A deep learning approach for solving linear programming problems

Dawen Wu, Abdel Lisser

A deep learning approach for solving linear programming problems

Dawen Wu[a] (dawen.wu@centralesupelec.fr), Abdel Lisser[a] (abdel.lisser@l2s.centralesupelec.fr)

[a] Université Paris-Saclay, CNRS, CentraleSupélec, Laboratoire des signaux et systèmes, 91190, Gif-sur-Yvette, France

Corresponding Author:

Dawen Wu

Address: Université Paris-Saclay, CNRS, CentraleSupélec, Laboratoire des signaux et systèmes, 91190, Gif-sur-Yvette, France

Tel: (+33) 750798387

Email: dawen.wu@centralesupelec.fr

A deep learning approach for solving linear programming problems

Dawen Wu[a,*], Abdel Lisser[a]

[a] *Université Paris-Saclay, CNRS, CentraleSupélec, Laboratoire des signaux et systèmes, 91190, Gif-sur-Yvette, France*

Abstract

Finding the optimal solution to a linear programming (LP) problem is a long-standing computational problem in Operations Research. This paper proposes a deep learning approach in the form of feed-forward neural networks to solve the LP problem. The latter is first modeled by an ordinary differential equations (ODE) system, the state solution of which globally converges to the optimal solution of the LP problem. A neural network model is constructed as an approximate state solution to the ODE system, such that the neural network model contains the prediction of the LP problem. Furthermore, we extend the capability of the neural network by taking the parameter of LP problems as an input variable so that one neural network can solve multiple LP instances in a one-shot manner. Finally, we validate the proposed method through a collection of specific LP examples and show concretely how the proposed method solves the example.

Keywords: Linear programming, ODE systems, Neural networks, Deep learning

1. Introduction

Linear programs are a class of optimization problems where the goal is to minimize a linear objective subject to linear constraints. They play an important role in operations research and have a wide range of real-world applications, such as transportation scheduling, manufacturing planning, and resource allocation.

We consider the LP problem of the following general form

$$\begin{cases} \min\limits_{\mathbf{x}} \mathbf{D}^T\mathbf{x} \\ \text{s.t.} \\ \quad \mathbf{A}\mathbf{x} - \mathbf{b} \leq 0, \end{cases} \tag{1}$$

where $\mathbf{A} \in \mathbb{R}^{m\times n}$, $\mathbf{b} \in \mathbb{R}^m$ and $\mathbf{D} \in \mathbb{R}^n$ are problem data. $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{u} \in \mathbb{R}^m$ are decision variables and dual variables, respectively. n and m represent the number of decision variables and the number of constraints, respectively.

This paper proposes a deep learning approach to solve LP problems, which composes of two parts. 1) The path followed by the decision and dual variables of an LP problem toward the optimal solution is modeled

*Corresponding author
Email address: dawen.wu@centralesupelec.fr, abdel.lisser@l2s.centralesupelec.fr (Abdel Lisser)

by an ODE system. 2) A feed-forward neural network is used as an approximate state solution to the ODE system.

1.1. Literature review

The LP problems have many applications in areas such as economics, engineering, and computer science (Dantzig, 1963; Gass, 2003). The simplex and interior-point methods are two classical and still widely used solution methods, that have been well developed and extensively studied in the last decades (Karmarkar, 1984; Nocedal & Wright, 2006; Gondzio, 2012). Academic or commercial software implement these two methods for user convenience, such as Cvxpy, Matlab, and Gurobi (Diamond & Boyd, 2016; Matlab, 2017; Gurobi Optimization, LLC, 2021).

The connection between ODE systems and optimization problems was first studied by Hopfield & Tank (1985), where the LP problem is modeled by the Hopfield network. Since then, ODE systems approaches have been applied to various optimization problems, including quadratic programming (Xia, 1996; Effati & Nazemi, 2006), linear complementarity problems (Li-Zhi & Hou-Duo, 1999; Xia & Wang, 2000), second-order cone programming(Ko et al., 2011; Nazemi & Sabeghi, 2020), nonlinear projection equations (Xia & Feng, 2007) and stochastic game problems (Wu & Lisser, 2022). These articles mentioned above use an ODE system to model the optimization problem, and then the optimal solution is obtained by solving the ODE system. These ODE systems are shown to have global convergence properties, i.e., the state solutions converge to the optimal solution as the time variable goes to infinity.

The numerical integration methods for solving ODE systems can be divided into two categories: explicit and implicit methods (Teschl, 2012). The explicit method determines the later state based on the current state, e.g., RK45, RK23 (Dormand & Prince, 1980; Bogacki & Shampine, 1989). The implicit method integrate the ODE system by solving equations involving the current and later states, e.g., Radau and BDF (Wanner & Hairer, 1996; Shampine & Reichelt, 1997). Software has been develop to implement these methods (Virtanen et al., 2020; Rackauckas & Nie, 2017).

Dissanayake & Phan-Thien (1994) first uses a neural network to approximate the solution of differential equations, where the loss function contains two terms satisfying the initial/boundary condition and satisfying the differential equation. Lagaris et al. (1998) develop a trail solution method that can ensure initial conditions always be satisfied. Flamant et al. (2020) takes the parameter of ODE system models as an input variable to the neural network, such that a neural network can be the solution of multiple differential equations, namely solution bundles. The universal approximation theorem is foundational for this line of research, which states that a neural network can approximate any continuous function to arbitrary accuracy (Cybenko, 1989; Hornik et al., 1989). Python packages are developed to implement these methods (Lu et al., 2021; Chen et al., 2020).

With the rapid growth of computing power and data, there has been a large amount of research on neural networks (Goodfellow et al., 2016). Neural networks have a large number of applications in computer vision, natural language processing, data mining, and other fields (Voulodimos et al., 2018; Deng & Liu, 2018; Liu et al., 2017). Most training methods for neural networks are based on gradient descent, and many variants

have been proposed to speed up training or to improve the generalization ability (Kingma & Ba, 2017; Nair & Hinton, 2010). Extreme learning machine is a new paradigm for training neural networks that do not involve any gradient-based iterative process (Huang et al., 2006; Zhang et al., 2020; Xiao et al., 2017). This method can apply to a wide range of deep learning tasks. One of our contributions is to link the LP problem to deep learning (DL) so that the LP problem can benefit from a wide range of research in the DL community, e.g., the extreme learning machine. In addition, our paper is expected to inspire new solution method to problems beyond optimization, e.g., optimal control problems (Zhang et al., 2021), or to be an integral part of existing methods (Djordjevic et al., 2022; Zhuang et al., 2022).

1.2. Contributions

The contributions of this paper are threefold and summarized as follows.

- We propose a deep learning approach to solve LP problems. The proposed approach uses neurodynamic optimization to model the LP problem by an ODE system. A neural network model is then trained to be an approximate state solution of the ODE system. In this way, the LP problem can be solved only by deep learning, involving neither optimization methods, e.g., simplex or interior point methods, nor numerical integration methods, e.g., RK45.
- We increase the model capacity by treating the parameter of LP problems as an input variable to the neural network. The neural network then becomes the solution to multiple LP instances. A novel algorithm is proposed to train such a neural network. Once trained, the model is able to solve multiple LP instances in a one-shot manner without any iterative process.
- In the experimental Section, we concretely show how to solve a specific LP instance by the proposed method. We explicitly present the LP instance, the corresponding ODE system, and the neural network model. We compare our proposed approach with the RK45 and convex solvers..

1.3. Organization

The remaining sections are organized as follows. Section 2 describes how to model an LP problem by an ODE system, and demonstrate the effectiveness of the ODE system from theoretical point of view. Section 3 sketches the case of using one neural network to solve one LP problem. Section 4 shows the case of using one neural network to solve multiple LP problems. Section 5 presents experimental results and discussions. Section 6 summarizes this paper and gives future directions.

2. Preliminaries

2.1. Karush-Kuhn-Tucker conditions

We consider the LP problem as presented in (1). The corresponding Lagrangian is

$$L(\mathbf{x}, \mathbf{u}) = \mathbf{D}^T\mathbf{x} + \mathbf{u}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) \tag{2}$$

The corresponding Karush-Kuhn-Tucker (KKT) conditions are

$$\begin{aligned} \mathbf{D} + \mathbf{A}^T\mathbf{u} &= 0, \\ \mathbf{u}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) &= 0, \\ \mathbf{A}\mathbf{x} - \mathbf{b} &\leq 0, \\ \mathbf{u} &\geq 0. \end{aligned} \tag{3}$$

For the LP problem, the KKT conditions are necessary and sufficient for optimality, i.e., the optimal solution $\mathbf{x}^*$ of the LP problem (1) can be obtained by finding a solution $(\mathbf{x}^*, \mathbf{u}^*)$ of the KKT condition (3).

2.2. ODE system for modeling LP

Let $\mathbf{x}(t) : \mathbb{R} \to \mathbb{R}^n$ and $\mathbf{u}(t) : \mathbb{R} \to \mathbb{R}^m$ be functions with respect to the time variable t, and $\mathbf{y}(t) = (\mathbf{x}^T(t), \mathbf{u}^T(t))^T$. The following ODE system models the LP problem (1) via the KKT conditions (3).

$$\frac{d\mathbf{y}}{dt} = \Phi(\mathbf{y}) = \begin{bmatrix} \frac{d\mathbf{x}}{dt} \\ \frac{d\mathbf{u}}{dt} \end{bmatrix} = \begin{bmatrix} -\left(\mathbf{D} + \mathbf{A}^T(\mathbf{u} + \mathbf{A}\mathbf{x} - \mathbf{b})^+\right) \\ (\mathbf{u} + \mathbf{A}\mathbf{x} - \mathbf{b})^+ - \mathbf{u} \end{bmatrix}. \tag{4}$$

The initial value problem (IVP) is developed by composing of the ODE system (4), an initial condition $\mathbf{y}(t_0) = \mathbf{y}_0$ and a time range $[0, T]$, i.e.,

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} -\left(\mathbf{D} + \mathbf{A}^T(\mathbf{u} + \mathbf{A}\mathbf{x} - \mathbf{b})^+\right) \\ (\mathbf{u} + \mathbf{A}\mathbf{x} - \mathbf{b})^+ - \mathbf{u} \end{bmatrix}, \quad \mathbf{y}(t_0) = \mathbf{y}_0, \quad t \in [0, T] \tag{5}$$

$\mathbf{y}(t), t \in [0, T]$ is called the state solution of the initial value problem (5) if it satisfies both the ODE system and the initial condition. $\mathbf{y}^*$ is called an equilibrium point of the ODE system (4) if it satisfies $\Phi(\mathbf{y}^*) = \mathbf{0}$. The following theorems and definition show how the ODE system (4) solves the LP problem (1).

Theorem 1 (Nazemi & Omidi (2013))**.** *$\mathbf{y}^* = (\mathbf{x}^*, \mathbf{u}^*)^T$ is an equilibrium point of the ODE system (4) if and only if it is a satisfied point of the KKT conditions (3).*

Definition 1. *Let $y(t)$ be a solution of the ODE system $\frac{d\mathbf{y}}{dt} = \Phi(\mathbf{y})$. Then the ODE system is said to be globally convergent to the solution set $\Theta^* = \{\mathbf{y}^* \mid \mathbf{y}^* = (\mathbf{x}^*, \mathbf{u}^*)$ solves (3)$\}$ if any state solution with an arbitrary initial point satisfies*

$$\lim_{t \to \infty} \operatorname{dist}(\mathbf{y}(t), \Theta^*) = 0,$$

where $\operatorname{dist}(\mathbf{y}, \Theta^*) = \inf_{\mathbf{y}^* \in \Theta^*} \|\mathbf{y} - \mathbf{y}^*\|$ and $\|\cdot\|$ denotes the l_2-norm.

Theorem 2 (Nazemi & Omidi (2013))**.** *The ODE system (4) is globally convergent to the solution set $\Theta^* = \{\mathbf{y}^* \mid \mathbf{y}^* = (\mathbf{x}^*, \mathbf{u}^*)$ solves (3)$\}$.*

According to Theorem 1, the optimal solution to the LP problem is equivalent to the equilibrium points of the corresponding ODE system. According to Theorem 2, for any given initial point, the state solution of

the ODE system will always converge to an optimal solution of the LP problem as the time t goes to infinity, i.e., $\lim_{t\to\infty} \mathbf{y}(t) = \mathbf{y}^*$. We refer the reader to (Nazemi & Omidi, 2013; Nazemi & Tahmasbi, 2013; Xia & Wang, 2000) for the proof of the theorems and other details.

3. One neural network for solving one LP problem

Before using the neural network, the ODE system requires two parameters to form an IVP, i.e., the initial point $\mathbf{y}_0$ and the time range $[0, T]$. Thanks to the global convergence property of the ODE system, the initial point can be simply fixed to zero without loss of convergence, i.e.,

$$\mathbf{y}(0) = \mathbf{0}. \tag{6}$$

Let the neural network model be defined as

$$\hat{\mathbf{y}}\left(t; \mathbf{w}\right) = (1 - e^{-t})\mathbf{NN}\left(t; \mathbf{w}\right), \quad t \in [0, T], \tag{7}$$

where $\mathbf{NN}$ represents a fully-connected neural network with weights $\mathbf{w}$. The multiplier $(1 - e^{-t})$ is used to guarantee the initial condition $\hat{\mathbf{y}}(0; \mathbf{w}) = 0$ must be satisfied regardless of weights $\mathbf{w}$. The multiplier greatly affects the region around $t = 0$ and gradually converges to 1 as time goes on. The neural network model $\hat{\mathbf{y}}\left(t; \mathbf{w}\right)$ is act as an approximate state solution to the IVP. The endpoint of the model (7),

$$\hat{\mathbf{y}}\left(T; \mathbf{w}\right) = (1 - e^{-T})\mathbf{NN}\left(T; \mathbf{w}\right), \tag{8}$$

represents an approximate solution to the KKT conditions.

Figure 1 shows the workflow of solving an LP problem by the neural network model. Consider solving LP problem (1), and the solution procedure can be summarised as follows. 1) The KKT conditions (3) of the LP problem are derived; The objective becomes finding a feasible point to the KKT conditions. 2) The ODE system (4) is built to model the KKT conditions; By considering the initial point (6) and a time range $[0, T]$, the ODE system forms an initial value problem. 3) The model $\hat{\mathbf{y}}\left(t; \mathbf{w}\right)$ as defined in (7) is used as an approximate state solution to this initial value problem; The endpoint $\hat{\mathbf{y}}\left(T; \mathbf{w}\right)$ is an approximate solution to the KKT conditions.

Since the initial condition $\hat{\mathbf{y}}(0; \mathbf{w}) = 0$ is always satisfied thanks to the multiplier construction, The training objective is to make $\hat{\mathbf{y}}(t; \mathbf{w})$ satisfying the corresponding ODE system by adjusting model's weights $\mathbf{w}$. Thus, the loss function is defined as

$$E(\mathbf{w}) = \frac{1}{|\mathbb{T}|} \sum_{t_i \in \mathbb{T}} \ell\left(\frac{\partial \hat{\mathbf{y}}\left(t_i; \mathbf{w}\right)}{\partial t_i}, \Phi(\hat{\mathbf{y}}\left(t_i; \mathbf{w}\right))\right), \tag{9}$$

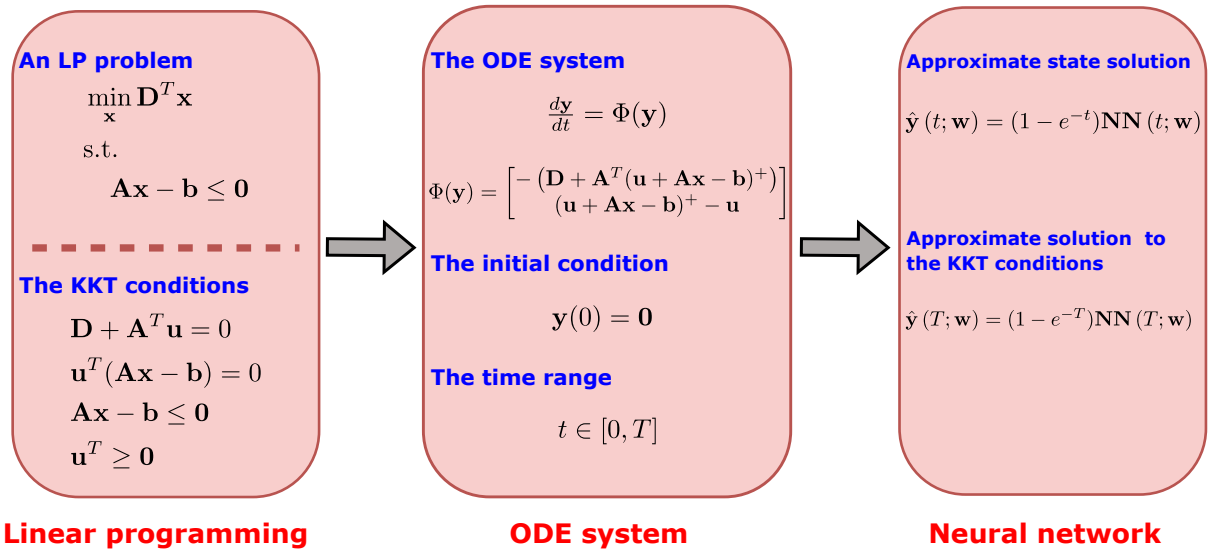


Figure 1: **The connection between the LP problem, ODE system and neural network.**

where Φ refers to the ODE system associated with the LP problem to be solved. $\mathbb{T}\subset\mathbb{R}_+$ refers to a set of collocation times to be trained, where each collocation time t_i is sampled by a uniform distribution over the interval $[0,T]$. $\frac{\partial\hat{\mathbf{y}}(t_i;\mathbf{w})}{\partial t_i}\in\mathbb{R}^{n+m}$ represents the partial derivative of $\hat{\mathbf{y}}$ with respect to t_i. $\Phi(\hat{\mathbf{y}}(t_j;\mathbf{w}))\in\mathbb{R}^{n+m}$ represents the partial derivative expected to be according to the ODE system. $\ell(\cdot,\cdot)$ is an error metric used to measure the difference between these two vectors.

Algorithm 1: Solving an LP problem by one neural network

Require: A time range $[0,T]$
Input : An LP problem
Output : Solution to the LP problem
Function Main**:**
  Derive the ODE system Φ of the LP problem.
  Construct a neural network model $\hat{\mathbf{y}}(t;\mathbf{w})=(1-e^{-t})\mathbf{NN}(t;\mathbf{w})$.
  while True **do**
    $\mathbb{T}\sim U(0,T)$ Uniformly sample a batch of t from the interval $[0,T]$
    Forward propagation: Compute $E(\mathbf{w})$ based on $\mathbb{T}$
    Backward propagation: Train the model weight $\mathbf{w}$ by $\nabla E(\mathbf{w})$
    Stopping criteria check.
  end
end

Algorithm 1 shows the training of the neural network model to solve an LP problem. The algorithm requires a time range $[0,T]$, which determines the input space of the model. The LP problem in hand should determine the choice of T, e.g., a larger T for a complex problem. In each training round, the neural network trains on a batch of time $\mathbb{T}$, which would is by the end of the iteration.

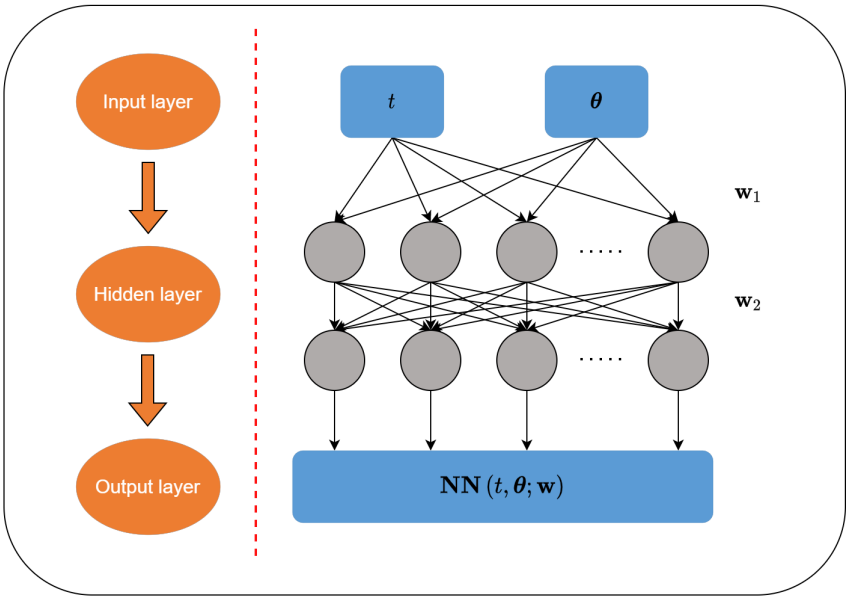


Figure 2: **The neural network model that receives t and $\boldsymbol{\theta}$ as input.** t refers to the time variable. $\boldsymbol{\theta}$ refers to the parameter of an LP problem, and $\mathbf{w}$ refers to the weights of the neural network.

The training algorithm is unsupervised since there is no pre-given fixed training dataset. Instead, the ODE system is given as an instruction to guide the neural network model. Since each round of training uses fresh data, there is no over-fitting to a particular training data set. Training can go on forever to constantly optimize the neural network's weights $\mathbf{w}$. However, it is realistic to set a stopping criteria to break the loop when the training is deemed good enough. Some possible stopping criteria could be: stop when the number of iterations exceeds a maximum; stop when the loss function or other metrics fall below a certain threshold.

After the training, the neural network model $\hat{\mathbf{y}}\,(t; \mathbf{w})$ is expected to be well approximate the true state solutiong $\mathbf{y}(t)$ on interval $[0, T]$. Furthermore, the endpoint $\hat{\mathbf{y}}(T; \mathbf{w}) = (\hat{\mathbf{x}}, \hat{\mathbf{u}})$ is expected to solve the KKT conditions, where $\hat{\mathbf{x}}$ is the final prediction to the LP problem.

4. One neural network for solving multiple LP problems

Section 3 presents the case of using one neural network model to solve one LP problem. This section demonstrates how to solve multiple LP problems using a single neural network model, significantly increasing computational efficiency.

Let the neural network model for solving multiple LP problems be defined as

$$\hat{\mathbf{y}}\,(t; \boldsymbol{\theta}; \mathbf{w}) = (1 - e^{-t})\mathbf{NN}\,(t, \boldsymbol{\theta}; \mathbf{w})\,, \tag{10}$$

where $\mathbf{NN}\,(t, \boldsymbol{\theta}; \mathbf{w})$ is a nerual network that receive the parameter $\boldsymbol{\theta}$ of an LP problem as input, as shown in Figure 2. The parameter $\boldsymbol{\theta}$ represents the problem data, such as $\mathbf{D}$, $\mathbf{A}$, $\mathbf{b}$ in (1) or a combination of them, and each different $\boldsymbol{\theta}$ represents a different LP problem. The model $\hat{\mathbf{y}}\,(t, \boldsymbol{\theta}; \mathbf{w})$ receive different LP problem

with a parameter $\boldsymbol{\theta}$ as input and predict their solutions. As in (7), the initial point is fixed to $\mathbf{y}(0) = \mathbf{0}$, and $(1 - e^{-t})$ ensures the initial conditions are satisfied. When considering only one LP problem, the parameter $\boldsymbol{\theta}$ is a constant and can be removed, and the model (10) degraded to (7).

The loss function of (10) is defined as

$$E(\mathbf{w}) = \frac{1}{|\Theta| * |\mathbb{T}|} \sum_{\boldsymbol{\theta}_j \in \Theta} \sum_{t_i \in \mathbb{T}} \ell \left(\frac{\partial \hat{\mathbf{y}}\left(t_i, \boldsymbol{\theta}_j; \mathbf{w}\right)}{\partial t_i}, \Phi_j(\hat{\mathbf{y}}\left(t_i, \boldsymbol{\theta}_j; \mathbf{w}\right)) \right), \tag{11}$$

where Θ refers to the set of parameters $\boldsymbol{\theta}$, and each $\boldsymbol{\theta}_j \in \Theta$ relate to an ODE system Φ_j. For a specific $\boldsymbol{\theta}_j$, $\frac{\partial \hat{\mathbf{y}}(t_i, \boldsymbol{\theta}_j; \mathbf{w})}{\partial t_j} \in \mathbb{R}^{n+m}$ and $\Phi_j(\hat{\mathbf{y}}\left(t_i, \boldsymbol{\theta}_j; \mathbf{w}\right)) \in \mathbb{R}^{n+m}$ represent the partial derivative of $\hat{\mathbf{y}}$ with respect to t_i and the derivative from the ODE system Φ_j, respectively.

Algorithm 2: Solving multiple LP problems by one neural network

Require: A time range: $[0, T]$
A probability distribution P for generating $\boldsymbol{\theta}$.

Function Main**:**
  while True **do**
    $\Theta \sim \mathrm{P}$ Generate a set of $\boldsymbol{\theta}$ according to the given probability distribution P
    $\mathbb{T} \sim U(0, T)$ Uniformly sample a batch of t from the interval $[0, T]$
    Forward propagation: Compute $E(\mathbf{w})$ based on the Θ and $\mathbb{T}$
    Backward propagation: Update $\mathbf{w}$ by $\nabla E(\mathbf{w})$
    Stopping criteria check.
  end
end

Algorithm 2 shows the training of the neural network model involving multiple LP problems. The algorithm aims at reducing the loss function (11). At each training iteration, a batch of data Θ and $\mathbb{T}$ is sampled to train the model, where Θ is sampled from the given probability distribution P, and $\mathbb{T}$ is sampled from the given time range $[0, T]$.

After trainings, the model $\hat{\mathbf{y}}\left(t, \boldsymbol{\theta}; \mathbf{w}\right)$ is expected to be able to solve LP problems with parameter $\boldsymbol{\theta}$ coming from the distribution P. Consider a LP problem with parameter $\boldsymbol{\theta}$; The model's final prediction to this LP problem is $\hat{y}(T, \boldsymbol{\theta}; \mathbf{w})$.

5. Numerical results

Section 5.1 gives the experimental setup, including the environment, hyperparameters, and evaluation metrics. Section 5.2 shows the case of solving a specific LP instance, as introduced in Section 3 whilst Section 5.3 shows the case of solving multiple LP instances, as introduced in Section 4.

5.1. Experimental setup

The experiments are conducted on the Google Colab Pro+ platform, which provides 51GB RAM, eight Intel Xeon 2.2GHz CPUs, and one Nvidia Tesla P100 16GB GPU. The neural network model is built by Pytorch 1.9.1 and CUDA 10.2.

The hyperparameters used for training the neural network are as follows

- The optimizer is Adam with a learning rate of 0.001.
- The loss function $\ell(\cdot, \cdot)$ is the mean square error.
- The batch size is 128, and the maximum number of iterations is 500000.

We compare the proposed approach to the ODE solver and the convex solvers. For the ODE solver, we select the RK45 method implemented by Scipy (Dormand & Prince, 1980; Virtanen et al., 2020). For the convex solvers, we select the ECOS, OSQP, and Gurobi provided by CVXPY (Domahidi et al., 2013; Stellato et al., 2020; Gurobi Optimization, LLC, 2021; Diamond & Boyd, 2016).

We use the following mean square error (MSE) metric to measure the prediction accuracy, defined by

$$\text{MSE} = \frac{1}{B}\sum_{i=1}^{B}(f_i(\mathbf{x}_i^*) - f_i(\hat{\mathbf{x}}_i))^2, \tag{12}$$

where B refers to the number of instances in a test batch. Consider the i-th instance in the test batch. $\mathbf{x}_i^*$ refers to an optimal solution, and $f_i(\mathbf{x}_i^*)$ refers to the true objective value obtained by the LP solver. $\mathbf{x}_i$ refers to a prediction, and $f_i(\mathbf{x}_i)$ refers to the predicted objective value. The evaluation indicator measure on the objective value $f_i(\hat{\mathbf{x}}_i)$ rather than on the predicted point $\hat{\mathbf{x}}_i$, because the true objective value is unique $f_i(\mathbf{x}_i^*)$, whereas there may be multiple optimal solutions $\mathbf{x}_i^*$. We also provide the mean objective value (MOV) to roughly describe a test batch, defined as

$$\text{MOV} = \frac{1}{B}\sum_{i=1}^{B} f_i(\mathbf{x}_i). \tag{13}$$

5.2. Case 1: Solving one LP problem

Consider the following specific LP problem

$$\begin{aligned} &\min_x -9.54x_1 - 8.16x_2 - 4.26x_3 - 11.43x_4 \\ &\text{s.t.} \\ &\quad 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 \leq 7.81. \end{aligned} \tag{14}$$

According to (4), the corresponding ODE system for this LP problem is

$$\frac{d\mathbf{y}}{dt} = \Phi(\mathbf{y}) = \begin{bmatrix} \frac{d\mathbf{x}}{dt} \\ \frac{du}{dt} \end{bmatrix} = \begin{bmatrix} -\left(-9.54 + 3.18^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - 7.81)^+\right) \\ -\left(-8.16 + 2.72^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - 7.81)^+\right) \\ -\left(-4.26 + 1.42^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - 7.81)^+\right) \\ -\left(-11.43 + 3.81^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - 7.81)^+\right) \\ (u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - 7.81)^+ - u \end{bmatrix}, \tag{15}$$

where u is the dual variable.

By choosing the initial point to be all-zero and the time range to be $[0, 10]$, the ODE system forms the initial value problem as follows

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} -\left(-9.54 + 3.18^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - 7.81)^+\right) \\ -\left(-8.16 + 2.72^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - 7.81)^+\right) \\ -\left(-4.26 + 1.42^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - 7.81)^+\right) \\ -\left(-11.43 + 3.81^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - 7.81)^+\right) \\ (u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - 7.81)^+ - u \end{bmatrix}, \quad \mathbf{y}(0) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad t \in [0, 10]. \tag{16}$$

Layers	Input Size	Output Size	Activation function
Fully connected	1	100	Tanh
Fully connected	100	5	

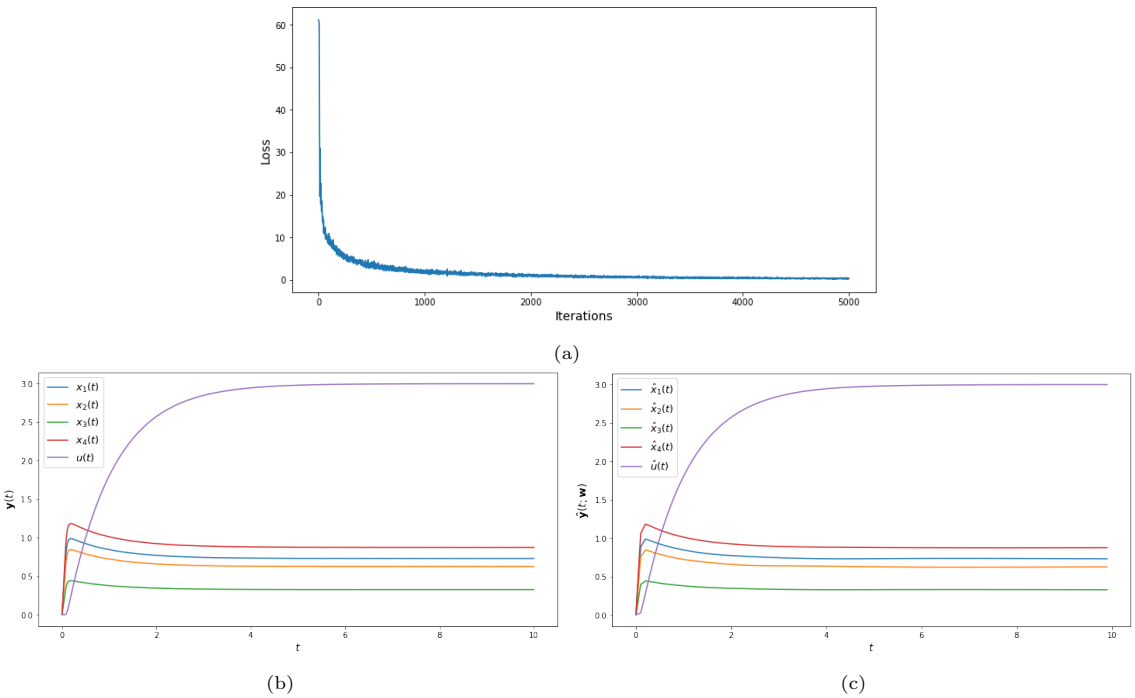
Table 1: **A two-layer neural network to solve the LP problem** (14)**.** The input is $t \in [0, 10]$, and the second layer of the neural network has no activation function.

We construct a model $\hat{\mathbf{y}}(t; \mathbf{w})$ as defined in (7) to solve this initial value problem and the LP problem, where Table 1 shows the structure of the neural network. As shown in Figure 3a, the loss value decreases from an initial value of 60 to less than 0.1, implying that the model is trained to satisfy the ODE system conditions(15). Figure 3b shows the true state solution obtained by the numerical integration method RK45, and Figure 3c shows the approximate state solution , i.e., the model $\hat{\mathbf{y}}(t; \mathbf{w})$ after training. The approximate state solution $\hat{\mathbf{y}}(t; \mathbf{w})$ is almost identical to the true state solution $\mathbf{y}(t)$, which validates the effectiveness of the training.

Approaches	$\mathbf{x}^*$	u^*
Convex optimization solver: ECOS	$[0.7294, 0.6238, 0.3257, 0.8739]$	2.9999
Numerical integration method: RK45	$[0.7294, 0.6238, 0.3257, 0.8739]$	2.9999
Neural network	$[0.7294, 0.6238, 0.3257, 0.8739]$	2.9999

Table 2: **Solutions of three different approaches to the LP problem** (14)

Table 2 shows the final solutions to the LP problem by the three approaches. The solution from the numerical integration method is the state solution's endpoint, i.e., $\mathbf{y}(10)$, and the solution of our neural



(a) (b) (c)

Figure 3: **(a) The loss values in the first 5000 iterations (b) The true state solution $\mathbf{y}(t)$ to the initial value problem** (16) **(c) The neural network approximate state solution $\hat{\mathbf{y}}(t;\mathbf{w})$ after training**

network method is $\hat{\mathbf{y}}(10;\mathbf{w})$. All three approaches achieve the same optimal solutions.

5.3. *Case 2: Solving multiple LP problems*

5.3.1. *b as variable*

We now use a neural network to solve multiple LP problems. Consider an LP problem with the following form

$$\begin{aligned} &\min_x -9.54x_1 - 8.16x_2 - 4.26x_3 - 11.43x_4 \\ &\text{s.t.} \\ &3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 \le b, \end{aligned} \tag{17}$$

where b is a variable within the interval $[0, 10]$. Every different $b \in [0, 10]$ of (17) represents a different LP problem. The initial value problem corresponding to (17) is

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} -\left(9.54 + 3.18^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - b)^+\right) \\ -\left(8.16 + 2.72^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - b)^+\right) \\ -\left(4.26 + 1.42^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - b)^+\right) \\ -\left(11.43 + 3.81^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - b)^+\right) \\ (u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - b)^+ - u \end{bmatrix}, \quad \mathbf{y}(0) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad t \in [0, 10]. \tag{18}$$

Layers	Input Size	Output Size	Activation function
Fully connected	2	100	Tanh
Fully connected	100	5	

Table 3: **A two-layer neural network to solve the LP problem of variable** b (17) The input is $t \in [0, 10]$ and $b \in [0, 10]$

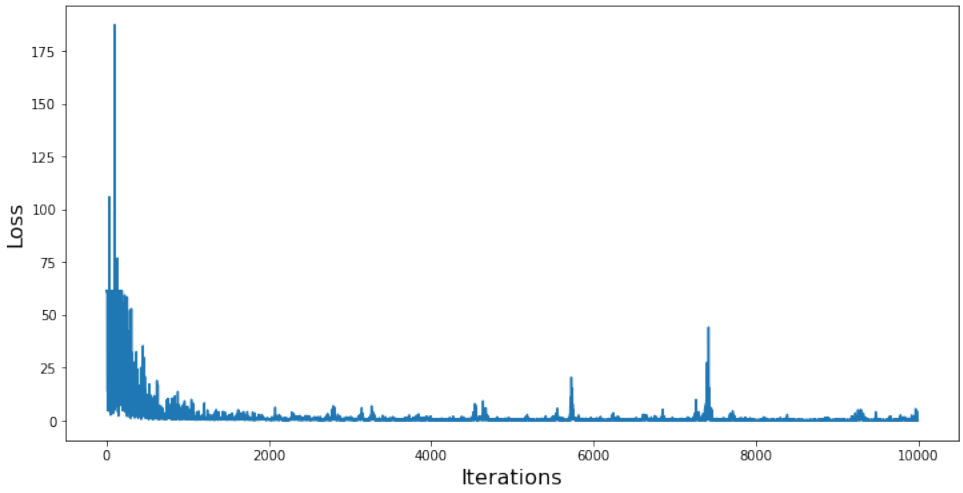


Figure 4: **The loss value versus iteration in the first 10,000 iterations**

We construct the neural network model as follows

$$\hat{\mathbf{y}}(t, b; \mathbf{w}) = (1 - e^{-t})\mathbf{NN}(t, b; \mathbf{w}), \tag{19}$$

which is able to receive various $b \in [0, 10]$ as input, and $\mathbf{NN}(t, b; \mathbf{w})$ is a fully connected network as shown in Table (3). Figure 4 shows the training of this model according to Algorithm 2. After 10,000 iterations, the loss value drops from 175 to less than 5, and after 500,000 iterations, it gradually converges to 0.01. Figure 5 shows the solutions of the neural network for three specific instances after the training is completed.

Numbers of instances	Neural network			ODE solver: RK45		Convex solvers			
	MOV	MSE	CPU time	MOV	CPU time	MOV	ECOS CPU time	OSQP CPU time	GUROBI CPU time
100	-16.90	0.089	**<=0.01**	-16.68	3.45	-16.69	0.37	0.61	0.46
300	-14.83	0.099	**<=0.01**	-14.57	10.34	-14.58	1.10	1.74	1.29
500	-14.52	0.101	**<=0.01**	-14.27	17.38	-14.28	1.89	2.93	2.25
1000	-15.40	0.095	**<=0.01**	-15.16	34.50	-15.16	3.74	5.90	4.36
3000	-15.22	0.095	**<=0.01**	-14.98	104.35	-14.99	10.97	17.45	13.26
5000	-15.39	0.095	**<=0.01**	-15.15	173.35	-15.16	18.60	29.28	22.13
10000	-15.25	0.095	**<=0.01**	-15.01	346.25	-15.02	37.28	57.99	44.73

Table 4: **Comparison between the neural network model, ODE solver and convex solvers on solving multiple instances of** (17) The metric MOV is defined in (13). The error metric MSE is defined in (12), where the MSE for the ODE solver and convex solvers are omitted because they provide exact solutions. The CPU time is in seconds, and <= 0.01 means the CPU times are less than or equal to 0.01 seconds.

Table 4 reports our neural network model's CPU times and accuracy along with two comparable ap-

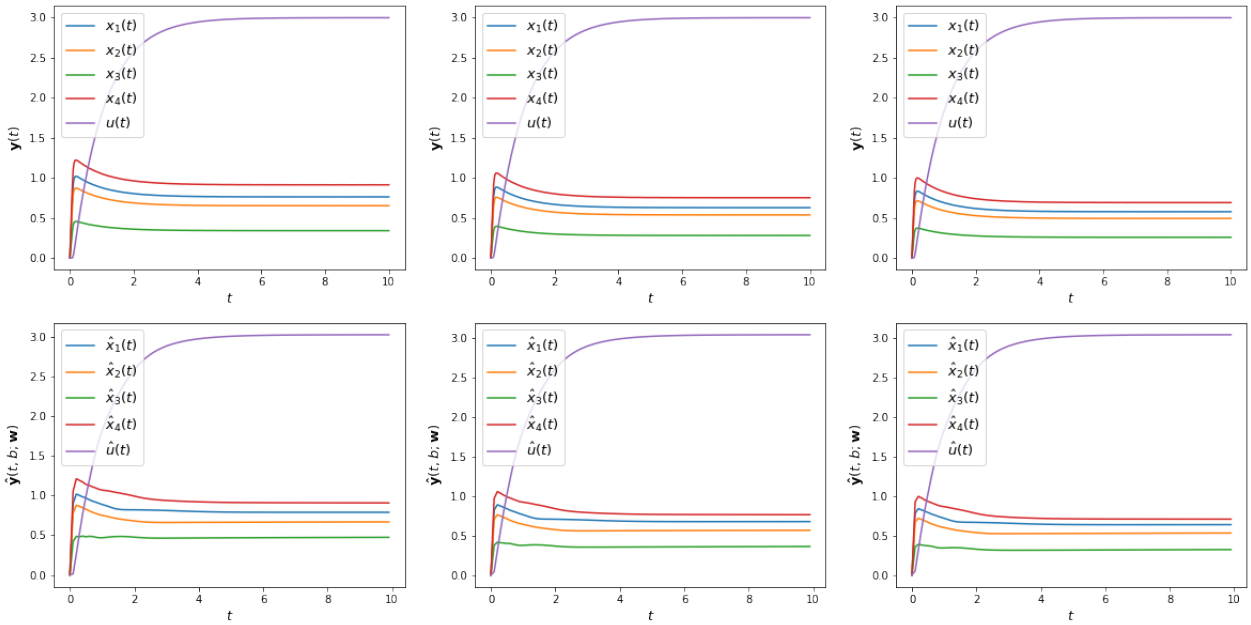


Figure 5: **Three instances of** (17) **solved by the RK45 method and the neural network**
Upper row: True state solutions obtained by the RK45 method
Lower row: Approximate state solutions obtained by the neural network
Left: For solving the LP problem of (17) with $b = 8.16$, the endpoints are $\mathbf{y}(t = 10) = [0.76, 0.65, 0.34, 0.91, 3.00]$, $\hat{\mathbf{y}}(t = 10, b = 8.16; \mathbf{w}) = [0.79, 0.67, 0.47, 0.91, 3.03]$
Middle: For solving the LP problem of (17) with $b = 6.72$, $\mathbf{y}(t = 10) = [0.63, 0.53, 0.28, 0.75, 3.00]$, $\hat{\mathbf{y}}(t = 10, b = 6.72; \mathbf{w}) = [0.67, 0.56, 0.36, 0.76, 3.04]$
Right: For solving the LP problem of (17) with $b = 6.18$, $\mathbf{y}(t = 10) = [0.58, 0.49, 0.26, 0.69, 3.00]$, $\hat{\mathbf{y}}(t = 10, b = 6.18; \mathbf{w}) = [0.64, 0.53, 0.32, 0.71, 3.04]$

proaches. As shown in the table, our neural network model significantly outperforms the ODE solver and convex solvers in terms of computational CPU time. The neural network model takes less than 0.01 seconds to solve either one or multiple instances. The MOVs that our model produces are close to the MOVs from the other two approaches, and the MSEs near 0.1 demonstrate that our model's predictions are acceptable.

After training, the neural network model can directly give the prediction $\hat{\mathbf{y}}(T, b; \mathbf{w})$ to any LP problem of $b \in [0, 10]$ in constant execute time, whereas an ODE solver must first compute all of the previous states before providing the final solution. When dealing with multiple instances, the ODE solver and convex solvers solve them one by one, and the computational time grows linearly with the number of instances. Our neural network can solve all these instances in one shot without suffering from this linear growth. The computational advantage becomes even more significant when the number of instances is large. Additionally, both in the training and prediction stages, our approach is highly conducive to parallelization.

5.3.2. **D** *as variable*

Consider an LP problem with the following form

$$\begin{aligned} &\min_x \mathbf{D}^T x \\ &\text{s.t.} \\ &\quad 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 \leq 7.81, \end{aligned} \tag{20}$$

where $\mathbf{D} = [D_1, D_2, D_3, D_4]^T \in \mathbb{R}^4$ is generated by a random number from the uniform distribution $U(0, 10)$ multiplied by the vector $[3.18, 2.72, 1.43, 3.81]^T$. Every different $\mathbf{D}$ represents a different LP problem. The corresponding initial value problem is

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} -\left(-D_1 + 3.18^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - 7.81)^+\right) \\ -\left(-D_2 + 2.72^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - 7.81)^+\right) \\ -\left(-D_3 + 1.42^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - 7.81)^+\right) \\ -\left(-D_4 + 3.81^T(u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - 7.81)^+\right) \\ (u + 3.18x_1 + 2.72x_2 + 1.42x_3 + 3.81x_4 - 7.81)^+ - u \end{bmatrix}, \quad \mathbf{y}(0) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad t \in [0, 10]. \tag{21}$$

Layers	Input Size	Output Size	Activation function
Fully connected	5	100	Tanh
Fully connected	100	5	

Table 5: **A two-layer neural network to solve the LP problem of variable D** (20) The input is $t \in [0, 10]$ and $\mathbf{D} \in \mathbb{R}^4$

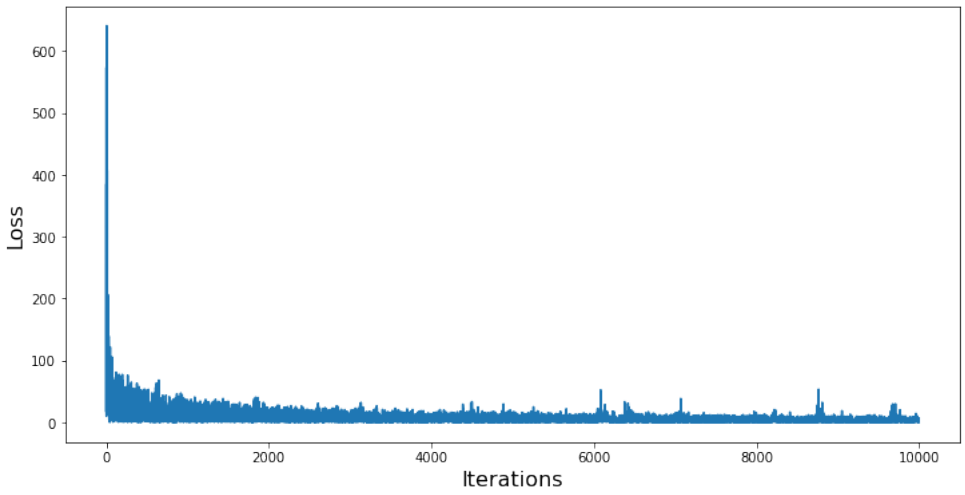


Figure 6: **The loss value versus iteration in the first 10,000 iterations**

We construct the neural network model for resolving this case as follows

$$\hat{\mathbf{y}}(t, \mathbf{D}; \mathbf{w}) = (1 - e^{-t})\mathbf{NN}(t, \mathbf{D}; \mathbf{w}), \tag{22}$$

which is able to receive various $\mathbf{D}$ as input, and $\mathbf{NN}(t, \mathbf{D}; \mathbf{w})$ is a fully connect network as shown in Table (5). Figure 6 shows the training of this model according to Algorithm 2. After 10,000 iterations, the loss value decreases from 600 to less than 10.

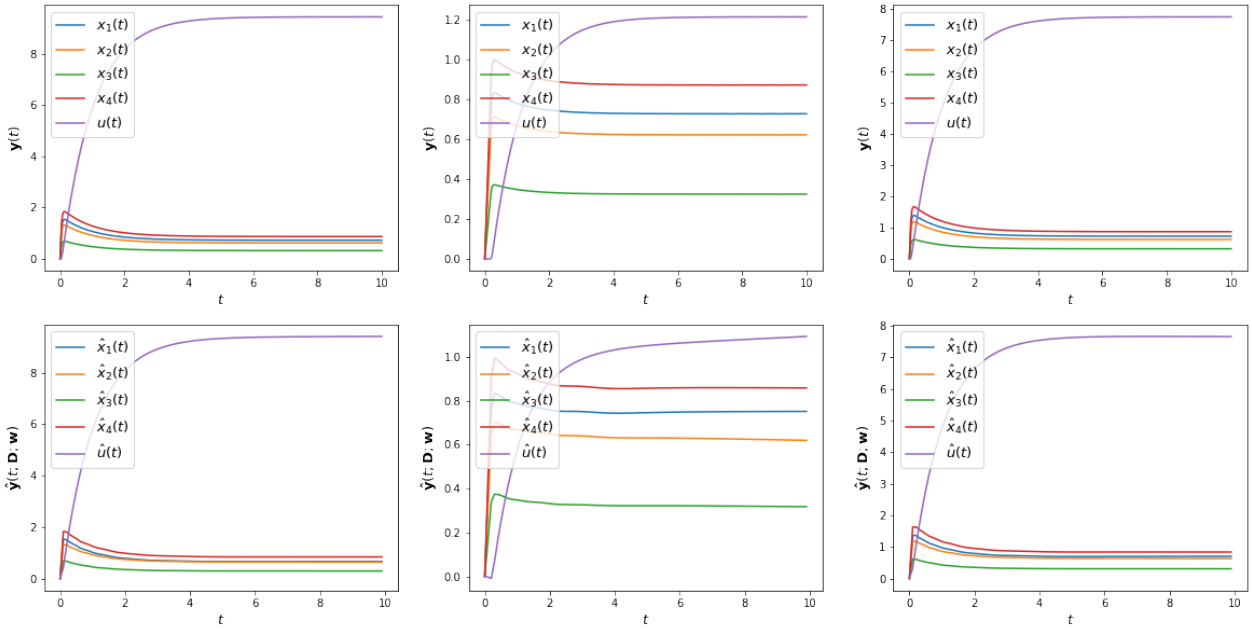


Figure 7: **Three instances of** (20) **solved by the RK45 method and the neural network**
Upper row: True state solutions obtained by the RK45 method
Lower row: Approximate state solutions obtained by the neural network
Left: For solving the LP problem of (20) with $\mathbf{D} = [-29.99, -25.66, -13.39, -35.94]$, the endpoints are $\mathbf{y}(t = 10) = [0.73, 0.62, 0.33, 0.87, 9.43]$, $\hat{\mathbf{y}}(t = 10, \mathbf{D}; \mathbf{w}) = [0.66, 0.64, 0.29, 0.84, 9.41]$
Middle: For solving the LP problem of (20) with $\mathbf{D} = [-3.87, -3.31, -1.73, -4.63]$, the endpoints are $\mathbf{y}(t = 10) = [0.73, 0.62, 0.33, 0.87, 1.22]$, $\hat{\mathbf{y}}(t = 10, \mathbf{D}; \mathbf{w}) = [0.75, 0.62, 0.32, 0.86, 1.09]$.
Right: For solving the LP problem of (20) with $\mathbf{D} = [-24.65, -21.09, -11.01, -29.54]$, the endpoints are $\mathbf{y}(t = 10) = [0.73, 0.62, 0.33, 0.87, 7.75]$, $\hat{\mathbf{y}}(t = 10, \mathbf{D}; \mathbf{w}) = [0.70, 0.63, 0.31, 0.84, 7.66]$

Numbers of linear programs	Neural network			ODE solver		Convex solvers			
	Mean values	Times	MSEs	Mean values	Times	Mean values	ECOS Times	OSQP Times	GUROBI Times
100	-37.95	**<=0.01**	0.17	-37.75	3.64	-37.75	0.38	0.61	0.45
300	-38.84	**<=0.01**	0.17	-38.65	11.13	-38.66	1.12	1.81	1.39
500	-39.10	**<=0.01**	0.20	-38.89	18.64	-38.90	1.92	3.02	2.29
1000	-39.94	**<=0.01**	0.20	-39.72	37.39	-39.73	3.81	6.16	4.51
3000	-38.95	**<=0.01**	0.20	-38.74	111.60	-38.75	11.73	18.01	13.70
5000	-39.32	**<=0.01**	0.20	-39.10	185.89	-39.11	19.13	30.11	22.63
10000	-39.67	**<=0.01**	0.20	-39.44	370.22	-39.45	38.33	60.13	45.44

Table 6: **Comparison between the neural network model, ODE solver and convex solvers on solving multiple instances of** (20) The metric MOV is defined in (13). The error metric MSE is defined in (12), where the MSE for the ODE solver and LP solvers are omitted because they provide exact solutions. The CPU time is in seconds, and <= 0.01 means the CPU times are less than or equal to 0.01 seconds.

Compared to the case of b as a variable, $\mathbf{D}$ being a variable is more challenging because of the higher dimensional of the input space. Figure 7 shows three specific examples solved by our neural network model and the RK45 method. Table 6 presents the results on much more instances in terms of accuracy and

computational time. Together, the figure and the table show how our neural network approach effectively and efficiently handles the situation where $\mathbf{D}$ is a variable.

6. Conclusion

In this paper, we present a deep learning approach in the form of feed-forward neural networks to solve LP problems. We show how a neural network can be used to solve a single LP problem and multiple LP problems, respectively. We demonstrate the effectiveness of the proposed method through specific LP examples. This paper is a beginning of a series of future works. We relate the most basic optimization problem, linear programming, to the machine learning (ML) community so that a large number of cutting-edge research from ML can directly benefit the linear programming study, such as parallel computing and GPU computing.

Since the neural network only gives approximate solutions, to improve the predicted accuracy, we can develop more sophisticated network architecture and training algorithms that take advantage of the problem structure. Another line of future direction could be extending the proposed approach to other optimization or operations research problems, such as quadratic or second-order conic programming.

Bibliography

Bogacki, P., & Shampine, L. F. (1989). A 3 (2) pair of runge-kutta formulas. *Applied Mathematics Letters*, *2*, 321–325.

Chen, F., Sondak, D., Protopapas, P., Mattheakis, M., Liu, S., Agarwal, D., & Giovanni, M. D. (2020). Neurodiffeq: A python package for solving differential equations with neural networks. *Journal of Open Source Software*, *5*, 1931. URL: https://doi.org/10.21105/joss.01931. doi:10.21105/joss.01931.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, *2*, 303–314. URL: https://doi.org/10.1007/BF02551274. doi:10.1007/BF02551274.

Dantzig, G. B. (1963). Linear programming and extensions, .

Deng, L., & Liu, Y. (2018). *Deep learning in natural language processing*. Springer.

Diamond, S., & Boyd, S. (2016). CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, *17*, 1–5.

Dissanayake, M. W. M. G., & Phan-Thien, N. (1994). Neural-network-based approximations for solving partial differential equations. *Communications in Numerical Methods in Engineering*, *10*, 195–201. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cnm.1640100303. doi:https://doi.org/10.1002/cnm.1640100303. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cnm.1640100303.

Djordjevic, V., Stojanovic, V., Tao, H., Song, X., He, S., & Gao, W. (2022). Data-driven control of hydraulic servo actuator based on adaptive dynamic programming. *Discrete and Continuous Dynamical Systems-S*, *15*, 1633.

Domahidi, A., Chu, E., & Boyd, S. (2013). Ecos: An socp solver for embedded systems. In *2013 European Control Conference (ECC)* (pp. 3071–3076). IEEE.

Dormand, J. R., & Prince, P. J. (1980). A family of embedded runge-kutta formulae. *Journal of computational and applied mathematics*, *6*, 19–26.

Effati, S., & Nazemi, A. (2006). Neural network models and its application for solving linear and quadratic programming problems. *Applied Mathematics and Computation*, *172*, 305–331. URL: https://www.sciencedirect.com/science/article/pii/S0096300305002055. doi:https://doi.org/10.1016/j.amc.2005.02.005.

Flamant, C., Protopapas, P., & Sondak, D. (2020). Solving differential equations using neural network solution bundles. arXiv:2006.14372.

Gass, S. I. (2003). *Linear programming: methods and applications*. Courier Corporation.

Gondzio, J. (2012). Interior point methods 25 years later. *European Journal of Operational Research*, *218*, 587–601.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Gurobi Optimization, LLC (2021). Gurobi Optimizer Reference Manual. URL: https://www.gurobi.com.

Hopfield, J. J., & Tank, D. W. (1985). "neural" computation of decisions in optimization problems. *Biological cybernetics*, *52*, 141–152.

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, *2*, 359–366. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208. doi:https://doi.org/10.1016/0893-6080(89)90020-8.

Huang, G.-B., Zhu, Q.-Y., & Siew, C.-K. (2006). Extreme learning machine: theory and applications. *Neurocomputing*, *70*, 489–501.

Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing* (pp. 302–311).

Kingma, D. P., & Ba, J. (2017). Adam: A method for stochastic optimization. arXiv:1412.6980.

Ko, C.-H., Chen, J.-S., & Yang, C.-Y. (2011). Recurrent neural networks for solving second-order cone programs. *Neurocomputing*, *74*, 3646–3653. URL: https://www.sciencedirect.com/science/article/pii/S0925231211004176. doi:https://doi.org/10.1016/j.neucom.2011.07.009.

Lagaris, I., Likas, A., & Fotiadis, D. (1998). Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, *9*, 987–1000. doi:10.1109/72.712178.

Li-Zhi, L., & Hou-Duo, Q. (1999). A neural network for the linear complementarity problem. *Mathematical and Computer Modelling*, *29*, 9–18. URL: https://www.sciencedirect.com/science/article/pii/S0895717799000266. doi:https://doi.org/10.1016/S0895-7177(99)00026-6.

Liu, W., Wang, Z., Liu, X., Zeng, N., Liu, Y., & Alsaadi, F. E. (2017). A survey of deep neural network architectures and their applications. *Neurocomputing*, *234*, 11–26.

Lu, L., Meng, X., Mao, Z., & Karniadakis, G. E. (2021). Deepxde: A deep learning library for solving differential equations. *SIAM Review*, *63*, 208–228. URL: http://dx.doi.org/10.1137/19M1274067. doi:10.1137/19m1274067.

Matlab (2017). *MATLAB version 9.3.0.713579 (R2017b)*. Natick, Massachusetts: The MathWorks Inc.

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning* ICML'10 (p. 807–814). Madison, WI, USA: Omnipress.

Nazemi, A., & Omidi, F. (2013). An efficient dynamic model for solving the shortest path problem. *Transportation Research Part C: Emerging Technologies*, *26*, 1–19. URL: https://www.sciencedirect.com/science/article/pii/S0968090X12000964. doi:https://doi.org/10.1016/j.trc.2012.07.005.

Nazemi, A., & Sabeghi, A. (2020). 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*, 181–203. URL: https://doi.org/10.1080/0952813X.2019.1647559. doi:10.1080/0952813X.2019.1647559. arXiv:https://doi.org/10.1080/0952813X.2019.1647559.

Nazemi, A., & Tahmasbi, N. (2013). A high performance neural network model for solving chance constrained optimization problems. *Neurocomputing*, *121*, 540–550.

Nocedal, J., & Wright, S. (2006). *Numerical optimization*. Springer Science & Business Media.

Rackauckas, C., & Nie, Q. (2017). Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia. *Journal of Open Research Software*, *5*.

Shampine, L. F., & Reichelt, M. W. (1997). The matlab ode suite. *SIAM journal on scientific computing*, *18*, 1–22.

Stellato, B., Banjac, G., Goulart, P., Bemporad, A., & Boyd, S. (2020). OSQP: an operator splitting solver for quadratic programs. *Mathematical Programming Computation*, *12*, 637–672. URL: https://doi.org/10.1007/s12532-020-00179-2. doi:10.1007/s12532-020-00179-2.

Teschl, G. (2012). *Ordinary differential equations and dynamical systems* volume 140. American Mathematical Soc.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., & SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*, 261–272. doi:10.1038/s41592-019-0686-2.

Voulodimos, A., Doulamis, N., Doulamis, A., & Protopapadakis, E. (2018). Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, *2018*.

Wanner, G., & Hairer, E. (1996). *Solving ordinary differential equations II* volume 375. Springer Berlin Heidelberg New York.

Wu, D., & Lisser, A. (2022). A dynamical neural network approach for solving stochastic two-player zero-sum games. *Neural Networks*, .

Xia, Y. (1996). A new neural network for solving linear and quadratic programming problems. *IEEE Transactions on Neural Networks*, *7*, 1544–1548. doi:10.1109/72.548188.

Xia, Y., & Feng, G. (2007). A new neural network for solving nonlinear projection equations. *Neural Networks*, *20*, 577–589.

Xia, Y., & Wang, J. (2000). A recurrent neural network for solving linear projection equations. *Neural Networks*, *13*, 337–350.

Xiao, W., Zhang, J., Li, Y., Zhang, S., & Yang, W. (2017). Class-specific cost regulation extreme learning machine for imbalanced classification. *Neurocomputing*, *261*, 70–82.

Zhang, J., Li, Y., Xiao, W., & Zhang, Z. (2020). Non-iterative and fast deep learning: Multilayer extreme learning machines. *Journal of the Franklin Institute*, *357*, 8925–8955.

Zhang, X., Wang, H., Stojanovic, V., Cheng, P., He, S., Luan, X., & Liu, F. (2021). Asynchronous fault detection for interval type-2 fuzzy nonhomogeneous higher-level markov jump systems with uncertain transition probabilities. *IEEE Transactions on Fuzzy Systems*, .

Zhuang, Z., Tao, H., Chen, Y., Stojanovic, V., & Paszke, W. (2022). Iterative learning control for repetitive tasks with randomly varying trial lengths using successive projection. *International Journal of Adaptive Control and Signal Processing*, *36*, 1196–1215.