0$. These observations indicate that the IPs initially do not solve Example 4 well and are far from the optimal solution.

Based on the five IPs, the OuC values at different algorithm iterations are shown in Table 5.2 and Figure 5.8. We observe that :

— As shown in Figure 5.8-(A), our algorithm quickly finds a feasible solution that satisfies both the inequality and equality constraints. Moreover, once the first feasible solution is found, the subsequent solutions given by the algorithm are all within the feasible set.

— The final solutions given by the algorithm are acceptable. After going through the entire solution process, the OuC values associated with the IPs decrease significantly. For IP1, the OuC decreases from 3000 to 5.81 ($100\% \to 0.2\%$), and similar results can be found for other IPs. Given the fact that problem

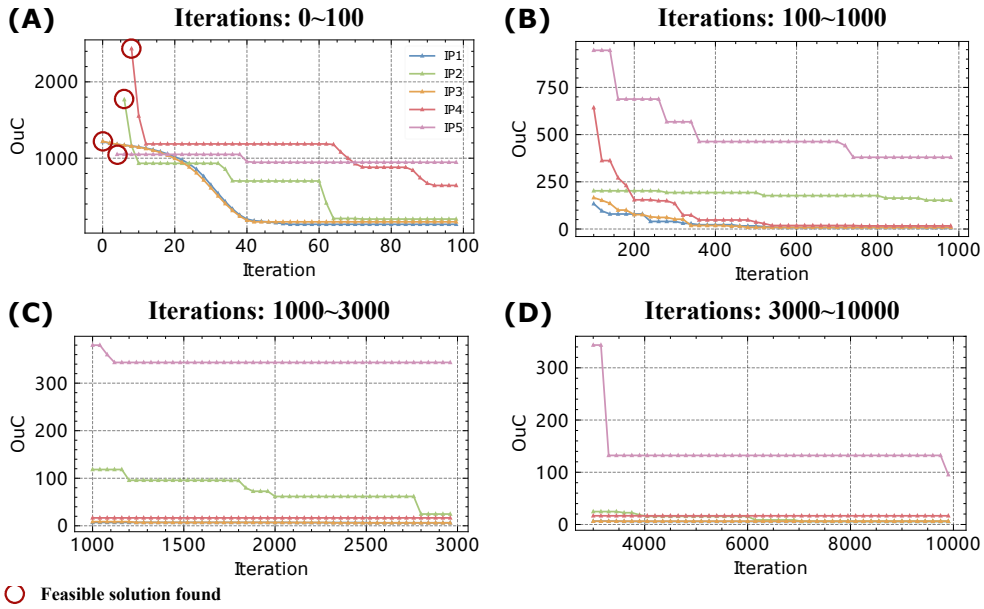


Figure 5.8 – OuC performance with various IP configurations. The detailed descriptions for the five IPs are given in Table 5.2. In (A), the red circle indicates the first time a feasible solution is found. (A), (B), (C), and (D) show the results of the algorithm iterations 0∼100, 100∼1000, 1000∼3000, and 3000∼10000, respectively.

(5.20) has a known lower bound 0 for the objective value. This indicates that the final solutions produced by the proposed algorithm are already very close to the optimal solution.

— The OuC decreasing speed or convergence rate varies under different IP configuration. The convergence rate and final result of IP1∼IP4 are significantly better than those of IP5. This may be because IP5 is the farthest from the optimal solution, and thus requires a larger time range and more model training. Nevertheless, the proposed algorithm still significantly improves the OuC performance of IP5 ($100\% \rightarrow 1.6\%$) with the given experimental setup.

— Most of the decrease in OuC values occurs in the first 1000 iterations. In particular, IP1, IP2, and IP4 reduce the OuC values to about 10 within only 1000 iterations, which is already very close to the final result, demonstrating the efficiency of the algorithm.

5.4.5 . NCOP Problem Set

Problem Set Description. We construct a set of NCOPs based on Example 4 (5.20), where each NCOP problem takes the following form :

$$\min_x f(x) = \|x\|_1$$

s.t.

$$\begin{aligned} g_i(x) &= x^2_{10*(i-1)+1} + x^2_{10*(i-1)+2} + \cdots + x^2_{10*(i-1)+10} - c^{(k)} \leq 0, \quad i = 1, 2, \ldots, 100 \\ h(x) &= A^{(k)}x - b^{(k)} = 0, \end{aligned} \tag{5.21}$$

where $x \in \mathbb{R}^{1000}$, $A^{(k)} \in \mathbb{R}^{1000}$, $b^{(k)} \in \mathbb{R}$, and $c^{(k)} \in \mathbb{R}$. $A^{(k)}$, $b^{(k)}$, $c^{(k)}$ are sampled from uniform distributions $U(1,5)$, $U(10,20)$, $U(20,30)$, respectively. We randomly generate 100 different problem data $\{(A^{(k)}, b^{(k)}, c^{(k)})\}$ to form 100 different NCOPs. These problem datasets $\{(A^{(k)}, b^{(k)}, c^{(k)})\}_{k=1}^{100}$ can be accessed from the link[1]. Consistent with previous experimental subsections, we set the time range for all NCOPs to $[0, 10]$ and the IP y_0 as an all-one vector.

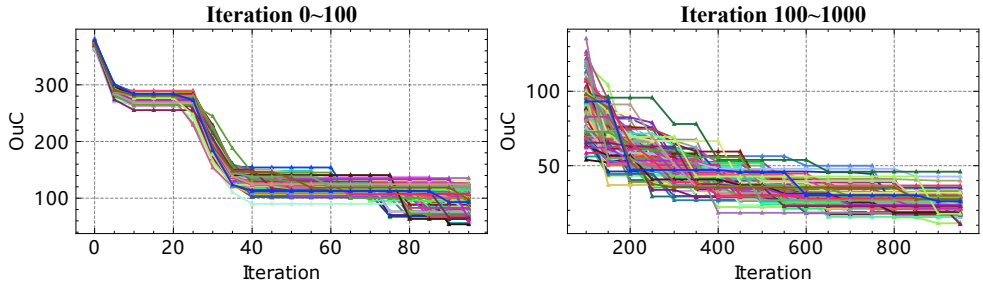


Figure 5.9 – OuC values of the 100 NCOPs at different training iterations. Left and right show the results of iterations 0∼100 and 100∼1000, respectively.

Figure 5.9 shows the resolution of these 100 NCOPs using our neural network approach, and Table 5.3 shows the statistical information for these OuC results at different iterations. A clear trend can be seen is that all the OuC values decrease as training progresses. Starting from a mean of 372.79, the OuC value drops to 24.00 after 1000 iterations (a reduction from 100% to 6.4%). This trend of decreasing OuC values is not only observed at the mean, but also consistently observed at the 25%, 50%, 75%, and 90% quantiles. Of particular note is the impressive magnitude of this reduction. The significant reduction in OuC values indicates that our method efficiently navigates the solution space, making steady progress towards optimality. This highlights the potential of our approach for tackling a wide variety of NCOPs. In summary, these results provide a strong indication of the effectiveness of our

1. https://drive.google.com/drive/folders/1D_3HP-fBp9tew4IgDroIQtgVb8vU-vGO?usp=drive_link

Table 5.3 – Statistical data of OuC values for 100 NCOPs at different training iterations. The table shows the mean, standard deviation (STD), and values at the 25%, 50% (median), 75%, and 90% quantiles of the OuC distribution at each iteration.

Iteration	Mean	STD	25% quantile	50% quantile	75% quantile	90% quantile
0	372.79	3.93	370.11	372.46	375.05	378.15
20	277.87	6.21	274.28	278.69	282.21	284.90
40	130.84	14.12	121.48	130.32	142.55	148.95
60	120.27	9.91	114.34	120.55	126.57	129.90
80	108.60	18.59	102.61	113.15	121.67	127.45
100	89.10	18.81	72.78	88.25	101.73	117.89
300	49.08	9.55	43.14	48.21	53.61	61.63
500	37.23	7.61	32.01	36.37	42.91	46.19
700	30.13	6.95	25.86	29.85	34.31	39.05
999	24.00	6.17	20.73	23.64	27.64	30.35

proposed method for solving NCOPs, demonstrating its robustness and efficiency over a wide range of problem datasets.

5.4.6 . Comparsions with Optimization Solvers

Problem	Our Approach		CVXPY-ECOS		CVXPY-SCS	
	OuC	CPU time	OuC	CPU time	OuC	CPU time
Example 1	27.43	1.78	27.12	0.02	27.12	0.03
Example 2	11.40	2.13	11.39	0.02	11.39	0.03
Example 3	0.08	12.23	0.00	0.02	0.00	0.03
Example 4	5.34	180.32	5.33	1.06	5.33	2.31

Table 5.4 – Performance comparison with the optimization solvers.

Experimental Setup. In this section, we compare the proposed approach with the optimization solvers. The solvers involved in this comparison are ECOS [225] and SCS [226], which can be called directly from the CVXPY [227] library. The target problems considered are the four NCOP examples presented in Sections 5.4.1 to 5.4.4. The accuracy and efficiency results are reported in Table 5.4.

Optimization Solver is Still the Best Choice. According to the results shown in the table, it is easy to conclude that the optimization solvers are much better than our approach at this stage. Therefore, when considering solving an optimization problem, various SOTA optimization solvers remain the best choice, offering more robustness and higher efficiency. It is important to emphasise that this chapter does not aim to outperform classical optimisation solvers, which have been developed over many years and are well established. Instead, the aim of this chapter is to present a novel way of solving these classical problems and to inspire future studies. In the experiments, we provide some preliminary guidelines. Sections 5.4.1 and 5.4.2 have shown that the proposed algorithm is not a naive combination

of neurodynamic optimization and PINN. Sections 5.4.3 to 5.4.5 have discussed some important hyperparameter settings in the proposed algorithm.

A Similar Situation with PINNs. In fact, there is a lot of experimental evidence that PINNs cannot outperform classical finite element methods in solving PDEs [228]. However, this does not diminish the innovation and significance of PINNs. For example, recent developments such as Fourier Neural Operators [229] and DeepONets [130] have demonstrated the potential for solving parameterized PDEs. In cases where the problem parameters change, finite element methods require starting from scratch to solve a new PDE, while neural networks can rapidly approximate by taking the parameters as additional inputs. As the unstoppable development of artificial intelligence continues, we will see the evolution and development of these emerging, neural network-based algorithms.

5.5 . Conclusion

In this study, we present a deep learning-based methodology for solving NCOPs. The proposed methodology is a fruitful fusion of neurodynamic optimization and PINNs. Methodologically, we have extended the PINN approach to accommodate neurodynamic optimization. In addition, we have developed a novel training algorithm that increases computational efficiency by exploiting the problem structure of NCOPs. Experimental results have demonstrated the effectiveness of the proposed method on a number of NCOPs. The computational performance can be further improved by tuning the hyperparameters and refining the training details.

In addition, our results have identified several avenues for future research. Specifically, we recommend investigating better methods for selecting initial points and time ranges, exploring different network architectures and advanced neurodynamic optimization techniques. Further development in these areas will undoubtedly improve the effectiveness and robustness of our approach, making it an important tool for addressing NCOPs in diverse applications.

6 - Solving Chance-Constrained Games at Various Confidence Levels with CCGnet

This chapter aims to solve the Nash equilibrium in chance-constrained games (CCGs), which are characterized by Singh et al. [38]. Specifically, we consider solving CCGs at different confidence levels. In the traditional approach, the CCGs at different confidence levels are treated as independent tasks and solved individually using standard optimization solvers. However, such an approach is obviously inefficient. In this chapter, we introduce CCGnet, a neural network model capable of efficiently solving CCGs at different confidence levels. Our experiments demonstrate the exceptional performance of CCGnet.

Similar to Chapters 4 and 5, the method proposed in this chapter is also based on neurodynamic optimization and Physics-Informed Neural Networks (PINNs). The key distinction between this chapter and the preceding two lies in the network architecture. While previous chapters employ a single neural network to solve one optimization problem, this chapter utilizes a single network to solve multiple optimization problems. This eliminates the need for training multiple neural networks, thereby significantly improving computational efficiency. The methodology in this chapter is inspired by the concept of ODE solution bundles [230], where a single neural network is used to solve multiple systems of ordinary differential equations. Here, we effectively adapt this framework for CCGs.

This chapter corresponds to the reference [148].

6.1 . Introduction

Game theory analyzes the strategic interactions between rational individuals in situations involving conflict or cooperation [9]. A Nash equilibrium is a state in which no player can improve his payoff by changing his strategy unilaterally. von Neumann [157] demonstrated the existence of a saddle point for two-person zero-sum games through the minimax theorem. Nash [10] showed that an equilibrium also exists in multi-player non-zero-sum games commonly called Nash equilibrium.

The games mentioned above are all deterministic. However, many real-world situations involve games where the player's payoff function or strategy set contains randomness. Such a game with randomness is called a stochastic Nash game. Ravat & Shanbhag [231] characterized the solution set for various types of stochastic Nash games. If players are assumed to be risk-neutral, the expected payoff criterion can be used to handle the randomness in the game [231, 232].

When considering a risk-averse case, the randomness in a game can be addressed through the chance constraint programming approach, known as a chance constraint game (CCG) [38, 40, 233]. In a CCG, players are guaranteed to receive

payoffs with a certain confidence level. For example, Singh & Lisser [40] studied a two-person zero-sum game with a random strategy set and characterized the saddle point as a primal-dual pair of second-order cone programs when the random variable follows an elliptical distribution.

Neurodynamic optimization uses ordinary differential equation (ODE) systems to solve optimization problems. Hopfield & Tank [234] proposed Hopfield networks for solving linear programming problems. Kennedy & Chua [46] developed an approach based on the penalty function method for solving nonlinear programming problems. However, this approach involves a penalty parameter and the optimal solution can only be obtained when the penalty term tends to infinity.. Since then, neurodynamic optimization has been well established for solving various optimization problems, such as convex optimization problems [32, 235], pseudoconvex problems [222, 192], distributed optimization problems [236, 237, 238, 239], and Nash equilibrium computation [240, 241, 242].

Deep learning is a type of machine learning that involves the use of deep neural networks, which consist of multiple layers of interconnected nodes, to identify complex patterns and relationships in data. It has been applied successfully to various fields, including computer vision [243], natural language processing [244], bioinformatics [245], game theory [179, 180], and operation research [246, 247, 248]. However, deep learning also has limitations, and researchers are working to improve its performance and address challenges such as bias and interpretability.

Approximation methods using deep learning for differential equations were first studied in the 1990s. Dissanayake & Phan-Thien [249] used a neural network as an approximate solution to a differential equation, where the neural network was trained to satisfy the given differential equation and boundary conditions. Lagaris et al. [193] proposed a neural network model that can satisfy boundary conditions by construction. They discussed the use of this method on ODE and PDE problems, respectively. This method was extended to irregular boundaries [194]. In recent years, with the rapid development of deep learning, these methods have been further extended for solving high-dimensional PDEs [122, 195]. Flamant et al. took the parameters of the ODE system as the input variables of the neural network, allowing the neural network to be used as the solution for a group of ODE systems [230]. The rapid development of this research direction has been made possible by automatic differentiation tools, which facilitate the computation of derivatives [97, 181].

The main contributions of this chapter can be summarized as follows :

— Our proposed CCGnet is able to receive instances of different parameters and solve them directly without any iterative process. In terms of computational time, CCGnet outperforms traditional solution approaches. This advantage becomes even more significant when solving multiple instances. For example, for 10,000 instances, the CCGnet model solves within 1.53 ms CPU time, while traditional numerical solvers take at least 22,500 ms CPU time.

— CCGnet transforms a CCG problem into a neural network training problem. Since our CCGnet model is based entirely on deep learning infrastructure, we can solve CCG without using any standard numerical solvers.

The remaining sections of this chapter are organized as follows : Section 6.2 presents the background knowledge needed for understanding the chapter, including the introduction of CCG and the neurodynamic optimization approach. Section 6.3 presents our proposed CCGnet approach. Section 6.4 gives numerical results of using CCGnet for solving CCG. Section 6.5 summarizes this chapter and gives future directions.

Table 6.1 – Notation list of Chapter 6

Notation	Definition
CCG	Chance-constrained game
NPE	Nonlinear projection equation
IVP	Initial value problem
CCG^θ, NPE^θ and IVP^θ	A CCG, NPE, IVP with parameter θ
$n \in \mathbb{N}$	The number of players
$x \in \mathbb{R}^n$	A strategy profile of a stochastic cournot competition
$y \in \mathbb{R}^{2n}$	Variable of a NPE
$\Phi(z) : \mathbb{R}^{2n} \to \mathbb{R}^{2n}$	An ODE system, $\frac{dz}{dt} = \Phi(z)$
$z(t) : \mathbb{R} \to \mathbb{R}^{2n}$	A state solution
$\hat{z}(t, \theta; \mathbf{w})$	A CCGnet model with model weight $\mathbf{w}$
$(t_0, z_0) \in \mathbb{R}^{2n+1}$	An initial point
$[t_0, T] \subset \mathbb{R}$	A time range
$\theta \in \Theta$	CCG parameter

The notations used in this chapter are listed in Table 6.1.

6.2 . Preliminaries

Section 6.2.1 introduces the CCG, including the definition and the existence theorem of a Nash equilibrium. Section 6.2.2 introduces an example of CCG, namely stochastic cournot games among electrical firms, and shows how to reformulate such a CCG as a nonlinear projection equation (NPE). Section 6.2.3 introduces the neurodynamic approach for solving the NPE.

6.2.1 . Chance-Constrained Game

Let an n-player game with continuous action and random payoffs be defined as a tuple $\left(I, \left(X^i\right)_{i \in I}, \left(r^i\right)_{i \in I}\right)$, where

— $I = \{1, 2, \ldots, n\}$ is a set of players.

— For each $i \in I$, let A_i be a finite action set of player i together with its generic element a_i. A vector $a = (a_1, a_2, \cdots, a_n)$ denotes an action profile

of the game. Let $A = \times_{i=1}^{n} A_i$ be the set of all action profiles of the game. Denote, $A_{-i} = \times_{j=1;j\neq i}^{n} A_j$, and $a_{-i} \in A_{-i}$ is a vector of actions a_j, $j \neq i$. $x^i \in \mathbb{R}^{A_i}$ is a strategy of player i. $x \in \mathbb{R}^A$ is a strategy profile. $x^{-i} \in \mathbb{R}^{A_{-i}}$ is a strategy profile without x^i. X^i, X^{-i} and X are feasible set for x^i, x^{-i} and x, respectively.

— Let $(\Omega, \mathcal{F}, P)$ be a probability space. $\xi^i : \Omega \to \mathbb{R}^{l_i}$ is a random vector, and $f^i : X \to \mathbb{R}^{l_i}$ is a function determining player i's payoff. Consider a strategy profile $x \in X$ and an event $\omega \in \Omega$, the payoff of player i is

$$r^i(x, \omega) = f^i(x) \cdot \xi^i(\omega). \tag{6.1}$$

The CCG defines the payoff function of player i as

$$u_i^{\alpha_i}(x) = \sup \left\{ \gamma \mid P\left(\left\{ \omega \mid r^i(x, \omega) \geq \gamma \right\} \right) \geq \alpha_i \right\}, \tag{6.2}$$

where $\alpha_i \in [0, 1]$ is a confidence level of player i, and $\alpha = (\alpha_i)_{i \in I} \in [0, 1]^n$. A strategy profile x^* is a Nash equilibrium, if the following holds

$$u_i^{\alpha_i}\left(x^{i*}, x^{-i*}\right) \geq u_i^{\alpha_i}\left(x^i, x^{-i*}\right), \quad \forall x^i \in X^i. \tag{6.3}$$

We consider the case where each random vector ξ^i, $\forall i \in I$ follows an elliptically symmetric distribution, i.e., $\mathrm{Ellip}\left(\mu_i, \Sigma_i, \varphi_i\right)$. μ_i is a location parameter. Σ_i is a positive definite matrix and φ_i is a characteristic generator function. Then, the payoff function of player i is

$$u_i^{\alpha_i}(x) = \mu_i^T f^i(x) + \left\| \Sigma_i^{1/2} f^i(x) \right\| \phi_{Z_i}^{-1}\left(1 - \alpha_i\right), \tag{6.4}$$

where $\phi_{Z_i}^{-1}(\cdot)$ is the quantile function of the distribution $\mathrm{Ellip}\left(\mu_i, \Sigma_i, \varphi_i\right)$.

Assumption 6.1 *The following conditions hold for each player i.*

— *$X^i \subset \mathbb{R}^{j_i}$ is a non-empty, convex and compact set.*

— *$f_k^i : \mathbb{R}^j \to \mathbb{R}$ is a continuous function, for all $k = 1, 2, \ldots, l_i$.*

— *For a given $x^{-i} \in X^{-i}$, $f_k^i\left(\cdot, x^{-i}\right)$ is an affine function, for all $k = 1, 2, \ldots, l_i$. Or, for a given $x^{-i} \in X^{-i}$, $f_k^i\left(\cdot, x^{-i}\right)$ is an non-positive and concave function, for all $k = 1, 2, \ldots, l_i$, and all elements of μ_i and Σ_i are non-negative.*

Theorem 6.1 (Singh & Lisser [39], Theorem 1) *Consider a chance constrained game $\left(I, \left(X^i\right)_{i \in I}, \left(r^i\right)_{i \in I}\right)$. For each player $i \in I$, the random vector ξ^i follows an elliptical distribution $\mathrm{Ellip}\left(\mu_i, \Sigma_i, \varphi_i\right)$. Let Assumption 6.1 holds. There exists a Nash equilibrium for this chance constrained game with any $\alpha \in (0.5, 1]^n$.*

6.2.2 . Stochastic Cournot Competition

We now consider an example of CCG, namely stochastic cournot competitions among electricity firms. We show how this CCG can be reformulated as an nonlinear projection equation.

Consider an electricity market with n competing firms. $x^i \in X^i \subset \mathbb{R}_+$ denote an amount of electricity generated by firm i. Each firm i has a finite capacity C^i, i.e., $X^i = \left[0, C^i\right]$. $x = \left(x^1, x^2, \ldots, x^n\right) \in \mathbb{R}^n$ denote a strategy profile. Let $(\Omega, \mathcal{F}, P)$ be a probability space. The unit market price is determined by x and an event ω,

$$P(x, \omega) = a - b \cdot \sum_{i=1}^{n} x^i + \zeta(\omega), \tag{6.5}$$

where $\zeta : \Omega \to \mathbb{R}$ is a random variable, and $a \in \mathbb{R}$ and $b \in \mathbb{R}_+$ are two market price factors.

The payoff function of firm i is

$$r^i(x, \omega) = x^i \cdot P(x, \omega) - c_i\left(x^i\right), \tag{6.6}$$

where $c_i\left(x^i\right)$ is the cost of firm i to produce x^i amount of electricity, and $c_i(\cdot)$ is assumed to be differentiable and convex.

The chance-constraint payoff function for player i with confidence level α_i is defined as

$$u_i^{\alpha_i}(x) = \sup\left\{\gamma \mid P\left(\left\{\omega \mid x^i\left(a - b \cdot \sum_{j=1}^{n} x^j\right) + x^i \cdot \zeta(\omega) - c_i\left(x^i\right) \geq \gamma\right\}\right) \geq \alpha_i\right\}. \tag{6.7}$$

If $x^i > 0, \forall i \in I$, we have

$$\begin{aligned} u_i^{\alpha_i}(x) &= \sup\left\{\gamma \mid P\left(\left\{\omega \mid \zeta(\omega) \leq \frac{\gamma - x^i\left(a - b \cdot \sum_{j=1}^{n} x^j\right) + c_i\left(x^i\right)}{x^i}\right\}\right) \leq 1 - \alpha_i\right\} \\ &= \sup\left\{\gamma \mid \gamma \leq x^i\left(a - b \cdot \sum_{j=1}^{n} x^j\right) - c_i\left(x^i\right) + x^i \phi_\zeta^{-1}\left(1 - \alpha_i\right)\right\} \\ &= x^i\left(a - b \cdot \sum_{j=1}^{n} x^j\right) - c_i\left(x^i\right) + x^i \phi_\zeta^{-1}\left(1 - \alpha_i\right). \end{aligned} \tag{6.8}$$

If $x^i = 0, \forall i \in I$, we have

$$u_i^{\alpha_i}(x) = -c_i\left(x^i\right). \tag{6.9}$$

Therefore, for a given x and α_i , the payoff of firm i is

$$u_i^{\alpha_i}(x) = x^i\left(a - b \cdot \sum_{j=1}^{n} x^j\right) - c_i\left(x^i\right) + x^i \phi_\zeta^{-1}\left(1 - \alpha_i\right). \tag{6.10}$$

Definition 6.1 *The nonlinear complementarity problem $NCP(F)$ is to find a vector $y^* \in \mathbb{R}^m$ such that*

$$0 \leq y^* \perp F(y^*) \geq 0, \tag{6.11}$$

where $F : \mathbb{R}^m \to \mathbb{R}^m$.

Theorem 6.2 ([39]) *Denote a decision vector $y = \left(x^1, \ldots, x^n, \lambda^1, \ldots, \lambda^n\right) \in \mathbb{R}^{2n}$, and let $F(y) = (F_1(y), \ldots, F_{2n}(y))$, where*

$$F_i(y) = \begin{cases} -\left(a - b\sum_{j=1;j\neq i}^{n} x^j\right) + 2bx^i + \frac{dc_i(x^i)}{dx^i} - \phi_\zeta^{-1}(1-\alpha_i) + \lambda^i, & \text{if } i = 1, \ldots, n \\ C^{i-n} - x^{i-n}, & \text{if } i = n+1, \ldots, 2n. \end{cases} \tag{6.12}$$

The strategy profile x^* of $y^* = (x^*, \lambda^*)$ is a Nash equilibrium of the CCG if and only if y^* is a solution of the $NCP(F)$.

Proposition 6.3 ([31, 250]) *The nonlinear projection equation $NPE(F)$ is to find a vector $y^* \in \mathbb{R}^m$ such that*

$$(y^* - F(y^*))^+ = y^*, \tag{6.13}$$

where $F : \mathbb{R}^m \to \mathbb{R}^m$ is a continuous function, $(y)^+ = \max(0, y)$. y^* is the solution of $NCP(F)$ if and only if y^* is the solution of $NPE(F)$.

By Theorem 6.1, a Nash equilibrium exists for this CCG with any $\alpha \in (0.5, 1]^n$. By Theorem 6.2 and Proposition 6.3, the Nash equilibrium of this CCG can be obtained by solving the $NPE(F)$. Note that NPEs are equivalent to variational inequalities and generalized complementarity problems, the proof is given by Browder fixed-point theorem [31, 250]. Here, Proposition 6.3 considers a simpler case, i.e., the equivalence between NPE and NCP.

6.2.3 . Neurodynamic Optimization

Xia & Feng [32] proposes the following ODE system to solve the $NPE(F)$

$$\frac{\mathrm{d}z}{\mathrm{d}t} = -F\left((z)^+\right) + (z)^+ - z. \tag{6.14}$$

The ODE system (6.14) can be simplified as $\frac{\mathrm{d}z}{\mathrm{d}t} = \Phi(z)$. Let z^* be an equilibrium point of the ODE system, i.e., $\Phi(z^*) = 0$. Then, we have

$$z^* = -F\left((z^*)^+\right) + (z^*)^+ \tag{6.15}$$

Applying the projection operator $(\cdot)^+$ on both side, we have

$$(z^*)^+ = \left(-F\left((z^*)^+\right) + (z^*)^+\right)^+, \tag{6.16}$$

and hence the point $(z^*)^+$ is a solution of $NPE(F)$.

Definition 6.2 *Consider an ODE system* $\frac{dz}{dt} = \Phi(z)$, $\Phi(z) : \mathbb{R}^m \to \mathbb{R}^m$, *and a given initial point* $(t_0, z_0) \in \mathbb{R}^{m+1}$. *A vector value function* $z(t) : \mathbb{R} \to \mathbb{R}^m$ *is called a state solution, if it satisfies the initial condition* $z(t_0) = z_0$ *and the ODE system* $\frac{dz}{dt} = \Phi(z)$.

Definition 6.3 *An ODE system* $\frac{dz}{dt} = \Phi(z)$ *converges globally to a solution set* $\mathcal{Z}^*$ *if for any given initial point, the state solution* $z(t)$ *satisfies*

$$\lim_{t\to\infty} \operatorname{dist}(z(t), \mathcal{Z}^*) = 0, \tag{6.17}$$

where $\operatorname{dist}(z(t), \mathcal{Z}^*) = \inf_{z^*\in\mathcal{Z}^*} \|z(t) - z^*\|$, *and* $\|\cdot\|$ *is the euclidean norm. In particular, if the set* $\mathcal{Z}^*$ *contains only one point* z^*, *then* $\lim_{t\to\infty} z(t) = z^*$, *and the ODE system is globally asymptotically stable at* z^*.

Theorem 6.4 (Xia & Feng [32], Theorem 1) *If* $\nabla G(z)$ *is symmetrical and positive semi-definite, then the ODE system* (6.14) *converges globally to the solution set of* $NPE(K, G)$. *In particular, if* $NPE(K, G)$ *has only one solution* z^*, *then* z^* *is globally asymptotically stable.*

6.3 . CCGnet

In Section 6.3.1, we summarize the reformulation of a CCG to an initial value problem (IVP) and present a method for parametrizing CCG instances using θ. Section 6.3.2 introduces the CCGnet model and its associated loss function. Section 6.3.3 presents the training algorithm and discusses newly introduced hyperparameters.

6.3.1 . Problem Setup

In this work, we consider the stochastic cournot game as introduced in Section 6.2.2. The CCG problem can be reformulated as an NPE where the solution $y^* = (x^*, \lambda^*)$ includes the Nash equilibrium x^* of the CCG. The neurodynamic approach, introduced in Section 6.2.3, models this NPE as an IVP, resulting in a state solution $z(t)$ for $t \in [t_0, T]$. According to the global convergence theorem, as T approaches infinity, $z(T)$ converges to y^*. Figure 6.1 summarizes this reformulation from a CCG to an IVP.

Next, we consider the case of multiple instances. We parameterize a CCG instance by $\theta \in \Theta$, where a different θ leads to a different CCG instance. The set Θ is typically an uncountably infinite set that represents a range of possible values for θ. For example, in the stochastic cournot game, θ can be the market price factor a or b, and Θ can be $[1, 5]$. We denote the CCG instance for θ as CCG^θ, the corresponding NPE as NPE^θ, and the IVP as IVP^θ.

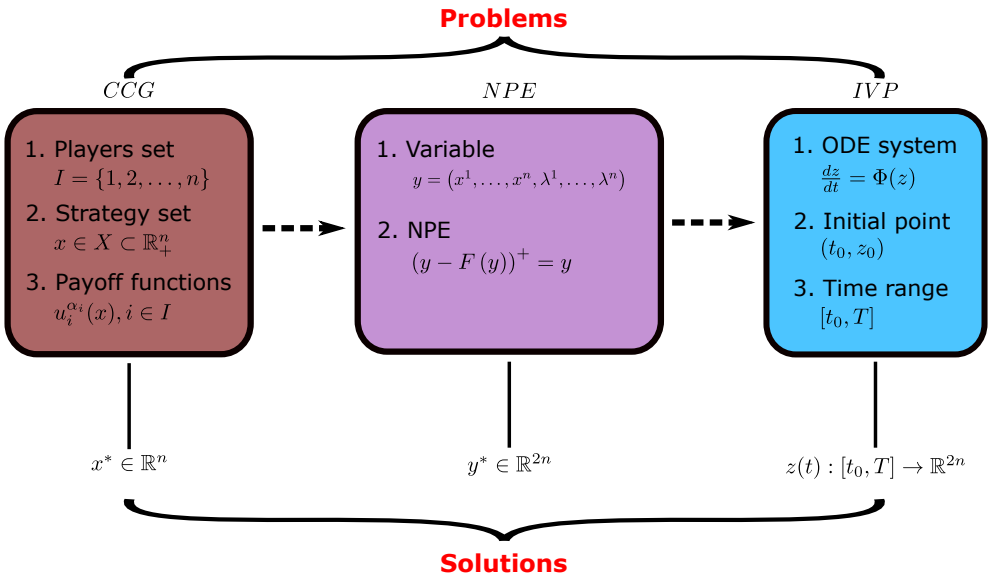


Figure 6.1 – The problem reformulation from a CCG to an IVP via an NPE. We use the stochastic cournot competition for illustration. In the left box of CCG, x^* represents the Nash equilibrium, and the payoff function $u_i^{\alpha_i}(x)$ is given by (6.10). In the middle box of NPE, y^* represents the solution of the NPE, and the function $F(y)$ is given by (6.12). In the right box of IVP, $z(t), t \in [t_0, T]$ represents the state solution, and the ODE system $\Phi(z)$ is given by (6.14).

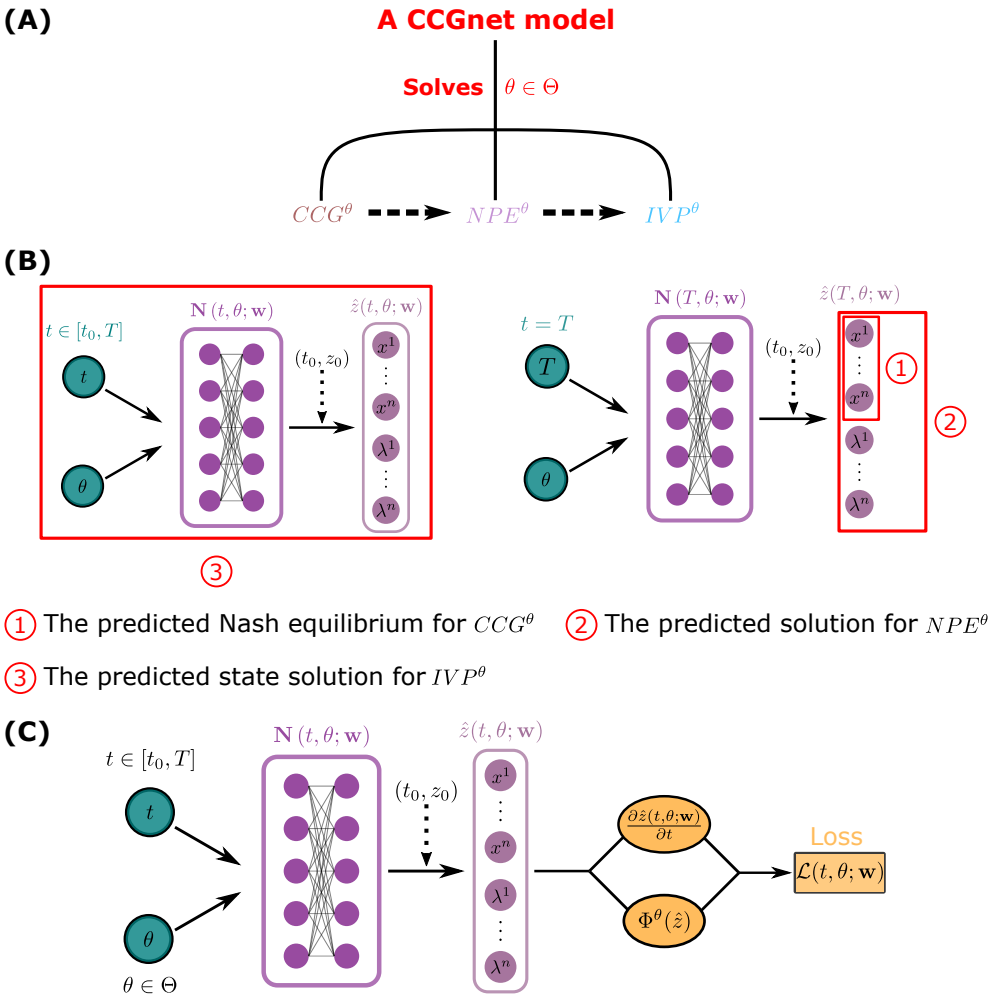
6.3.2 . CCGnet Framework

The CCGnet model is defined by the following equation :

$$\hat{z}\left(t, \theta; \mathbf{w}\right) = z_0 + (1 - e^{-(t-t_0)})\mathbf{N}\left(t, \theta; \mathbf{w}\right), \tag{6.18}$$

where t is an input time that falls within the time range $[t_0, T] \subset \mathbb{R}$. (t_0, z_0) is an initial point. $\mathbf{N}\left(t, \theta; \mathbf{w}\right)$ is a fully-connected neural network with weight $\mathbf{w}$. We put $\theta \in \Theta$ as an input to the neural network, allowing the CCGnet model to solve multiple CCG instances. The terms z_0 and $\left(1 - e^{-(t-t_0)}\right)$ in (6.18) ensure that the CCGnet model satisfies the initial condition (t_0, z_0) by construction, i.e., $\hat{z}\left(t_0, \theta; \mathbf{w}\right) = z_0$. This construction method for handling initial conditions was introduced by Lagaris et al [193]. We use an exponential multiplier of $1 - e^{-(t-t_0)}$, which has been shown to achieve better convergence than the Lagaris method [206].

The CCGnet model solves CCG^θ, NPE^θ and IVP^θ, for any $\theta \in \Theta$, as shown in Figure 6.2-(A). For a given instance of parameter θ, the CCGnet model's predicted state solution for the IVP^θ is $\hat{z}\left(t, \theta; \mathbf{w}\right)$, where $t \in [t_0, T]$. This is obtained by using t as a variable and keeping θ constant. The predicted solution



① The predicted Nash equilibrium for CCG^θ ② The predicted solution for NPE^θ

③ The predicted state solution for IVP^θ

Figure 6.2 – CCGnet framework. (A) One CCGnet model to solve multiple CCG instances. (B) The CCGnet predictions for the CCG^θ, NPE^θ and IVP^θ. (C) The computation flow of the CCGnet loss.

for the NPE^θ is $\hat{z}(T, \theta; \mathbf{w}) = [\hat{x}_1, \ldots, \hat{x}_n, \hat{\lambda}^1, \ldots, \hat{\lambda}^n]$, which is obtained by using a constant value of T and a constant value of θ. The predicted Nash equilibrium for the CCG^θ is $[\hat{x}_1, \ldots, \hat{x}_n]$. Figure 6.2-(B) illustrates how the CCGnet model gives predictions for these three problems.

The loss function is defined by the following equation :

$$\mathcal{L}(t, \theta; \mathbf{w}) = e^{(-\tau(t-t_0))} \ell \left(\frac{\partial \hat{z}(t, \theta; \mathbf{w})}{\partial t}, \Phi^\theta(\hat{z}(t, \theta; \mathbf{w})) \right). \tag{6.19}$$

Φ^θ is the ODE system corresponding to CCG^θ. $\frac{\partial \hat{z}(t,\theta;\mathbf{w})}{\partial t}$ is the partial derivative of the CCGnet model with respect to time t. $\ell(\cdot, \cdot)$ is an error metric, e.g., mean

square error. The term $\ell\left(\frac{\partial \hat{z}(t,\theta;\mathbf{w})}{\partial t}, \Phi^{\theta}(\hat{z}(t,\theta;\mathbf{w}))\right)$ represents how well the CG-Gnet model solves the ODE system Φ^{θ} at time t. The weighting function $e^{(-\tau(t-t_0))}$ is an exponentially decaying function with respect to time t, with $\tau \in \mathbb{R}$ as a hyper-parameter. We include this weighting function to avoid initial errors which might increase the global errors exponentially [230]. For a given instance of parameter $\theta \in \Theta$ and a time $t \in [t_0, T]$, the computational flow of the loss value $\mathcal{L}(t, \theta; \mathbf{w})$ is shown in Figure 6.2-(C). The batch loss is defined as follows :

$$\mathcal{L}(\mathbb{T}, \theta; \mathbf{w}) = \frac{1}{|\mathbb{T}|} \sum_{t \in \mathbb{T}} \mathcal{L}(t, \theta; \mathbf{w}), \tag{6.20}$$

where $\mathbb{T} \subset [t_0, T]$ is the batch of t, and $|\mathbb{T}|$ represents the batch size.

The objective function of the CCGnet model is given by the following equation :

$$E(\mathbf{w}) = \int_{\theta \in \Theta} \int_{t \in [t_0, T]} \mathcal{L}(t, \theta; \mathbf{w}), dt, d\theta. \tag{6.21}$$

The goal of training the CCGnet model is to minimize the objective function, i.e.,

$$\min_{\mathbf{w}} E(\mathbf{w}). \tag{6.22}$$

The loss value $\mathcal{L}(t, \theta; \mathbf{w})$ measures the error of the instance with θ at time t. The objective value $E(\mathbf{w})$ measures the overall error for all instances of $\theta \in \Theta$ over the time range $[t_0, T]$.

6.3.3 . CCGnet Training

Algorithm 4 Training of the CCGnet model for solving $CCG^{\theta}\ \forall \theta \in \Theta$

Input : Time range $[t_0, T]$; Initial point (t_0, z_0); Parameter set Θ
Output : The CCGnet model after training

function
  while iter $\leq$ Max iteration **do**
    $\theta \sim \Theta$: Uniformly sample a θ from the set Θ
    $\mathbb{T} \sim U(t_0, T)$: Sample collocation points $\mathbb{T}$ from the interval $[t_0, T]$
    Φ^{θ} : Derive the ODE system Φ^{θ} related to the instance CCG^{θ}
    Forward propagation : Compute the batch loss $\mathcal{L}(\mathbb{T}, \theta; \mathbf{w})$
    Backward propagation : Update the model weights $\mathbf{w}$ by $\nabla_{\mathbf{w}} \mathcal{L}(\mathbb{T}, \theta; \mathbf{w})$
  end while
end function

Algorithm 4 presents the training of a CCGnet model for solving $CCG^{\theta}\ \forall \theta \in \Theta$. At each training iteration, a value of θ is randomly sampled from the set Θ, and a batch of time points $\mathbb{T}$ is randomly sampled from the time range $[t_0, T]$, forming a training data $(\theta, \mathbb{T})$. The CCGnet model is then trained using this batch of data, and once the iteration finishes, the batch is discarded. The goal of the algorithm

is to minimize the objective function $E(\mathbf{w})$, and the batch loss $\mathcal{L}(\mathbb{T}, \theta; \mathbf{w})$ is an estimate of $E(\mathbf{w})$.

The time range $[t_0, T]$ is a hyperparameter that affects both the prediction accuracy and training difficulty of the CCGnet model. Given a CCG^θ and its corresponding NPE^θ with solution y^*, the initial value problem IVP^θ with time range $[t_0, T]$ has state solution $z(t)$, where $z(T) \approx y^*$. The predicted state solution $\hat{z}(t, \theta; \mathbf{w})$ approximates $z(t)$ on $[t_0, T]$, such that $\hat{z}(T, \theta; \mathbf{w}) \approx z(T) \approx y^*$. Increasing the time range $[t_0, T]$ leads to higher accuracy for $z(T)$ and, subsequently, a higher accuracy limit for $\hat{z}(T, \theta; \mathbf{w})$. However, a larger time range also makes training more challenging as the model has a larger input space to learn.

The size of the time range is a trade-off that must be considered carefully. If the time range is too small, the model will have a lower accuracy limit and will not be able to surpass it, regardless of the number of training iterations. If the time range is too large, the model will require more iterations to reach its accuracy limit. Thus, it is important to choose a time range that is appropriate for the number of training iterations.

6.4 . Numerical Results

We conducted our experiments using the Google Colab platform and built the neural network with Pytorch 1.9.1 and the ODE system with JAX 0.3.13 [207]. The hyperparameters for training are as follows :

— The ADAM optimizer [27] was used for training with a learning rate of 0.001 and a batch size of 512. The maximum number of iterations was set to 10,000.

— The CCGnet model consists of a fully connected neural network with three hidden layers, each containing 100 neurons and using the tanh activation function.

— The time range for the model was set to the interval $[0, 1]$ with an initial point of $(0, \mathbf{0})$.

— The mean squared error (MSE) was used as the error metric and the weighting hyperparameter was set to $\tau = 0.5$.

The CCGnet model was compared to four numerical integration methods : RK45, LSODA, BDF, and DOP853 [208, 251, 252, 253]. These four methods can be accessed using Scipy [209].

We consider a concrete example of stochastic cournot competitions as introduced in Section 6.2.2. The number of electricity firms are $n = 5$, and the cost function of firm i is defined as $c_i\left(x^i\right) = \left(x^i\right)^2$. The random variable follows the normal distribution $\zeta \sim N\left(\mu, \sigma^2\right)$. The Nash equilibrium of this game can be reformulated as the following NPE

$$P_Y\left(y - (My + q)\right) = y, \tag{6.23}$$

where $Y = \{y \in \mathbb{R}^{10} \mid y \geq 0\}$, $y = \left(x^1, x^2, x^3, x^4, x^5, \lambda^1, \lambda^2, \lambda^3, \lambda^4, \lambda^5\right)^T$,

$$M = \begin{pmatrix} 2b+2 & b & b & b & b & 1 & 0 & 0 & 0 & 0 \\ b & 2b+2 & b & b & b & 0 & 1 & 0 & 0 & 0 \\ b & b & 2b+2 & b & b & 0 & 0 & 1 & 0 & 0 \\ b & b & b & 2b+2 & b & 0 & 0 & 0 & 1 & 0 \\ b & b & b & b & 2b+2 & 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad (6.24)$$

$$q = \begin{pmatrix} -a - \phi_\zeta^{-1}(1-\alpha_1) \\ -a - \phi_\zeta^{-1}(1-\alpha_2) \\ -a - \phi_\zeta^{-1}(1-\alpha_3) \\ -a - \phi_\zeta^{-1}(1-\alpha_4) \\ -a - \phi_\zeta^{-1}(1-\alpha_5) \\ C^1 \\ C^2 \\ C^3 \\ C^4 \\ C^5 \end{pmatrix}. \quad (6.25)$$

The two market price factors are $a = 1$ and $b = 2$, The capacity of each firm is $C^1 = C^2 = C^3 = C^4 = C^5 = 5$. The confidence level of each firm are $\alpha^1 = \alpha^2 = \alpha^3 = \alpha^4 = \alpha^5 = 0.6$. The mean and the variance of the normal distribution is $\mu = 1$ and $\sigma^2 = 2$.

We use the following metric to evaluate the accuracy of the prediction $\hat{y}$

$$\epsilon = \|P_Y(\hat{y} - (M\hat{y} + q)) - \hat{y}\|. \quad (6.26)$$

We consider three different ways to parameterize this CCG problem. Subsection 6.4.1 parameterize the market price factor a as a variable. Subsection 6.4.2 parameterize the market price factor b as a variable. Subsection 6.4.3 parameterize the confidence level $\bar{\alpha}$ as a variable. We construct three independent CCGnet models, each corresponding to one subsection, then train and test these three models separately.

The experimental setup for Sections 6.4.1, 6.4.2, and 6.4.3 is the same and each subsection includes the following results : (1) the training loss of the CCGnet model, (2) the predicted state solutions of four IVPs corresponding to four CCG instances, (3) the predicted solutions to the four CCG instances, and (4) a comparison of CPU time between the CCGnet model and the numerical integration methods. Finally, Section 6.4.4 compares the advantages and limitations of our method to the numerical integration methods.

6.4.1 . Case 1 : a as Variable

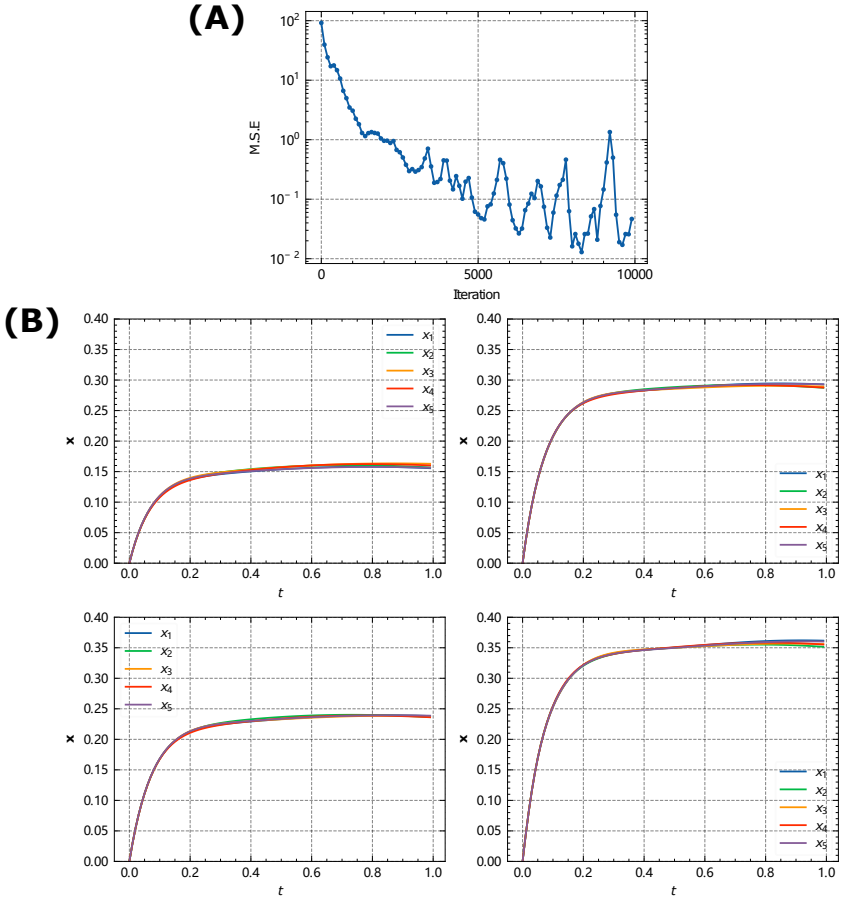


Figure 6.3 – Case 1 : a as variable. (A) The training loss versus the number of iterations. (B) The predicted state solutions for the four example instances.
The top-left, top-right, bottom-left, and bottom-right show the results of $a = 1.37$, $a = 3.27$, $a = 2.53$, and $a = 4.20$, respectively.

We developed a CCGnet model, denoted as :

$$\hat{z}(t, a; \mathbf{w}), \quad t \in [0, 1], a \in [1, 5], \tag{6.27}$$

to solve for the case when the market price factor a is a variable with a parameter set of $\Theta = [1, 5]$. At each iteration, a market price factor a is uniformly sampled from the interval $[1, 5]$, and a batch of time $\mathbb{T}$ is uniformly sampled from the time range $[0, 1]$, together forming the batch $(a, \mathbb{T})$ to train the model. Figure 6.3-(A) shows the loss value during training, which decreases from an initial value of 91.75 to 0.05.

We selected four different values of a to represent four different instances and used the CCGnet model to solve them. Denote $\hat{z}(t, a; \mathbf{w}) = \Big(\hat{x}(t, a; \mathbf{w}), \hat{\lambda}(t, a; \mathbf{w})\Big)$.

Table 6.2 – Case 1 : a as variable. The predicted Nash equilibrium for the four example instances. CCGnet prediction refers to the predicted Nash equilibrium from the CCGnet model. Nash equilibrium refers to the true value.

Index	a	CCGnet prediction	Nash equilibrium
1	1.37	[0.16, 0.16, 0.16, 0.16, 0.16]	[0.14, 0.14, 0.14, 0.14, 0.14]
2	3.27	[0.29, 0.29, 0.29, 0.29, 0.29]	[0.28, 0.28, 0.28, 0.28, 0.28]
3	2.53	[0.24, 0.24, 0.24, 0.24, 0.24]	[0.23, 0.23, 0.23, 0.23, 0.23]
4	4.20	[0.36, 0.35, 0.36, 0.35, 0.36]	[0.35, 0.35, 0.35, 0.35, 0.35]

In the following results, we only present the results of $\hat{x}(t, a; \mathbf{w})$. Figure 6.3-(B) shows the predicted state solutions for $\hat{x}(t, a = 1.37; \mathbf{w})$, $\hat{x}(t, a = 3.27; \mathbf{w})$, $\hat{x}(t, a = 2.53; \mathbf{w})$, and $\hat{x}(t, a = 4.20; \mathbf{w})$. Table 6.2 shows the predicted Nash equilibria for these four instances at $t = 1$, i.e., $\hat{x}(t = 1, a = 1.37; \mathbf{w})$, $\hat{x}(t = 1, a = 3.27; \mathbf{w})$, $\hat{x}(t = 1, a = 2.53; \mathbf{w})$, and $\hat{x}(t = 1, a = 4.20; \mathbf{w})$.

Table 6.3 – Case 1 : a as variable. The computational performance of the CCGnet model and the neurodynamic approach. Each row represents a test batch of many different instances. With or without GPU, refers to whether the CCGnet model uses CUDA. RK45, LSODA, BDF, and DOP853 are four numerical integration methods.

Instance number	CCGnet			Neurodynamic approach			
	CPU time (without GPU) (ms)	CPU time (with GPU) (ms)	ϵ error	RK45 CPU time (ms)	LSODA CPU time (ms)	BDF CPU time (ms)	DOP853 CPU time (ms)
1	< 1	< 1	0.27	2.51	2.23	8.94	3.41
100	< 1	< 1	0.21	250	217	852	341
500	1.07	< 1	0.20	1260	1120	4340	1760
1000	2.18	< 1	0.20	2590	2280	8830	3510
5000	8.94	1.1	0.20	12800	11200	43900	17400
10000	17.5	1.53	0.20	25500	22500	84000	34500

Table 6.3 compares the computational performance of the CCGnet model, $\hat{z}(t, a; \mathbf{w})$, and the neurodynamic approach when solving multiple instances. When solving a single instance, the CCGnet model has a CPU time of less than 1 ms, which is faster than the best result of 2.23 ms for the neurodynamic approach. When solving 10,000 instances, the CCGnet model takes only 1.53 ms of CPU time, significantly faster than the best result of 22,500 ms for the neurodynamic approach. On average, the CCGnet model has an error of $\epsilon = 0.2$.

6.4.2 . Case 2 : b as Variable

We developed a CCGnet model, denoted as :

$$\hat{z}(t, b; \mathbf{w}), \quad t \in [0, 1], b \in [1, 5], \tag{6.28}$$

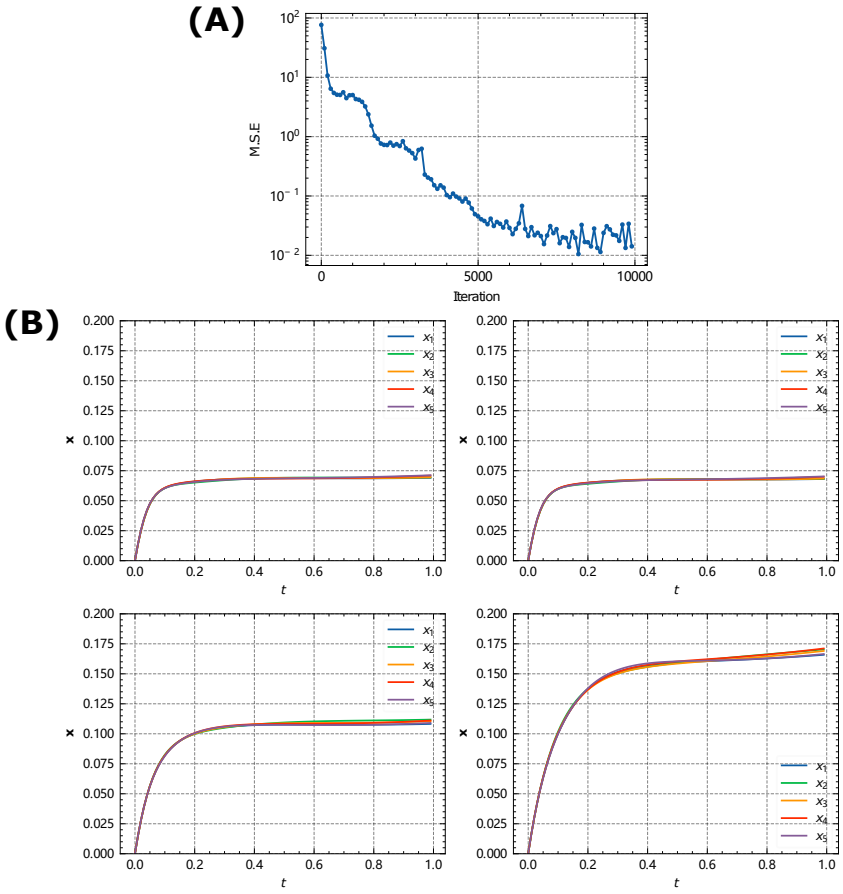


Figure 6.4 – Case 2 : b as variable. (A) The training loss versus the number of iterations. (B) The predicted state solutions for the four example instances. The top-left, top-right, bottom-left, and bottom-right show the results of $b = 3.83$, $b = 3.90$, $b = 2.34$, and $b = 1.46$, respectively.

to solve for the case when the market price factor b is a variable with a parameter set of $\Theta = [1, 5]$. At each iteration, a market price factor b is uniformly sampled from the interval $[1, 5]$ and a batch of time points, denoted as $\mathbb{T}$, is uniformly sampled from the time range $[0, 1]$. These two samples, $(b, \mathbb{T})$, form a batch used to train the model. Figure 6.4-(A) shows the loss value during training, which decreases from an initial value of 76.51 to 0.01.

We tested this CCGnet model on four example instances with different values of b, specifically 3.83, 3.90, 2.34, and 1.46. Figure 6.4-(B) shows the predicted state solutions for $\hat{x}(t, b = 3.83; \mathbf{w})$, $\hat{x}(t, b = 3.90; \mathbf{w})$, $\hat{x}(t, b = 2.34; \mathbf{w})$, and $\hat{x}(t, b = 1.46; \mathbf{w})$. Table 6.4 shows the predicted Nash equilibria for these four instances at $t = 1$, i.e., $\hat{x}(t = 1, b = 3.83; \mathbf{w})$, $\hat{x}(t = 1, b = 3.90; \mathbf{w})$, $\hat{x}(t = 1, b = 2.34; \mathbf{w})$, and $\hat{x}(t = 1, b = 1.46; \mathbf{w})$.

Table 6.5 compares the computational performance of the CCGnet model,

Table 6.4 – Case 2 : b as variable. The predicted Nash equilibrium for the four example instances. CCGnet prediction refers to the predicted Nash equilibrium from the CCGnet model. Nash equilibrium refers to the true value.

Index	b	CCGnet prediction	Nash equilibrium
1	3.83	[0.07, 0.07, 0.07, 0.07, 0.07]	[0.07, 0.07, 0.07, 0.07, 0.07]
2	3.90	[0.07, 0.07, 0.07, 0.07, 0.07]	[0.06, 0.06, 0.06, 0.06, 0.06]
3	2.34	[0.11, 0.11, 0.11, 0.11, 0.11]	[0.10, 0.10, 0.10, 0.10, 0.10]
4	1.46	[0.17, 0.17, 0.17, 0.17, 0.17]	[0.15, 0.15, 0.15, 0.15, 0.15]

Table 6.5 – Case 2 : b as variable. The computational performance of the CCGnet model and the neurodynamic approach. Each row represents a test batch of many different instances. With or without GPU, refers to whether the CCGnet model uses CUDA. RK45, LSODA, BDF, and DOP853 are four numerical integration methods.

Instance number	CCGnet			Neurodynamic approach			
	CPU time (ms)	CPU time (with GPU) (ms)	ϵ error	RK45 CPU time (ms)	LSODA CPU time (ms)	BDF CPU time (ms)	DOP853 CPU time (ms)
1	< 1	< 1	0.04	3.09	2.64	8.70	3.68
100	< 1	< 1	0.07	282	247	841	355
500	1.03	< 1	0.07	1510	1410	4290	1840
1000	2.02	< 1	0.07	3000	2620	8550	3740
5000	8.61	1.07	0.07	15000	13000	42700	18500
10000	17.1	1.50	0.07	30200	26100	84000	36900

$\hat{z}(t, b; \mathbf{w})$, and the neurodynamic approach. When solving a single instance, the CCGnet model has a CPU time of less than 1 ms, outperforming the best result of 2.64 ms for the neurodynamic approach. When solving 10,000 instances, the CCGnet model takes only 1.50 ms of CPU time, faster than the best result of 26,100 ms for the neurodynamic approach. On average, the CCGnet model has an error of $\epsilon = 0.07$.

6.4.3 . Case 3 : $\bar{\alpha}$ as Variable

We studied the case where the confidence level is a variable and assumed that $\alpha^1 = \alpha^2 = \alpha^3 = \alpha^4 = \alpha^5$. We denote this common value as $\bar{\alpha}$ and build a CCGnet model, denoted as :

$$\hat{z}(t, \alpha; \mathbf{w}), \quad t \in [0, 1], \alpha \in [0.5, 0.9], \tag{6.29}$$

to solve for this case when $\bar{\alpha}$ is a variable with a parameter set of $\Theta = [0.5, 0.9]$. At each iteration, a value of $\bar{\alpha}$ is uniformly sampled from the interval $[0.5, 0.9]$ and a batch of time points, denoted as $\mathbb{T}$, is uniformly sampled from the time range $[0, 1]$. These two samples, $(\bar{\alpha}, \mathbb{T})$, form a batch used to train the model. Figure

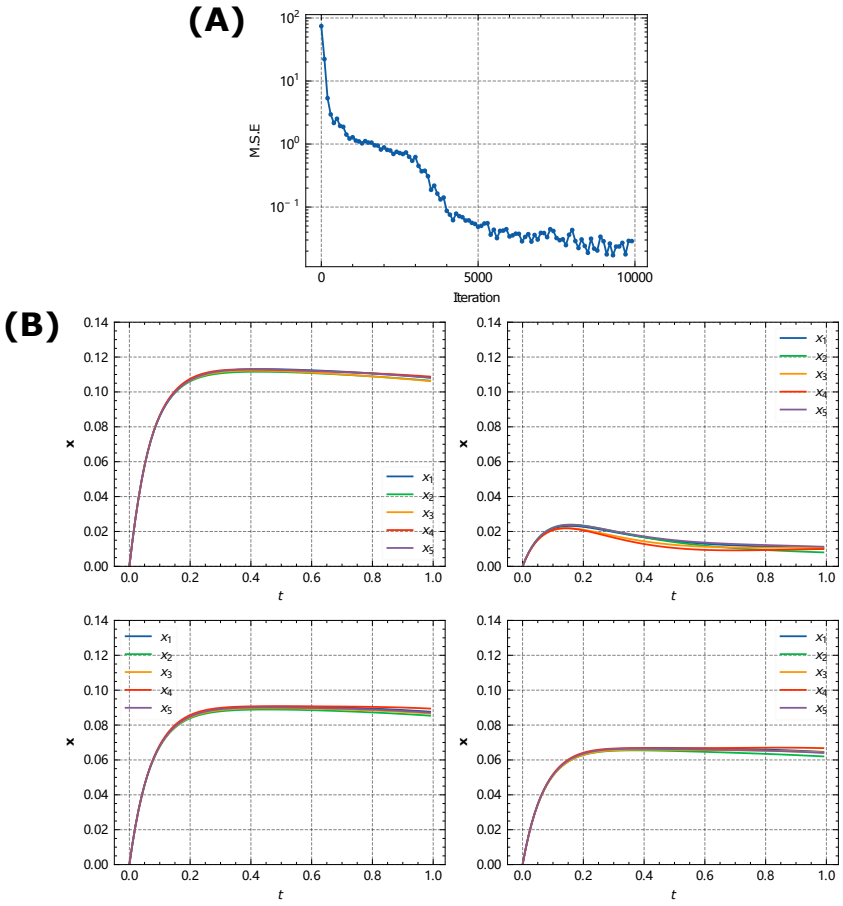


Figure 6.5 – Case 3 : $\bar{\alpha}$ as variable. (A) The training loss versus the number of iterations (B) The predicted state solutions for the four instances The top-left, top-right, bottom-left, and bottom-right show the results of $\bar{\alpha} = 0.60$, $\bar{\alpha} = 0.90$, $\bar{\alpha} = 0.67$, and $\bar{\alpha} = 0.75$, respectively.

Table 6.6 – Case 3 : $\bar{\alpha}$ as variable. The predicted Nash equilibrium for the four example instances. CCGnet prediction refers to the predicted Nash equilibrium from the CCGnet model. Nash equilibrium refers to the true value.

Index	$\bar{\alpha}$	CCGnet endpoint	Nash equilibrium
1	0.60	[0.11, 0.11, 0.11, 0.11, 0.11]	[0.12, 0.12, 0.12, 0.12, 0.12]
2	0.90	[0.01, 0.01, 0.01, 0.01, 0.01]	[0.01, 0.01, 0.01, 0.01, 0.01]
3	0.67	[0.09, 0.09, 0.09, 0.09, 0.09]	[0.10, 0.10, 0.10, 0.10, 0.10]
4	0.75	[0.06, 0.06, 0.06, 0.07, 0.06]	[0.08, 0.08, 0.08, 0.08, 0.08]

6.5-(A) shows the loss value during training, which decreases from an initial value of 74.39 to 0.03.

We tested this CCGnet model on four example instances with different values

of $\bar{\alpha}$, specifically 0.60, 0.90, 0.67, and 0.75. Figure 6.5-(B) shows the predicted state solutions for $\hat{x}(t, \bar{\alpha} = 0.60; \mathbf{w})$, $\hat{x}(t, \bar{\alpha} = 0.90; \mathbf{w})$, $\hat{x}(t, \bar{\alpha} = 0.67; \mathbf{w})$, and $\hat{x}(t, \bar{\alpha} = 0.75; \mathbf{w})$. Table 6.6 shows the predicted Nash equilibria for these four instances at $t = 1$, i.e., $\hat{x}(t = 1, \bar{\alpha} = 0.60; \mathbf{w})$, $\hat{x}(t = 1, \bar{\alpha} = 0.90; \mathbf{w})$, $\hat{x}(t = 1, \bar{\alpha} = 0.67; \mathbf{w})$, and $\hat{x}(t = 1, \bar{\alpha} = 0.75; \mathbf{w})$.

Table 6.7 – Case 3 : $\bar{\alpha}$ as variable. The computational performance of the CCGnet model and the neurodynamic approach. Each row represents a test batch of many different instances. With or without GPU, refers to whether the CCGnet model uses CUDA. RK45, LSODA, BDF, and DOP853 are four numerical integration methods.

Instance number	CCGnet			Neurodynamic approach			
	CPU time (ms)	CPU time (with GPU) (ms)	ϵ error	RK45 CPU time (ms)	LSODA CPU time (ms)	BDF CPU time (ms)	DOP853 CPU time (ms)
1	< 1	< 1	0.16	2.57	2.27	8.02	3.47
100	< 1	< 1	0.15	267	225	838	347
500	1.02	< 1	0.16	1320	1180	4230	1770
1000	2.05	< 1	0.15	2640	2420	8440	3540
5000	8.52	1.08	0.15	13200	12200	42300	17700
10000	17.9	1.41	0.15	26600	24300	87000	35700

Table 6.7 compares the computational performance of the CCGnet model, $\hat{z}(t, \alpha; \mathbf{w})$, and the neurodynamic approach. When solving a single instance, the CCGnet model has a CPU time of less than 1 ms, faster than the best result of 2.27 ms for the neurodynamic approach. When solving 10,000 instances, the CCGnet model takes only 1.14 ms of CPU time, significantly faster than the best result of 24,300 ms for the neurodynamic approach. On average, the CCGnet model has an error of $\epsilon = 0.15$.

6.4.4 . Discussion

The main advantage of the CCGnet model is its computational performance. It can directly predict the Nash equilibrium without any iterative process, making it much faster than numerical integration methods. This advantage becomes even more significant when there are a large number of instances to solve. For example, when solving 10,000 different instances, the CCGnet model can predict all the Nash equilibria in a one-shot manner with only 1.5 ms, while numerical methods require more than 20,000 ms to solve each instance one after another. Additionally, the CCGnet model can utilize a GPU to further accelerate the solution process, while GPU-based numerical methods are still under development.

One limitation of the CCGnet model is its prediction accuracy compared to exact solutions obtained through numerical integration methods. While numerical integration methods can provide exact solutions given sufficient computational time, the CCGnet model can only provide predictions. The accuracy of these predictions depends on technical details such as the neural network structure, training

algorithm, and hyperparameter settings, which are active areas of research in machine learning and deep learning. As these areas progress, the CCGnet model has the potential to improve its prediction accuracy.

6.5 . Conclusion

This chapter introduced a deep learning approach called CCGnet for solving chance-constrained games. CCGnet is based on neurodynamic optimization, which models a chance-constrained game as an ODE system. One of the key benefits of CCGnet is its ability to solve multiple instances in a very short amount of CPU time, significantly faster than traditional methods. We provide a detailed description of the proposed method, including the parametrization of CCG instances, the model framework, the training algorithm, and a discussion of hyperparameters.

However, it is important to note that the proposed method should not be considered a replacement for standard solvers like RK45 and BDF. These methods have been well-developed over many years. Our purpose is to link the machine learning community and CCG. We believe that with the rapid growth of research on machine learning, both methodologically and experimentally, this work will continue to contribute to the efficient solution of CCG problems.

There are many potential avenues for future research. Some examples include : 1) Choosing the hyperparameter initial point to be all zero may not always lead to the best computational performance. Is it possible to find other choices that lead to better results ? 2) We used a uniform distribution to sample the dataset. Could other sampling methods lead to better results ? 3) We used a fully-connected network structure. How can we design a more suitable neural network structure and activation function ?

7 - Conclusions and Perspective

7.1 . Conclusions

This thesis used deep learning to solve four types of nonlinear optimization problems : bimatrix games, nonlinear projection equations, nonsmooth convex optimization problems, and chance-constrained games.

For bimatrix games, they can be represented directly by their two payoff matrices, and we use CNNs to solve for such problems. Specifically, the CNN takes a bimatrix game as input and outputs a predicted Nash equilibrium.

For more complex nonlinear optimization problems, they cannot be fed directly into a neural network. For these complex problems, we need to use advanced tools namely neurodynamic optimization and PINNs. In this thesis, we use the combination of neurodynamic optimization and PINNs to solve nonlinear projection equations, nonsmooth convex optimization problems, and chance-constrained games. Specifically, we first model the problem as an ODE system using neurodynamic optimization, and the ODE system must be designed to have a global convergence property. Then , we use a PINN-like neural network to solve the ODE system. We solve the original optimization problem by training the nerual network toward solving the ODE system.

In the field of numerical optimization, a well-known principle in algorithm design is to take advantage of the problem structure to improve computational performance. We have applied this principle to the design of our deep learning algorithms, as described below.

— In Chapter 3, we deliberately used CNNs to handle bimatrix games, because the CNNs are much better at handling matrix-type data. Note that using fully connected networks still works for bimatrix games, but it would result in poor performance. This is because fully connected networks process the elements of the two payoff matrices individually, losing the structured information of them.

— In Chapters 4 and 5, we specifically designed training algorithms that focus on improving the end state of the model. If the proposed algorithm were just a simple combination of neurodynamic optimization and PINNs, its performance would be unsatisfactory, as evidenced by the experimental results in Sections 4.5.2 and 5.4.3. In addition, we used the task-specific NPE error (for Chapter 4) and the OuC metric (for Chapter 5), to facilitate training, resulting in significant performance improvements.

— In Chapter 6, we specifically designed a network architecture that allows a single model to solve multiple optimization problems. The problem therein is to solve multiple CCGs that are interrelated and differ only in their confidence

levels α. We designed a network architecture that takes the confidence level α as input and outputs the corresponding predicted Nash equilibrium. Based on this design, a single neural network can solve CCGs at different confidence levels without retraining, greatly improving computational efficiency.

7.2 . Perspective

Finally, we outline some possible future directions :

— Methodology side : Integrate the proposed approaches with cutting-edge ideas, concepts, or theories in deep learning to improve computational efficiency and accuracy. For example, in Chapter 3, the basic CNN model could be replaced by a more advanced architecture such as transformer. In Chapters 4 and 5, the proposed framework can be combined with multi-task learning to solve multiple optimization problems in parallel, or with transfer learning to allow the neural network to use past experience to solve new optimization problems.

— Application side : Apply the proposed approaches to solve real-world problems. For example, certain equilibrium problems in electricity markets can be modeled as NPE problems, which can be solved with our approach. In addition, many operations research problems are typically transformed into large-scale optimization problems that are difficult to solve with traditional solvers. Our approaches provide a viable alternative for solving these challenging optimization problems.

— Theoretical side : Conduct convergence analysis for the proposed approaches. A limitation of this thesis is the lack of rigorous mathematical theories to guarantee the convergence. This limitation stems from the use of neural networks, whose convergence analysis remains a challenging problem. To alleviate this situation, we can use the prediction from the neural network as an initial point to warm start traditional solvers to obtain a converged solution. However, relying solely on neural networks to obtain theoretically guaranteed solutions is still a major challenge. Future research in convergence analysis should take advantage of the extensive knowledge of ODE stability as well as the universal approximation theorems of neural networks.

Bibliographie

[1] X. Chen, X. Deng, S.-H. Teng, Settling the complexity of computing two-player nash equilibria, Journal of the ACM (JACM) 56 (3) (2009) 1–57.

[2] G. Dantzig, Linear programming and extensions, Princeton university press, 1963.

[3] S.-C. Fang, S. Puthenpura, Linear optimization and extensions : theory and algorithms, Prentice-Hall, Inc., 1993.

[4] J. Nocedal, S. Wright, Numerical optimization, Springer Science & Business Media, 2006.

[5] J. H. Holland, Genetic algorithms, Scientific american 267 (1) (1992) 66–73.

[6] A. S. Lewis, M. L. Overton, Nonsmooth optimization via quasi-newton methods, Mathematical Programming 141 (2013) 135–163.

[7] Y. Yang, Q.-S. Jia, Z. Xu, X. Guan, C. J. Spanos, Proximal admm for nonconvex and nonsmooth optimization, Automatica 146 (2022) 110551.

[8] J. Larson, M. Menickelly, S. M. Wild, Derivative-free optimization methods, Acta Numerica 28 (2019) 287–404.

[9] N. Nisan, T. Roughgarden, E. Tardos, V. V. Vazirani, Algorithmic Game Theory, Cambridge University Press, 2007. doi:10.1017/CBO9780511800481.

[10] J. F. Nash, Equilibrium points in n-person games, Proceedings of the National Academy of Sciences 36 (1) (1950) 48–49. doi:10.1073/pnas.36.1.48.

[11] K. J. Arrow, G. Debreu, Existence of an equilibrium for a competitive economy, Econometrica : Journal of the Econometric Society (1954) 265–290.

[12] R. J. Aumann, Correlated equilibrium as an expression of bayesian rationality, Econometrica : Journal of the Econometric Society (1987) 1–18.

[13] S. Hart, A. Mas-Colell, A simple adaptive procedure leading to correlated equilibrium, Econometrica 68 (5) (2000) 1127–1150.

[14] C. Daskalakis, P. W. Goldberg, C. H. Papadimitriou, The complexity of computing a nash equilibrium, SIAM Journal on Computing 39 (1) (2009) 195–259.

[15] C. M. Bishop, N. M. Nasrabadi, Pattern recognition and machine learning, Springer, 2006.

[16] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016, http://www.deeplearningbook.org.

[17] A. Krizhevsky, I. Sutskever, G. E. Hinton, ImageNet classification with deep convolutional neural networks, Communications of the ACM 60 (6) (2017) 84–90. doi:10.1145/3065386.

[18] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, Imagenet : A large-scale hierarchical image database, in : 2009 IEEE conference on computer vision and pattern recognition, Ieee, 2009, pp. 248–255.

[19] J. Devlin, M. Chang, K. Lee, K. Toutanova, BERT: pre-training of deep bi-directional transformers for language understanding, CoRR abs/1810.04805 (2018). arXiv:1810.04805.
URL http://arxiv.org/abs/1810.04805

[20] A. I. Károly, P. Galambos, J. Kuti, I. J. Rudas, Deep learning in robotics : Survey on model structures and training strategies, IEEE Transactions on Systems, Man, and Cybernetics : Systems 51 (1) (2020) 266–279.

[21] Y. Cao, T. A. Geddes, J. Y. H. Yang, P. Yang, Ensemble deep learning in bioinformatics, Nature Machine Intelligence 2 (9) (2020) 500–508.

[22] Y. LeCun, Y. Bengio, Others, Convolutional networks for images, speech, and time series, The handbook of brain theory and neural networks 3361 (10) (1995) 255–258.
URL http://www.iro.umontreal.ca/$\sim$lisa/pointeurs/handbook-convo.pdf

[23] M. Schuster, K. K. Paliwal, Bidirectional recurrent neural networks, IEEE transactions on Signal Processing 45 (11) (1997) 2673–2681.

[24] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, Generative adversarial networks, Communications of the ACM 63 (11) (2020) 139–144. arXiv:1406.2661, doi:10.1145/3422622.
URL http://arxiv.org/abs/1406.2661

[25] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, S. Y. Philip, A comprehensive survey on graph neural networks, IEEE transactions on neural networks and learning systems 32 (1) (2020) 4–24.

[26] M. Raissi, P. Perdikaris, G. E. Karniadakis, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, Journal of Computational Physics 378 (2019) 686–707. doi:https://doi.org/10.1016/j.jcp.2018.10.045.
URL https://www.sciencedirect.com/science/article/pii/S0021999118307125

[27] D. P. Kingma, J. Ba, Adam : A method for stochastic optimization, arXiv preprint arXiv :1412.6980 (2014).

[28] J. Duchi, E. Hazan, Y. Singer, Adaptive subgradient methods for online learning and stochastic optimization., Journal of machine learning research 12 (7) (2011).

[29] C. E. Lemke, J. T. Howson, Jr, Equilibrium points of bimatrix games, Journal of the Society for industrial and Applied Mathematics 12 (2) (1964) 413–423.

[30] R. Savani, B. Von Stengel, Hard-to-solve bimatrix games, Econometrica 74 (2) (2006) 397–429.

[31] P. T. Harker, J.-S. Pang, Finite-dimensional variational inequality and nonlinear complementarity problems : a survey of theory, algorithms and applications, Mathematical programming 48 (1-3) (1990) 161–220.

[32] Y. Xia, G. Feng, A new neural network for solving nonlinear projection equations, Neural Networks 20 (5) (2007) 577–589.

[33] Y. Xia, J. Wang, A Bi-Projection Neural Network for Solving Constrained Quadratic Optimization Problems, IEEE Transactions on Neural Networks and Learning Systems 27 (2) (2016) 214–224. doi:10.1109/TNNLS.2015.2500618.

[34] H. Che, J. Wang, A collaborative neurodynamic approach to global and combinatorial optimization, Neural Networks 114 (2019) 15–27.

[35] S. Boyd, L. Vandenberghe, Convex optimization, Cambridge university press, 2004.

[36] V. V. Singh, O. Jouini, A. Lisser, Existence of nash equilibrium for chance-constrained games, Operations Research Letters 44 (5) (2016) 640–644.

[37] S. Peng, V. V. Singh, A. Lisser, General sum games with joint chance constraints, Operations Research Letters 46 (5) (2018) 482–486.

[38] V. V. Singh, A. Lisser, A characterization of nash equilibrium for the games with random payoffs, Journal of Optimization Theory and Applications 178 (3) (2018) 998–1013.

[39] V. V. Singh, A. Lisser, Variational inequality formulation for the games with random payoffs, Journal of Global Optimization 72 (4) (2018) 743–760. doi:10.1007/s10898-018-0664-8.
URL https://doi.org/10.1007/s10898-018-0664-8

[40] V. V. Singh, A. Lisser, A second-order cone programming formulation for two player zero-sum games with chance constraints, European Journal of Operational Research 275 (3) (2019) 839–845.

[41] H. K. Khalil, Nonlinear Systems, Pearson Education, Prentice Hall, 2002.
URL https://books.google.fr/books?id=t_d1QgAACAAJ

[42] D. W. Tank, J. J. Hopfield, SIMPLE 'NEURAL' OPTIMIZATION NETWORKS : AN A/D CONVERTER, SIGNAL DECISION CIRCUIT, AND A LINEAR PROGRAMMING CIRCUIT., IEEE transactions on circuits and systems CAS-33 (5) (1986) 533–541. doi:10.1109/tcs.1986.1085953.

[43] Y. Xia, G. Feng, J. Wang, A recurrent neural network with exponential convergence for solving convex quadratic program and related linear piecewise equations, Neural Networks 17 (7) (2004) 1003–1015.

[44] S. Liu, J. Wang, A simplified dual neural network for quadratic programming with its kwta application, IEEE Transactions on Neural Networks 17 (6) (2006) 1500–1510.

[45] X. Hu, J. Wang, An improved dual neural network for solving a class of quadratic programming problems and its k-winners-take-all application, IEEE Transactions on Neural networks 19 (12) (2008) 2022–2031.

[46] M. P. Kennedy, L. O. Chua, Neural networks for nonlinear programming, IEEE Transactions on Circuits and Systems 35 (5) (1988) 554–562.

[47] J. Wang, A deterministic annealing neural network for convex programming, Neural networks 7 (4) (1994) 629–641.

[48] Y. Xia, G. Feng, J. Wang, A novel recurrent neural network for solving nonlinear optimization problems with inequality constraints, IEEE Transactions on neural networks 19 (8) (2008) 1340–1353.

[49] N. Liu, S. Qin, A neurodynamic approach to nonlinear optimization problems with affine equality and convex inequality constraints, Neural Networks 109 (2019) 147–158.

[50] G. Li, Z. Yan, J. Wang, A one-layer recurrent neural network for constrained nonconvex optimization, Neural Networks 61 (2015) 10–21.

[51] W. Bian, L. Ma, S. Qin, X. Xue, Neural network for nonsmooth pseudoconvex optimization with general convex constraints, Neural Networks 101 (2018) 1–14.

[52] C. Xu, Y. Chai, S. Qin, Z. Wang, J. Feng, A neurodynamic approach to nonsmooth constrained pseudoconvex optimization problem, Neural Networks 124 (2020) 180–192.

[53] X. Le, J. Wang, A two-time-scale neurodynamic approach to constrained minimax optimization, IEEE Transactions on Neural Networks and Learning Systems 28 (3) (2016) 620–629.

[54] Q. Liu, J. Wang, A projection neural network for constrained quadratic minimax optimization, IEEE Transactions on Neural Networks and Learning Systems 26 (11) (2015) 2891–2900.

[55] Q. Liu, J. Wang, $l\_\{1\}$-minimization algorithms for sparse signal reconstruction based on a projection neural network, IEEE Transactions on Neural Networks and Learning Systems 27 (3) (2015) 698–707.

[56] Y. Zhao, X. He, T. Huang, J. Huang, Smoothing inertial projection neural network for minimization lp- q in sparse signal reconstruction, Neural Networks 99 (2018) 31–41.

[57] Y. Zhao, X. Liao, X. He, R. Tang, W. Deng, Smoothing inertial neurodynamic approach for sparse signal reconstruction via lp-norm minimization, Neural Networks 140 (2021) 100–112.

[58] Y. Wang, X. Li, J. Wang, A neurodynamic optimization approach to supervised feature selection via fractional programming, Neural Networks 136 (2021) 194–206.

[59] Y. Wang, J. Wang, H. Che, Two-timescale neurodynamic approaches to supervised feature selection based on alternative problem formulations, Neural Networks 142 (2021) 180–191.

[60] M.-F. Leung, J. Wang, Minimax and biobjective portfolio selection based on collaborative neurodynamic optimization, IEEE Transactions on Neural Networks and Learning Systems 32 (7) (2020) 2825–2836.

[61] M.-F. Leung, J. Wang, D. Li, Decentralized robust portfolio optimization based on cooperative-competitive multiagent systems, IEEE Transactions on Cybernetics 52 (12) (2021) 12785–12794.

[62] P.-L. Loh, M. J. Wainwright, Regularized m-estimators with nonconvexity : Statistical and algorithmic theory for local optima, Advances in Neural Information Processing Systems 26 (2013).

[63] K. Kurdyka, T. Mostowski, A. Parusinski, Proof of the gradient conjecture of R. Thom, Annals of Mathematics 152 (3) (2000) 763–792.
URL http://www.jstor.org/stable/2661354

[64] M. Forti, P. Nistri, M. Quincampoix, Convergence of neural networks for programming problems via a nonsmooth Łojasiewicz inequality, IEEE Transactions on Neural Networks 17 (6) (2006) 1471–1486.

[65] Q. Liu, Z. Guo, J. Wang, A one-layer recurrent neural network for constrained pseudoconvex optimization and its application for dynamic portfolio optimization, Neural Networks 26 (2012) 99–109.

[66] J. Bolte, A. Daniilidis, A. Lewis, The Łojasiewicz inequality for nonsmooth subanalytic functions with applications to subgradient dynamical systems, SIAM Journal on Optimization 17 (4) (2007) 1205–1223.

[67] W. Bian, X. Xue, Subgradient-based neural networks for nonsmooth nonconvex optimization problems, IEEE Transactions on Neural Networks 20 (6) (2009) 1024–1038.

[68] S. Qin, X. Yang, X. Xue, J. Song, A one-layer recurrent neural network for pseudoconvex optimization problems with equality and inequality constraints, IEEE transactions on cybernetics 47 (10) (2016) 3063–3074.

[69] Y. Xia, H. Leung, J. Wang, A projection neural network and its application to constrained optimization problems, IEEE Transactions on Circuits and Systems I : Fundamental Theory and Applications 49 (4) (2002) 447–458.

[70] Q. Liu, J. Cao, Y. Xia, A delayed neural network for solving linear projection equations and its analysis, IEEE transactions on neural networks 16 (4) (2005) 834–843.

[71] Q. Liu, J. Wang, A one-layer projection neural network for nonsmooth optimization subject to linear equalities and bound constraints, IEEE Transactions on Neural Networks and Learning Systems 24 (5) (2013) 812–824.

[72] Y. Xia, New cooperative projection neural network for nonlinearly constrained variational inequality, Science in China Series F : Information Sciences 52 (10) (2009) 1766–1777.

[73] X. He, T. Huang, J. Yu, C. Li, C. Li, An inertial projection neural network for solving variational inequalities, IEEE transactions on cybernetics 47 (3) (2016) 809–814.

[74] L. Jin, S. Li, B. Hu, M. Liu, A survey on projection neural networks and their applications, Applied Soft Computing 76 (2019) 533–544.

[75] Y. Xia, J. Wang, A bi-projection neural network for solving constrained quadratic optimization problems, IEEE transactions on neural networks and learning systems 27 (2) (2015) 214–224.

[76] Y. Xia, J. Wang, A recurrent neural network for solving linear projection equations, Neural Networks 13 (3) (2000) 337–350.

[77] X.-B. Gao, L.-Z. Liao, A new projection-based neural network for constrained variational inequalities, IEEE transactions on neural networks 20 (3) (2009) 373–388.

[78] X. Gao, L.-Z. Liao, A novel neural network for generally constrained variational inequalities, IEEE Transactions on Neural Networks and Learning Systems 28 (9) (2016) 2062–2075.

[79] Y. Xia, A cooperative projection neural network for fast solving linear reconstruction problems, in : International Symposium on Neural Networks, Springer, 2017, pp. 511–520.

[80] Q. Liu, T. Huang, J. Wang, One-layer continuous-and discrete-time projection neural networks for solving variational inequalities and related optimization problems, IEEE Transactions on Neural Networks and Learning Systems 25 (7) (2013) 1308–1318.

[81] T. M. Shami, A. A. El-Saleh, M. Alswaitti, Q. Al-Tashi, M. A. Summakieh, S. Mirjalili, Particle swarm optimization : A comprehensive survey, IEEE Access 10 (2022) 10031–10061.

[82] Q. Liu, S. Yang, J. Wang, A collective neurodynamic approach to distributed constrained optimization, IEEE Transactions on Neural Networks and Learning Systems 28 (8) (2016) 1747–1758.

[83] X. Wen, L. Luan, S. Qin, A continuous-time neurodynamic approach and its discretization for distributed convex optimization over multi-agent systems, Neural Networks 143 (2021) 52–65.

[84] S. Yang, Q. Liu, J. Wang, A collaborative neurodynamic approach to multiple-objective distributed optimization, IEEE Transactions on Neural Networks and Learning Systems 29 (4) (2017) 981–992.

[85] S. Qin, X. Le, J. Wang, A neurodynamic optimization approach to bilevel quadratic programming, IEEE transactions on neural networks and learning systems 28 (11) (2016) 2580–2591.

[86] H. Che, J. Wang, A two-timescale duplex neurodynamic approach to biconvex optimization, IEEE Transactions on Neural Networks and Learning Systems 30 (8) (2018) 2503–2514.

[87] H. Che, J. Wang, A two-timescale duplex neurodynamic approach to mixed-integer optimization, IEEE Transactions on Neural Networks and Learning Systems 32 (1) (2020) 36–48.

[88] M.-F. Leung, J. Wang, A collaborative neurodynamic approach to multiobjective optimization, IEEE Transactions on Neural Networks and Learning Systems 29 (11) (2018) 5738–5748.

[89] J. Wang, J. Wang, Q.-L. Han, Neurodynamics-based model predictive control of continuous-time under-actuated mechatronic systems, IEEE/ASME Transactions on Mechatronics 26 (1) (2020) 311–322.

[90] Z. Yan, J. Wang, Nonlinear model predictive control based on collective neurodynamic optimization, IEEE Transactions on Neural Networks and Learning Systems 26 (4) (2015) 840–850.

[91] J. Fan, J. Wang, A collective neurodynamic optimization approach to nonnegative matrix factorization, IEEE transactions on neural networks and learning systems 28 (10) (2016) 2344–2356.

[92] J. Wang, J. Wang, H. Che, Task assignment for multivehicle systems based on collaborative neurodynamic optimization, IEEE Transactions on Neural Networks and Learning Systems 31 (4) (2019) 1145–1154.

[93] J. Zhao, J. Yang, J. Wang, W. Wu, Spiking neural network regularization with fixed and adaptive drop-keep probabilities, IEEE Transactions on Neural Networks and Learning Systems 33 (8) (2021) 4096–4109.

[94] W. Zhou, H.-T. Zhang, J. Wang, Sparse bayesian learning based on collaborative neurodynamic optimization, IEEE Transactions on Cybernetics 52 (12) (2021) 13669–13683.

[95] X. Li, J. Wang, S. Kwong, Hash bit selection based on collaborative neurodynamic optimization, IEEE Transactions on Cybernetics 52 (10) (2021) 11144–11155.

[96] X. Li, J. Wang, S. Kwong, Hash bit selection via collaborative neurodynamic optimization with discrete hopfield networks, IEEE Transactions on Neural Networks and Learning Systems 33 (10) (2021) 5116–5124.

[97] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, J. M. Siskind, Automatic differentiation in machine learning : a survey, Journal of machine learning research 18 (2018).

[98] M. Raissi, A. Yazdani, G. E. Karniadakis, Hidden fluid mechanics : Learning velocity and pressure fields from flow visualizations, Science 367 (6481) (2020) 1026–1030.

[99] L. Lu, X. Meng, Z. Mao, G. E. Karniadakis, DeepXDE: A Deep Learning Library for Solving Differential Equations, SIAM Review 63 (1) (2021) 208–228. doi:10.1137/19m1274067.
URL http://dx.doi.org/10.1137/19M1274067

[100] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, L. Yang, Physics-informed machine learning, Nature Reviews Physics 3 (6) (2021) 422–440.

[101] Y. Chen, L. Lu, G. E. Karniadakis, L. D. Negro, Physics-informed neural networks for inverse problems in nano-optics and metamaterials, Optics Express 28 (8) (2020) 11618.

[102] A. Yazdani, L. Lu, M. Raissi, G. E. Karniadakis, Systems biology informed deep learning for inferring parameters and hidden dynamics, PLOS Computational Biology 16 (11) (2020).

[103] M. Daneker, Z. Zhang, G. E. Karniadakis, L. Lu, Systems biology : Identifiability analysis and parameter identification via systems-biology informed neural networks, arXiv preprint arXiv :2202.01723 (2022).

[104] G. Pang, L. Lu, G. E. Karniadakis, fpinns : Fractional physics-informed neural networks, SIAM Journal on Scientific Computing 41 (4) (2019).

[105] D. Zhang, L. Lu, L. Guo, G. E. Karniadakis, Quantifying total uncertainty in physics-informed neural networks for solving forward and inverse stochastic problems, Journal of Computational Physics 397 (2019) 108850.

[106] A. F. Psaros, K. Kawaguchi, G. E. Karniadakis, Meta-learning pinn loss functions, Journal of Computational Physics 458 (2022) 111121.

[107] J. Yu, L. Lu, X. Meng, G. E. Karniadakis, Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems, Computer Methods in Applied Mechanics and Engineering 393 (2022) 114823.

[108] S. Wang, Y. Teng, P. Perdikaris, Understanding and Mitigating Gradient Flow Pathologies in Physics-Informed Neural Networks, SIAM Journal on Scientific Computing 43 (5) (2021) A3055–A3081. doi:10.1137/20M1318043.
URL https://doi.org/10.1137/20M1318043

[109] S. Wang, X. Yu, P. Perdikaris, When and why pinns fail to train : A neural tangent kernel perspective, Journal of Computational Physics 449 (2022) 110768.

[110] Z. Xiang, W. Peng, X. Liu, W. Yao, Self-adaptive loss balanced physics-informed neural networks, Neurocomputing (2022).

[111] X. Meng, Z. Li, D. Zhang, G. E. Karniadakis, Ppinn : Parareal physics-informed neural networks for time-dependent pdes, Journal of Computational Physics 370 (2020) 113250.

[112] K. Shukla, A. D. Jagtap, G. E. Karniadakis, Parallel physics-informed neural networks via domain decomposition, Journal of Computational Physics 447 (2021) 110683.

[113] A. D. Jagtap, G. E. Karniadakis, Extended physics-informed neural networks (xpinns) : A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations., in : AAAI spring symposium : MLPS, Vol. 10, 2021.

[114] C. L. Wight, J. Zhao, Solving allen-cahn and cahn-hilliard equations using the adaptive physics informed neural networks, arXiv preprint arXiv :2007.04542 (2020).

[115] A. Krishnapriyan, A. Gholami, S. Zhe, R. Kirby, M. W. Mahoney, Characterizing possible failure modes in physics-informed neural networks, Advances in Neural Information Processing Systems 34 (2021) 26548–26560.

[116] R. Mattey, S. Ghosh, A novel sequential method to train physics informed neural networks for allen cahn and cahn hilliard equations, Computer Methods in Applied Mechanics and Engineering 390 (2022) 114474.

[117] K. Haitsiukevich, A. Ilin, Improved training of physics-informed neural networks with model ensembles, arXiv preprint arXiv :2204.05108 (2022).

[118] S. Wang, S. Sankaran, P. Perdikaris, Respecting causality is all you need for training physics-informed neural networks, arXiv preprint arXiv :2203.07404 (2022).

[119] L. Lu, R. Pestourie, W. Yao, Z. Wang, F. Verdugo, S. G. Johnson, Physics-Informed Neural Networks with Hard Constraints for Inverse Design, SIAM Journal on Scientific Computing 43 (6) (2021) B1105–B1132. doi:10.1137/21M1397908.
URL https://doi.org/10.1137/21M1397908

[120] P. L. Lagari, L. H. Tsoukalas, S. Safarkhani, I. E. Lagaris, Systematic construction of neural forms for solving partial differential equations inside rectangular domains, subject to initial, boundary and interface conditions, International Journal of Artificial Intelligence Tools 29 (05) (2020) 2050009.

[121] S. Dong, N. Ni, A method for representing periodic functions and enforcing exactly periodic boundary conditions with deep neural networks, Journal of Computational Physics 435 (2021) 110242.

[122] W. E, B. Yu, The Deep Ritz Method: A Deep Learning-Based Numerical Algorithm for Solving Variational Problems, Communications in Mathematics and Statistics 6 (1) (2018) 1–12. doi:10.1007/s40304-018-0127-z.
URL https://doi.org/10.1007/s40304-018-0127-z

[123] E. Samaniego, C. Anitescu, S. Goswami, V. M. Nguyen-Thanh, H. Guo, K. Hamdia, X. Zhuang, T. Rabczuk, An energy approach to the solution of partial differential equations in computational mechanics via machine learning : Concepts, implementation and applications, Computer Methods in Applied Mechanics and Engineering 362 (2020) 112790.

[124] J. N. Fuhg, C. Böhm, N. Bouklas, A. Fau, P. Wriggers, M. Marino, Model-data-driven constitutive responses : Application to a multiscale computational framework, International Journal of Engineering Science (2021). doi:10.1016/J.IJENGSCI.2021.103522.

[125] J. He, D. Abueidda, S. Koric, I. Jasiuk, On the use of graph neural networks and shape-function-based gradient computation in the deep energy method, International Journal for Numerical Methods in Engineering 124 (4) (2023) 864–879.

[126] D. W. Abueidda, S. Koric, R. A. Al-Rub, C. M. Parrott, K. A. James, N. A. Sobh, A deep learning energy method for hyperelasticity and viscoelasticity, European Journal of Mechanics-A/Solids 95 (2022) 104639.

[127] J. He, S. Kushwaha, C. Chadha, S. Koric, D. W. Abueidda, I. Jasiuk, Deep energy method in topology optimization applications, ArXiv (2022). doi:10.48550/ARXIV.2207.03072.

[128] D. W. Abueidda, S. Koric, E. Guleryuz, N. A. Sobh, Enhanced physics-informed neural networks for hyperelasticity, International Journal for Numerical Methods in Engineering 124 (7) (2023) 1585–1601.

[129] E. Haghighat, M. Raissi, A. Moure, H. Gomez, R. Juanes, A physics-informed deep learning framework for inversion and surrogate modeling in solid mechanics, Computer Methods in Applied Mechanics and Engineering (2021). doi:10.1016/J.CMA.2021.113741.

[130] L. Lu, P. Jin, G. Pang, Z. Zhang, G. E. Karniadakis, Learning nonlinear operators via deeponet based on the universal approximation theorem of operators, Nature Machine Intelligence (2021). doi:10.1038/S42256-021-00302-5.

[131] V. Oommen, K. Shukla, S. Goswami, R. Dingreville, G. E. Karniadakis, Learning two-phase microstructure evolution using neural operators and autoencoder architectures, npj Computational Materials 8 (1) (2022) 190.

[132] L. Tan, L. Chen, Enhanced deeponet for modeling partial differential operators considering multiple input functions, arXiv.org (2022).

[133] S. Wang, H. Wang, P. Perdikaris, Improved architectures and training algorithms for deep operator networks, Journal of Scientific Computing (2021). doi:10.1007/S10915-022-01881-0.

[134] S. Koric, D. Abueidda, Data-driven and physics-informed deep learning operators for solution of heat conduction equation with parametric heat source, International Journal of Heat and Mass Transfer (2023). doi:10.1016/J.IJHEATMASSTRANSFER.2022.123809.

[135] S. Koric, A. Viswantah, D. Abueidda, N. Sobh, K. Khan, Deep learning operator network for plastic deformation with variable loads and material properties, Engineering computations (2023). doi:10.1007/S00366-023-01822-X.

[136] M. Yin, E. Zhang, Y. Yu, G. E. Karniadakis, Interfacing finite elements with deep neural operators for fast multiscale modeling of mechanics problems, Computer methods in applied mechanics and engineering 402 (2022) 115027.

[137] S. Goswami, M. Yin, Y. Yu, G. E. Karniadakis, A physics-informed variational deeponet for predicting crack path in quasi-brittle materials, Computer Methods in Applied Mechanics and Engineering 391 (2022) 114587.

[138] S. Goswami, K. Kontolati, M. Shields, G. Karniadakis, Deep transfer learning for partial differential equations under conditional shift with deeponet, ArXiv (2022). doi:10.48550/ARXIV.2204.09810.

[139] V. Kumar, S. Goswami, D. J. Smith, G. E. Karniadakis, Real-time prediction of multiple output states in diesel engines using a deep neural operator framework, arXiv preprint arXiv :2304.00567 (2023).

[140] D. Bertsimas, B. Stellato, The voice of optimization, Machine Learning 110 (2021) 249–277.

[141] D. Bertsimas, B. Stellato, Online mixed-integer optimization in milliseconds, INFORMS Journal on Computing 34 (4) (2022) 2229–2248.

[142] K. Gregor, Y. LeCun, Learning fast approximations of sparse coding, in : Proceedings of the 27th international conference on international conference on machine learning, 2010, pp. 399–406.

[143] X. Chen, J. Liu, Z. Wang, W. Yin, Theoretical linear convergence of unfolded ista and its practical weights and thresholds, Advances in Neural Information Processing Systems 31 (2018).

[144] J. Liu, X. Chen, Alista : Analytic weights are as good as learned weights in lista, in : International Conference on Learning Representations (ICLR), 2019.

[145] D. Wu, A. Lisser, Predicting nash equilibria in bimatrix games using a robust bi-channel convolutional neural network, IEEE Transactions on Artificial Intelligence (2023) 1–13doi:10.1109/TAI.2023.3321584.

[146] D. Wu, A. Lisser, Neuro-pinn : A hybrid framework for efficient nonlinear projection equation solutions, International Journal for Numerical Methods in Engineeringdoi:https://doi.org/10.1002/nme.7377.

[147] D. Wu, A. Lisser, Enhancing neurodynamic approach with physics-informed neural networks for solving non-smooth convex optimization problems, Neural Networks (2023). doi:https://doi.org/10.1016/j.neunet.2023.08.014.
URL https://www.sciencedirect.com/science/article/pii/S0893608023004331

[148] D. Wu, A. Lisser, Ccgnet : A deep learning approach to predict nash equilibrium of chance-constrained games, Information Sciences 627 (2023) 20–33.

[149] D. Bertsimas, C. W. Kim, A prescriptive machine learning approach to mixed-integer convex optimization, INFORMS Journal on Computing (2023).

[150] A. Cauligi, P. Culbertson, B. Stellato, D. Bertsimas, M. Schwager, M. Pavone, Learning mixed-integer convex optimization strategies for robot planning and control, in : 2020 59th IEEE Conference on Decision and Control (CDC), IEEE, 2020, pp. 1698–1705.

[151] D. Bertsimas, C. W. Kim, A machine learning approach to two-stage adaptive robust optimization, arXiv preprint arXiv :2307.12409 (2023).

[152] V. Monga, Y. Li, Y. C. Eldar, Algorithm unrolling : Interpretable, efficient deep learning for signal and image processing, IEEE Signal Processing Magazine 38 (2) (2021) 18–44. doi:10.1109/MSP.2020.3016905.

[153] T. Blumensath, M. E. Davies, Iterative Thresholding for Sparse Approximations, Journal of Fourier Analysis and Applications 14 (5) (2008) 629–654. doi:10.1007/s00041-008-9035-z.
URL https://doi.org/10.1007/s00041-008-9035-z

[154] A. Beck, M. Teboulle, A fast iterative shrinkage-thresholding algorithm for linear inverse problems, SIAM journal on imaging sciences 2 (1) (2009) 183–202.

[155] Z. Wang, Q. Ling, T. Huang, Learning deep 0 encoders, in : Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 30, 2016.

[156] Z. Wang, D. Liu, S. Chang, Q. Ling, Y. Yang, T. S. Huang, D3 : Deep dual-domain based fast restoration of jpeg-compressed images, in : Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 2764–2772.

[157] J. v. Neumann, Zur Theorie der Gesellschaftsspiele, Mathematische Annalen 100 (1) (1928) 295–320. doi:10.1007/BF01448847.

[158] K. C. Border, Others, Fixed Point Theorems with Applications to Economics and Game Theory, Cambridge Books (1990).

[159] N. Karmarkar, A new polynomial-time algorithm for linear programming, in : Proceedings of the sixteenth annual ACM symposium on Theory of computing, 1984, pp. 302–311.

[160] M. J. Todd, The many facets of linear programming, Mathematical programming 91 (3) (2002) 417–436.

[161] S. Minaee, Y. Boykov, F. Porikli, A. Plaza, N. Kehtarnavaz, D. Terzopoulos, Image segmentation using deep learning : A survey, IEEE transactions on pattern analysis and machine intelligence 44 (7) (2021) 3523–3542.

[162] K. Han, Y. Wang, H. Chen, X. Chen, J. Guo, Z. Liu, Y. Tang, A. Xiao, C. Xu, Y. Xu, A survey on vision transformer, IEEE transactions on pattern analysis and machine intelligence 45 (1) (2022) 87–110.

[163] D. W. Otter, J. R. Medina, J. K. Kalita, A survey of the usages of deep learning for natural language processing, IEEE transactions on neural networks and learning systems 32 (2) (2020) 604–624.

[164] Q. Guo, F. Zhuang, C. Qin, H. Zhu, X. Xie, H. Xiong, Q. He, A survey on knowledge graph-based recommender systems, IEEE Transactions on Knowledge and Data Engineering 34 (8) (2020) 3549–3568.

[165] I. Goodfellow, Y. Bengio, A. Courville, Y. Bengio, I. G. Courville, Y. Bengio, Aaron, Deep learning, Nature 29 (7553) (2016) 1–73.
URL http://www.deeplearningbook.org

[166] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in : Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Vol. 2016-Decem, 2016, pp. 770–778. arXiv:1512.03385, doi:10.1109/CVPR.2016.90.

[167] G. Huang, Z. Liu, L. Van Der Maaten, K. Q. Weinberger, Densely connected convolutional networks, in : Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Vol. 2017-Janua, 2017, pp. 2261–2269. arXiv:1608.06993, doi:10.1109/CVPR.2017.243.

[168] M. Tan, Q. Le, Efficientnet : Rethinking model scaling for convolutional neural networks, in : International conference on machine learning, PMLR, 2019, pp. 6105–6114.

[169] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamar´, M. A. Fadhel, M. Al-Amidie, L. Farhan, Review of deep learning : Concepts, CNN architectures, challenges, applications, future directions, Journal of big Data 8 (1) (2021) 1–74.

[170] Y. Xu, H. Zhang, Convergence of deep convolutional neural networks, Neural Networks (2022).

[171] A. Khan, A. Sohail, U. Zahoora, A. S. Qureshi, A survey of the recent architectures of deep convolutional neural networks, Artificial Intelligence Review 53 (8) (2020) 5455–5516. doi:10.1007/s10462-020-09825-6.
URL https://doi.org/10.1007%2Fs10462-020-09825-6

[172] R. Song, W. Zhang, Y. Zhao, Y. Liu, Unsupervised multi-view CNN for salient view selection and 3D interest point detection, International Journal of Computer Vision 130 (5) (2022) 1210–1227.

[173] F. Logothetis, R. Mecca, I. Budvytis, R. Cipolla, A cnn based approach for the point-light photometric stereo problem, International Journal of Computer Vision (2022) 1–20.

[174] W. Hackbusch, Multi-grid methods and applications, Vol. 4, Springer Science & Business Media, 2013.

[175] H. Gao, L. Sun, J.-X. Wang, PhyGeoNet : Physics-informed geometry-adaptive convolutional neural networks for solving parameterized steady-state PDEs on irregular domain, Journal of Computational Physics 428 (2021) 110079.

[176] J. He, J. Xu, MgNet : A unified framework of multigrid and convolutional neural network, Science china mathematics 62 (2019) 1331–1354.

[177] Y. Chen, B. Dong, J. Xu, Meta-mgnet : Meta multigrid networks for solving parameterized partial differential equations, Journal of Computational Physics 455 (2022) 110996.

[178] L. Khachiyan, E. Boros, K. Borys, V. Gurvich, K. Elbassioni, Generating all vertices of a polyhedron is hard, in : Twentieth Anniversary Volume :, Springer, 2009, pp. 1–17.

[179] D. Wu, A. Lisser, Using cnn for solving two-player zero-sum games, Expert Systems with Applications (2022) 117545.

[180] D. Wu, A. Lisser, Mg-cnn : A deep cnn to predict saddle points of matrix games, Neural Networks (2022).

[181] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, Pytorch : An imperative style, high-performance deep learning library, in : Advances in Neural Information Processing Systems 32, Curran Associates, Inc., 2019, pp. 8024–8035.

[182] V. Knight, J. Campbell, Nashpy : A Python library for the computation of Nash equilibria, Journal of Open Source Software 3 (30) (2018) 904.

[183] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant, Array programming with NumPy, Nature 585 (7825) (2020) 357–362. doi:10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2

[184] H. Tsaknakis, P. G. Spirakis, An optimization approach for approximate nash equilibria, Internet Mathematics 5 (4) (2008) 365–382.

[185] A. Deligkas, M. Fasoulakis, E. Markakis, A polynomial-time algorithm for 1/3-approximate nash equilibria in bimatrix games, arXiv preprint arXiv :2204.11525 (2022).

[186] D. P. Bertsekas, Nonlinear programming, Journal of the Operational Research Society 48 (3) (1997) 334.

[187] S. M. Robinson, Normal maps induced by linear transformations, Mathematics of Operations Research 17 (3) (1992) 691–714.

[188] R. L. Burden, J. D. Faires, A. M. Burden, Numerical analysis, Cengage learning, 2015.

[189] Q. Liu, J. Wang, A one-layer recurrent neural network with a discontinuous hard-limiting activation function for quadratic programming, IEEE transactions on neural networks 19 (4) (2008) 558–570.

[190] Z. Guo, Q. Liu, J. Wang, A one-layer recurrent neural network for pseudoconvex optimization subject to linear equality constraints, IEEE Transactions on Neural Networks 22 (12) (2011) 1892–1900.

[191] W. Jia, N. Liu, S. Qin, An Adaptive Continuous-Time Algorithm for Nonsmooth Convex Resource Allocation Optimization, IEEE Transactions on Automatic Control 67 (11) (2022) 6038–6044. doi:10.1109/TAC.2021.3137054.

[192] N. Liu, J. Wang, S. Qin, A one-layer recurrent neural network for nonsmooth pseudoconvex optimization with quasiconvex inequality and affine equality constraints, Neural Networks 147 (2022) 1–9.

[193] I. E. Lagaris, A. Likas, D. I. Fotiadis, Artificial neural networks for solving ordinary and partial differential equations, IEEE Transactions on Neural Networks 9 (5) (1998) 987–1000. doi:10.1109/72.712178.

[194] K. S. McFall, J. R. Mahan, Artificial neural network method for solution of boundary value problems with exact satisfaction of arbitrary boundary conditions, IEEE Transactions on Neural Networks 20 (8) (2009) 1221–1233.

[195] J. Han, A. Jentzen, W. E, Solving high-dimensional partial differential equations using deep learning, Proceedings of the National Academy of Sciences 115 (34) (2018) 8505–8510.

[196] S. Huang, W. Feng, C. Tang, J. Lv, Partial Differential Equations Meet Deep Neural Networks : A Survey, arXiv preprint arXiv :2211.05567 (2022).

[197] Z. Mao, A. D. Jagtap, G. E. Karniadakis, Physics-informed neural networks for high-speed flows, Computer Methods in Applied Mechanics and Engineering 360 (2020) 112789.

[198] S. Cai, Z. Mao, Z. Wang, M. Yin, G. E. Karniadakis, Physics-informed neural networks (PINNs) for fluid mechanics : A review, Acta Mechanica Sinica (2022) 1–12.

[199] D. W. Abueidda, Q. Lu, S. Koric, Meshless physics-informed deep learning method for three-dimensional solid mechanics, International Journal for Numerical Methods in Engineering 122 (23) (2021) 7182–7201.

[200] Y. Ghaffari Motlagh, P. K. Jimack, R. de Borst, Deep learning phase-field model for brittle fractures, International Journal for Numerical Methods in Engineering (2022).

[201] D. Zhang, L. Guo, G. E. Karniadakis, Learning in Modal Space: Solving Time-Dependent Stochastic PDEs Using Physics-Informed Neural Networks, SIAM Journal on Scientific Computing 42 (2) (2020) A639–A665. doi:10.1137/19M1260141.
URL https://doi.org/10.1137/19M1260141

[202] R. Sharma, V. Shankar, Accelerated Training of Physics Informed Neural Networks (PINNs) using Meshless Discretizations, arXiv preprint arXiv :2205.09332 (2022).

[203] S. Rezaei, A. Harandi, A. Moeineddin, B. X. Xu, S. Reese, A mixed formulation for physics-informed neural networks as a potential solver for engineering problems in heterogeneous domains: Comparison with finite element method, Computer Methods in Applied Mechanics and Engineering 401 (2022) 115616. arXiv:2206.13103, doi:10.1016/j.cma.2022.115616.
URL https://doi.org/10.1016/j.cma.2022.115616

[204] J. He, D. Abueidda, R. A. Al-Rub, S. Koric, I. Jasiuk, A deep learning energy-based method for classical elastoplasticity, International Journal of Plasticity 162 (2023) 103531.

[205] F. Chen, D. Sondak, P. Protopapas, M. Mattheakis, S. Liu, D. Agarwal, M. D. Giovanni, Neurodiffeq: A python package for solving differential equations with neural networks, Journal of Open Source Software 5 (46) (2020) 1931. doi:10.21105/joss.01931.
URL https://doi.org/10.21105/joss.01931

[206] M. Mattheakis, D. Sondak, A. S. Dogra, P. Protopapas, Hamiltonian neural networks for solving equations of motion, arXiv preprint arXiv :2001.11107 (2020).

[207] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, Q. Zhang, Jax: composable transformations of python+numpy programs (2023).
URL http://github.com/google/jax

[208] J. R. Dormand, P. J. Prince, A family of embedded runge-kutta formulae, Journal of computational and applied mathematics 6 (1) (1980) 19–26.

[209] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, SciPy 1.0 Contributors, SciPy 1.0 : Fundamental

Algorithms for Scientific Computing in Python, Nature Methods 17 (2020) 261–272. doi:10.1038/s41592-019-0686-2.

[210] A. D. Jagtap, E. Kharazmi, G. E. Karniadakis, Conservative physics-informed neural networks on discrete domains for conservation laws : Applications to forward and inverse problems, Computer Methods in Applied Mechanics and Engineering 365 (2020) 113028.

[211] A. D. Jagtap, K. Kawaguchi, G. E. Karniadakis, Adaptive activation functions accelerate convergence in deep and physics-informed neural networks, Journal of Computational Physics 404 (2020) 109136.

[212] T. De Ryck, S. Lanthaler, S. Mishra, On the approximation of functions by tanh neural networks, Neural Networks 143 (2021) 732–750.

[213] S. Mishra, R. Molinaro, Estimates on the generalization error of physics-informed neural networks for approximating a class of inverse problems for pdes, IMA Journal of Numerical Analysis 42 (2) (2022) 981–1022.

[214] P. Jain, P. Kar, Others, Non-convex optimization for machine learning, Foundations and Trends®in Machine Learning 10 (3-4) (2017) 142–363.

[215] B. Jiang, T. Lin, S. Ma, S. Zhang, Structured nonconvex and nonsmooth optimization : algorithms and iteration complexity analysis, Computational Optimization and Applications 72 (1) (2019) 115–157.

[216] Y. Carmon, J. C. Duchi, O. Hinder, A. Sidford, Lower bounds for finding stationary points i, Mathematical Programming 184 (1-2) (2020) 71–120.

[217] Y. Arjevani, Y. Carmon, J. C. Duchi, D. J. Foster, N. Srebro, B. Woodworth, Lower bounds for non-convex stochastic optimization, Mathematical Programming 199 (1-2) (2023) 165–214.

[218] M. Mäkelä, Survey of bundle methods for nonsmooth optimization, Optimization methods and software 17 (1) (2002) 1–29.

[219] X. Xue, W. Bian, Subgradient-based neural networks for nonsmooth convex optimization problems, IEEE Transactions on Circuits and Systems I : Regular Papers 55 (8) (2008) 2378–2391.

[220] S. Qin, W. Bian, X. Xue, A new one-layer recurrent neural network for nonsmooth pseudoconvex optimization, Neurocomputing 120 (2013) 655–662.

[221] S. Qin, X. Xue, A two-layer recurrent neural network for nonsmooth convex optimization problems, IEEE transactions on neural networks and learning systems 26 (6) (2014) 1149–1160.

[222] J. Liu, X. Liao, A projection neural network to nonsmooth constrained pseudoconvex optimization, IEEE Transactions on Neural Networks and Learning Systems (2021).

[223] Y. Wang, J. Wang, D. Tao, Neurodynamics-driven supervised feature selection, Pattern Recognition 136 (2023) 109254.

[224] Z.-G. Hou, L. Cheng, M. Tan, X. Wang, Distributed adaptive coordinated control of multi-manipulator systems using neural networks, Robot intelligence : An advanced knowledge processing approach (2010) 49–69.

[225] A. Domahidi, E. Chu, S. Boyd, ECOS : An SOCP solver for embedded systems, in : 2013 European Control Conference (ECC), IEEE, 2013, pp. 3071–3076.

[226] O. Brendan, C. Eric, P. Neal, B. Stephen, Operator Splitting for Conic Optimization via Homogeneous Self-Dual Embedding, Journal of Optimization Theory and Applications 169 (3) (2016) 1042–1068. doi:10.1007/s10957-016-0892-3.
URL https://doi.org/10.1007/s10957-016-0892-3

[227] S. Diamond, S. Boyd, CVXPY : A Python-embedded modeling language for convex optimization, Journal of Machine Learning Research 17 (83) (2016) 1–5.

[228] T. G. Grossmann, U. J. Komorowska, J. Latz, C.-B. Schönlieb, Can physics-informed neural networks beat the finite element method ?, arXiv preprint arXiv :2302.04107 (2023).

[229] Z. Li, N. B. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. M. Stuart, A. Anandkumar, Fourier neural operator for parametric partial differential equations, in : 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021, OpenReview.net, 2021.
URL https://openreview.net/forum?id=c8P9NQVtmnO

[230] C. Flamant, P. Protopapas, D. Sondak, Solving differential equations using neural network solution bundles (2020). arXiv:2006.14372.

[231] U. Ravat, U. V. Shanbhag, On the characterization of solution sets of smooth and nonsmooth convex stochastic nash games, SIAM Journal on Optimization 21 (3) (2011) 1168–1199.

[232] H. Jiang, U. V. Shanbhag, S. P. Meyn, Distributed computation of equilibria in misspecified convex stochastic nash games, IEEE Transactions on Automatic Control 63 (2) (2017) 360–371.

[233] H. N. Nguyen, A. Lisser, V. V. Singh, Random games under elliptically distributed dependent joint chance constraints, Journal of Optimization Theory and Applications 195 (1) (2022) 249–264.

[234] J. J. Hopfield, D. W. Tank, "neural" computation of decisions in optimization problems, Biological cybernetics 52 (3) (1985) 141–152.

[235] A. Nazemi, A. Sabeghi, A new neural network framework for solving convex second-order cone constrained variational inequality problems with an application in multi-finger robot hands, Journal of Experimental & Theoretical Artificial Intelligence 32 (2) (2020) 181–203.

[236] Y.-W. Lv, G.-H. Yang, C.-X. Shi, Differentially private distributed optimization for multi-agent systems via the augmented lagrangian algorithm, Information Sciences 538 (2020) 39–53.

[237] J. Zou, R. Sun, S. Yang, J. Zheng, A dual-population algorithm based on alternative evolution and degeneration for solving constrained multi-objective optimization problems, Information Sciences 579 (2021) 89–102.

[238] Z. Wang, J. Liu, D. Wang, W. Wang, Distributed cooperative optimization for multiple heterogeneous euler-lagrangian systems under global equality and inequality constraints, Information Sciences 577 (2021) 449–466.

[239] C. Xu, Q. Liu, T. Huang, Resilient penalty function method for distributed constrained optimization under byzantine attack, Information Sciences 596 (2022) 362–379.

[240] C.-X. Shi, G.-H. Yang, Distributed nash equilibrium computation in aggregative games : An event-triggered algorithm, Information Sciences 489 (2019) 289–302.

[241] D. Wu, A. Lisser, A dynamical neural network approach for solving stochastic two-player zero-sum games, Neural Networks (2022).

[242] D. Wu, A. Lisser, Improved saddle point prediction in stochastic two-player zero-sum games with a deep learning approach, Engineering Applications of Artificial Intelligence 126 (2023) 106664.

[243] M. Raza, S. S. Khan, M. Ali, Deep Learning for Computer Vision : A Comprehensive Review, IEEE Access 9 (2021) 62530–62558.

[244] L. Tan, X. Zhang, Deep learning for natural language processing : A review, IEEE Access 8 (2020) 138913–138931.

[245] Y. Hu, H. Chen, F. Zhuang, Deep Learning in Bioinformatics : A Comprehensive Survey, Briefings in Bioinformatics 21 (3) (2020) 742–761.

[246] D. Wu, A. Lisser, A deep learning approach for solving linear programming problems, Neurocomputing (2022).

[247] V. Nair, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, P. Wang, et al., Solving mixed integer programs using neural networks, arXiv preprint arXiv :2012.13349 (2020).

[248] Y. Bengio, A. Lodi, A. Prouvost, Machine learning for combinatorial optimization : a methodological tour d'horizon, European Journal of Operational Research 290 (2) (2021) 405–421.

[249] M. W. M. G. Dissanayake, N. Phan-Thien, Neural-network-based approximations for solving partial differential equations, Communications in Numerical Methods in Engineering 10 (3) (1994) 195–201. doi:https://doi.org/10.1002/cnm.1640100303.

URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cnm.1640100303

[250] B. C. Eaves, On the basic theorem of complementarity, Mathematical Programming 1 (1) (1971) 68–75. doi:10.1007/BF01584073.
URL https://doi.org/10.1007/BF01584073

[251] L. Petzold, Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations, SIAM journal on scientific and statistical computing 4 (1) (1983) 136–148.

[252] L. F. Shampine, M. W. Reichelt, The matlab ode suite, SIAM journal on scientific computing 18 (1) (1997) 1–22.

[253] E. Hairer, S. P. Nørsett, G. Wanner, Solving ordinary differential equations. 1, Nonstiff problems, Springer-Vlg, 1993.