Deep learning used in optimization problems

***Hau T. Mai, Sangeun Park, Taeseop Kim**

Tuan Anh Nguyen, Ngoc Tan Nguyen, †Jeahong Lee

Deep Learning Architecture Research Center, Sejong University, South Korea

*Presenting author: maitienhaunx@gmail.com

†Corresponding author: jhlee@sejong.ac.kr

Abstract: Optimization problems in engineering often find the minimum or maximum value of the objective function under a prescribed set of constraints. Gradient-based algorithms restrict to apply derivative and partial derivative for some cases, while in other case derivative-free algorithms are used but they have a slow speed of convergence. This paper presents a capable deep learning to approximate the cost and constraints function for solving convex optimization. The analysis phase and the first order partial derivative predicted from a machine learning model so that the solution optimization can be calculated. A few examples illustrate to evaluate the performance of the model.

Keywords: Deep learning, optimization problems, approximate function, convex optimization

1. Introduction

Optimization problems are an important part of the design process [3]. We easy to solve the simple problems with linear constraint and objective functions. However, in the case of nonlinear constraint or objective functions, it has a few difficulties to obtain a global solution. In recent years, two groups of optimal algorithms became the most popular that were evolutionary and gradient-based algorithms.

By using evolutionary algorithms, the gradient information of function is not needed and typically makes use of a set of design points. The advantages of these methods are extremely robust, find near global optimum, easy to implement and suite for discrete optimization problems. The big drawbacks associated with these algorithms are the issue of computational expensive, poor constraint-handling abilities, problem-specific parameter tuning, and limited problem size [1].

On the contrary case, gradient-based optimization techniques make use of gradient information to find the optimum solution. In addition to solving problems with a large

number of design variables and the low number of function evaluations, they typically require little problem-specific parameter tuning. They have difficulty solving discrete optimization problems and only locate a local optimum [1]. If the objective and constraints that are not known explicitly have a small number of function evaluations, the above global optimization techniques are not applicable to found the global optimum.

Recently, machine learning (ML) has been applied in various fields, especially the deep learning (DL), such as automatic drive, image recognition, natural language processing, computer visions, healthcare, and financial sector [2], [3], [4], [5] . In computational physics and engineering problems, deep learning has also been applied. Liang Liang et al. [6] use DL to estimate the stress distribution of the human thoracic aorta. Zhou et al. [7] use DL to select cutting tools for special-shaped machining features of complex products. Cha et al. [8] use DL to detect cracks on concrete images. Oishi et al. [9] utilize DL optimizes the numerical quadrature rule for the FEM element stiffness matrix in the element-by-element basis. Zhang et al. [10] use the deep convolutional neural network (CNN) to solve the topology optimization problem. The use of deep learning can be effective to circumvent the expensive computation in optimization problems.

In this study, we use DL to approximate objective and constraints function into other functions. The value and derivative of the new function which predicted from the trained model are applied to resolve convex optimization problems. The remaining of the paper is organized as follows. Section 2 describes some preliminaries about the deep neural network (DNN). In section 3, the capability of DL approximate function and their derivatives give details. Section 4 provides several numerical examples to show the computational efficiency. Finally, section 5 concludes the paper.

2. Deep learning

Deep learning is a subfield of machine learning in artificial intelligence (AI) that is based on learning several levels of representations, corresponding to a hierarchy of features or factors or concepts, where higher-level concepts are defined from lower-level ones, and the same lowerlevel concepts can help to define many higher-level concepts [11]. The relationship between DL and associated fields are shown in Figure 1 [12].

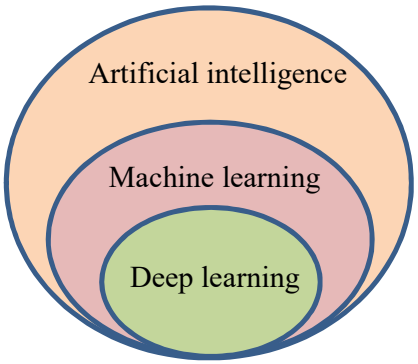

Figure 1. Venn diagram of the components of artificial intelligence

Machine learning is a set of mathematical relationships between the inputs and outputs, whether it be a linear or non-linear relationship of a given data. The model parameters estimated from the learning process such that the model can perform the specified task [12].

In figure 2, we consider a basic structure of the multi-layer feedforward neural network with the full-connected layer, which consists of the input layer, the output layer, and the second hidden layer.

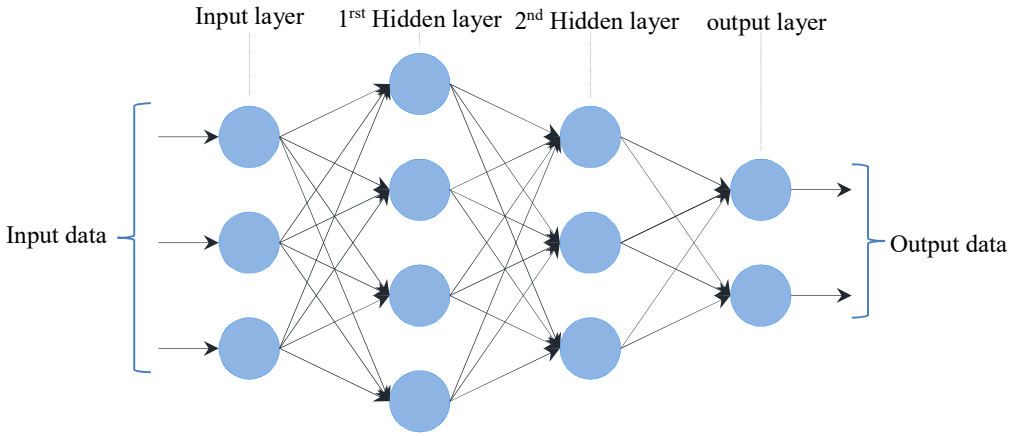

Figure 2. Typical feedforward neural network composed of four layers

In a layered neural network, the neurons are organized in the form of layers. The first layer is called the input layer, the last layer is called the output layer, and the layers between are hidden layers. Each neuron in a particular layer is connected with all neurons in the next layer. the number of neurons in the hidden layer is usually determined through the trial and error procedure. The non-linear function can approximate with any precision desired when the feedforward neural network has more than three layers [9].

The block diagram of an artificial neuron is depicted in figure 3 [12]. The connection between the *i*th and *j*th neuron is characterized by the weights w_{ji} and the jth neuron by the bias b_j^p, which reflects the degree of importance of the given connection in the neural network (NN). The output value of the *j*th neuron is determined by Eqs. (1) [9]:

$$x_j^p = f(z_j^p)$$
$$z_j^p = \sum_{i=1}^{n_{p-1}} w_{ji}^{p-1}.x_i^{p-1} + b_j^p \quad (1)$$

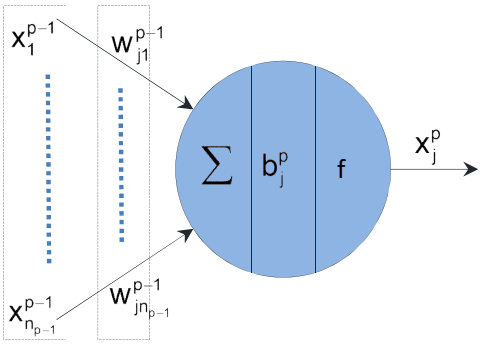

Figure 3. Block diagram of artificial neuron

where x_i^{p-1} is the output of the ith neuron in the (p-1)th layer, w_{ji}^{p-1} is weight to connect between the *i*th neuron in the *(p-1)*th layer and the *j*th neuron in the *p*th layer, b_j^p is the bias of the *j*th neuron in the *p*th layer, z_j^p is the input value of the activation function at the *j*th neuron of the *p*th layer, x_j^p is the output value of the activation function at the *j*th neuron of the *p*th layer, f is the activation function, n_{p-1} is the number of neurons in the *(p-1)*th layer.

In supervised learning, training a NN is the process of adjusting values for the weights and biases of the network to perform the desired function correctly. The error back-propagation is usually adapted to minimize the sum of the squared differences between the computed and correct output values E [5]:

$$E = \frac{1}{2}\sum_{j=1}^{n_N}\left(y_j - \hat{y}_j\right)^2 \quad (2)$$

where y_j is the computed output value of the *j*th neuron in the output layer, $\hat{y}_j$ is the required output value of the *j*th neuron in the output layer, n_N is the number neuron of the output layer. Eqs. (2) is known as the loss function.

In the back-propagation algorithm, the steepest-descent minimization method is used. The new weights and biases are adjusted for the next iteration of the network trains [12]:

$$w_{ji(new)}^{p-1} = w_{ji(old)}^{p-1} + \eta \sum_{k=1}^{n_k} \left(\frac{\partial E}{\partial w_{ji(old)}^{p-1}} \right)_k + \alpha \, \Delta w_{ji(previous)}^{p-1} \tag{3}$$

$$b_{j(new)}^{p} = b_{j(old)}^{p} + \eta \sum_{k=1}^{n_k} \left(\frac{\partial E}{\partial b_{j(old)}^{p}} \right)_k + \alpha \, \Delta b_{j(previous)}^{p} \tag{4}$$

where:
$$\frac{\partial E}{\partial w_{ji}^{p-1}} = \frac{\partial E}{\partial x_j^p} \frac{\partial x_j^p}{\partial w_{ji}^{p-1}} = \frac{\partial E}{\partial x_j^p} f'(z_j^p) x_i^{p-1} \tag{5}$$

$$\frac{\partial E}{\partial b_j^p} = \frac{\partial E}{\partial x_j^p} \frac{\partial x_j^p}{\partial b_j^p} = \frac{\partial E}{\partial x_j^p} f'(z_j^p) \tag{6}$$

if $j \in$ output layer

$$\frac{\partial E}{\partial x_j^p} = y_j - \hat{y}_j \tag{7}$$

$j \in$ hidden layer

$$\frac{\partial E}{\partial x_j^p} = \sum_{i \in \Gamma_p} \frac{\partial E}{\partial b_j^p} w_{ji}^{p-1} \tag{8}$$

Where η is the learning coefficient $\left(\eta \in (0,1)\right)$, n_k is the number of training pairs in training set, α is the momentum factor $\left(\alpha \in (0,1)\right)$. They are multiplied to the weights and biases adjustment which are proportional to the amount of the previous weights and biases change. So, selecting the appropriate values for the learning coefficient and the momentum factor will help to improve the training speed, accuracy takes into the ability of the training process [13]. Several research has been proposed to adaptive this parameter [13].

When the error defined in Eqs.(2) gradually decreases epoch by epoch [9], and the model learns fit the training data too well but has poor fit with new datasets, which is called overfitting. It is also shown that the machine learning model will lose generalization capability to any data from the problem domain. Otherwise, if the error back-propagation is

large and the algorithm does not fit the data well enough, it may occur underfitting. The problems of overfitting and underfitting are depicted in Figure 4 when the model learning approximate a nonlinear function. To limit the overfitting or underfitting, we have a few of the most popular solutions as follows: validation, cross-validation, train with more data, remove features, early stopping, regularization, dropout and so on.

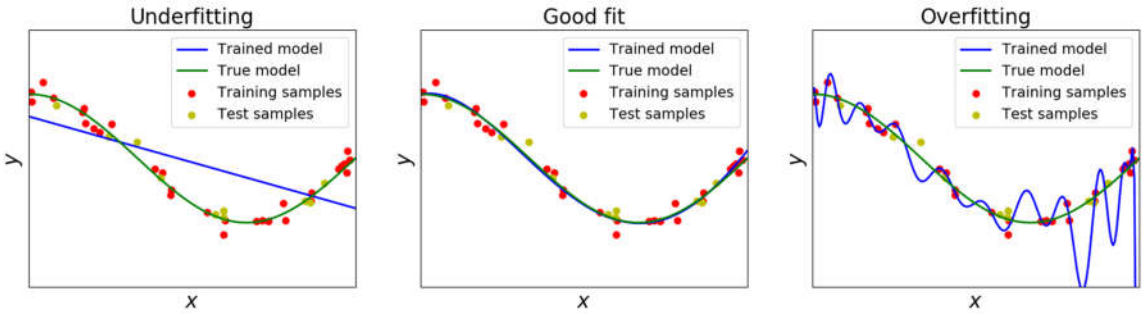

Figure 4. Underfitting and overfitting

A DNN is an artificial NN with multiple hidden layers (more than three hidden layers [14]) between the input and output layers [11], which can gain better complex non-linear relationships. However, increasing network depth is hard to train because of the vanishing gradient problem. Repeat multiplication may make the gradient extremely small, which is back-propagation to previous layers. Since the weights and biases in Eq. (3, 4) are not be adjusted, the model gets saturated or even starts degrading rapidly. To circumvent this difficulty, the concept of skip connection first introduced by He et al [15]. As show in Figure 5, they allow a short path for gradient to flow through and the higher layer will perform at least as good as the lower layer.

3. Approximate of function and their derivatives by DNN

We first need to prepare datasets for training and testing with various conditions to gaurantee the generality model and avoid overfitting. A input – output relationship is considered: the vector $\mathbf{x} = \left\{x_1^c,\ x_2^c, ..., x_n^c\right\}$ as the input and the vector $\mathbf{y} = \left\{y_1^c,\ y_2^c, ..., y_m^c\right\}$ as the output, where c is the *c*th data pairs of a large set of data pairs [9]. They must be normalized so that the value belong to the set of all numbers between 0 and 1. Normalization schemes for the vectors critically affect the performance and training of the NN [16], [12], [17], [13].

Secondly, The DL model is built with the hidden layers, neurons, activation function (AF), optimization algorithm and other parameters. We consider the DNN showed in Figure 6, which is used to training for approximating function.

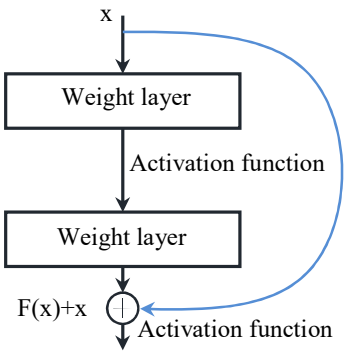

Figure 5. Skip connection

where **x** is the input vector, y is the output vector, $\mathbf{w}^p$ is the connection weigth between the *p*th layer and *(p-1)*th layer, $\mathbf{b}^p$ is the bias of the *p*th layer, p is the number hidden layer (p≥3).

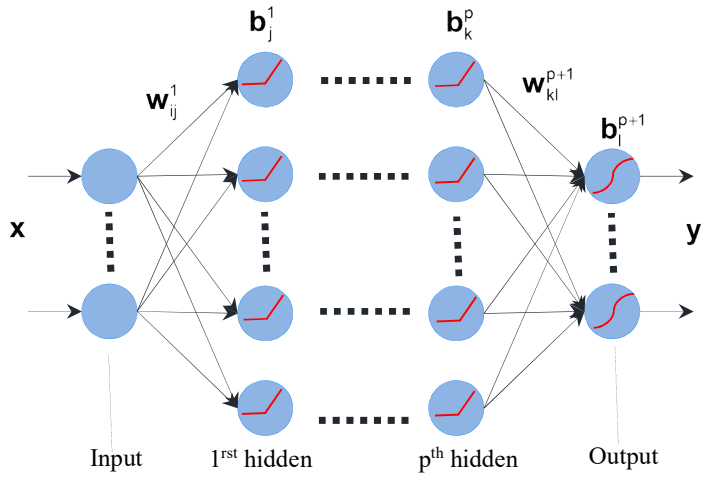

Figure 6. Deep neural network

AF is employed in neurons of the hidden and output layer to compute the weighted sum of input and biases [12]. We need to apply AF to the NN, which can be learn something complex and represent a nonlinear arbitrary functional mapping between inputs and outputs. Most popular types of AF shown in Table 1. Rectified linear units (ReLU) was proposed introduction by Nair and Hinton 2010, which has strong biological and mathematical underpinnings. It offers better performance and generalization in deep learning compared to the other AF [12], [14]. The main advantages of using ReLU are faster learning, improve the convergence and eliminate the vanishing gradient problem. Almost all DL models, ReLU often uses in the hidden layers and combine the dropout technique to reduce the overfitting [10], [6], [8], [3], [4], [14]. The softmax function is another types of AF used to compute

probability distribution from a vector of real numbers. It is used in the output layer of most common practice DL applications [8], [3], [4], [14].

Table 1. Activation functions

No.	Function	Equation	Derivative
1	ReLU	$f(\mathbf{x}) = \begin{cases} x_i & \text{for } x_i > 0 \\ 0 & \text{for } x_i \le 0 \end{cases}$	$f'(\mathbf{x}) = \begin{cases} 1 & \text{for } x_i > 0 \\ 0 & \text{for } x_i \le 0 \end{cases}$
2	LeakyReLU	$f(\mathbf{x}) = \begin{cases} x_i & \text{for } x_i > 0 \\ \alpha x_i & \text{for } x_i \le 0 \end{cases}$	$f'(\mathbf{x}) = \begin{cases} 1 & \text{for } x_i > 0 \\ \alpha & \text{for } x_i \le 0 \end{cases}$
3	Softmax	$f(x_i) = \frac{e^{x_i}}{\sum_{i=1}^{k} e^{x_k}}$ for $i = 1,\ldots,k$	$f'(\mathbf{x}) = f(\mathbf{x})(1 - f(\mathbf{x}))$
4	Softsign	$f(\mathbf{x}) = \frac{x}{1+\lvert x\rvert}$	$f'(\mathbf{x}) = \begin{cases} \frac{1}{(1+x)^2} & \text{for } x \ge 0 \\ \frac{1}{(1-x)^2} & \text{for } x < 0 \end{cases}$
5	Softplus	$f(\mathbf{x}) = \ln(1 + e^x)$	$f'(\mathbf{x}) = \frac{1}{1+e^{-x}}$
6	Sigmoid	$f(\mathbf{x}) = \frac{1}{1+e^{-x}}$	$f'(\mathbf{x}) = f(\mathbf{x})(1 - f(\mathbf{x}))$
7	Tanh	$f(\mathbf{x}) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$f'(\mathbf{x}) = (1 - f(x)^2)$

Shiyu et al. [18] indicated the relationship between the number of hidden layers, neurons and approximation error. Shallow networks require exponentially more neurons than a deep network to achieve the level of accuracy for function approximation. However, we have been at least as difficult to determine exactly the number of hidden layers, neurons, and parameters for the problem under consideration.

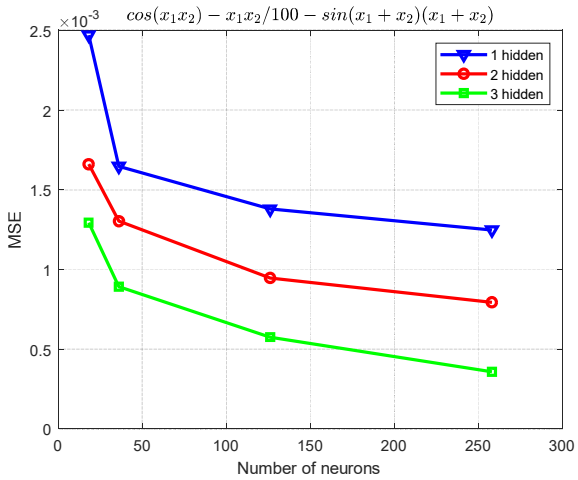

Figure 7. Various number of hidden layer

We approximate a trigonometric polynomial $f(\mathbf{x}) = \cos(x_1 x_2) - \frac{x_1 x_2}{100} - \sin(x_1 + x_2)(x_1 + x_2)$ with two input variables using standard DL with 1, 2 or 3 hidden layers, the total number of neurons in the hidden layers 18, 36, 126 and 258. In Figure 7, it is shown that increasing suitable the number of hidden layers will decrease mean squared error (MSE) of the testing.

As the next step, we will use our data to perform the training process where the values in weigths and biases are adjusted to improve the predictability of the model. After once training is complete, we need evaluation of our model against data that has never been used training. It allows us to see how the model might predict the dataset that it has not yet seen.

Constrained optimization problem may be written as follows:

$$\min \quad y_0(\mathbf{x}) \tag{9}$$

$$\text{subject to} \quad \begin{aligned} y_i(\mathbf{x}) &= 0 \qquad i = 1,.....,n \\ y_j(\mathbf{x}) &\le 0 \qquad j = (n+1),.....,m \end{aligned} \tag{10}$$

where y_0 is the objective function, y_1, y_2, …, y_m are the constraint functions, $\mathbf{x} \in \mathbf{R}^d$ is the vector of d design variables and m is the equality and inequality constraints.

A DL model that contains the input variables and the output in the objective and constraint functions, is built to approximate the problem. The value of functions are predicted by the trained model according to [1]

$$\bar{\mathbf{y}}_o(\mathbf{x}) = f\left(\mathbf{Z}_j^{(p+1)}\right) \qquad o = 0,1,...,m \tag{11}$$

$$\begin{aligned} \mathbf{Z}_j^{(k)} &= \sum_{i=1}^{m} w_{ij}^k f(\mathbf{Z}_j^{(k-1)}) + \mathbf{b}_j \qquad k = 1,...,(p+1) \\ f\left(\mathbf{Z}^{(0)}\right) &= \mathbf{x} \end{aligned} \tag{12}$$

and the corresponding first order derivatives are given by Eq. (13)

$$\frac{\partial \bar{\mathbf{y}}_o(\mathbf{x})}{\partial \mathbf{x}} = \left(\prod_{i=1}^{p+1} f'\left(\mathbf{Z}^{(i)}\right) \mathbf{w}^i \right) \mathbf{I} \tag{13}$$

where $\bar{\mathbf{y}}_o$ is the value of function approximation, f is the activation function, p is the number of the hidden layers, $\mathbf{Z}_j^{(k)}$ is the output vector of all neurons in the *k*th layer, j is the number of neurons in the layer, w_{ij}^k is the weights between the *k*th layer and *(k-1)*th layer, $\mathbf{b}_j$ is the bias vector, m is the number of hidden neurons in the *k*th layer, and $\mathbf{I}_{(dxd)}$ is the identity matrix.

Therefore, the optimization problem would be solved on the approximate model according to

$$\min \quad \bar{y}_0(\mathbf{x}) \tag{14}$$

$$\text{subject to} \quad \begin{aligned} &\bar{y}_i(\mathbf{x}) = 0 \quad i = 1,.....,n \\ &\bar{y}_j(\mathbf{x}) \le 0 \quad j = (n+1),.....,m \end{aligned} \tag{15}$$

The Lagrangian for the problem using Eqs. (14) to (15) is given as

$$L(x,\lambda) = \bar{y}_0(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \bar{y}_i \tag{16}$$

Therefore, the following optimization problem could be solved with Kuhn-Tucker as follows:

$$\begin{aligned} &\frac{\partial L}{\partial x_j} = \frac{\partial \bar{y}_0}{\partial x_j} + \sum_{i=1}^{m} \lambda_i \frac{\partial \bar{y}_i}{\partial x_j} \qquad j = 1 \text{ to } d \\ &\bar{y}_i = 0 \qquad i = 1 \text{ to } n \\ &\lambda_i \bar{y}_i = 0 \qquad i = (n+1) \text{ to } m \\ &\lambda_i \ge 0 \qquad i = (n+1) \text{ to } m \end{aligned} \tag{17}$$

4. Numerical results

In order to demonstrate the performance of deep learning, benchmark and real-world engineering constrained optimization problems will be presented and compared the predicted and optimal values. The dataset is generated according to the domain of the variables [17]. The network includes an input layer, an output layer, and three hidden layers. ReLU was used in each hidden layer, and the output layer of the generator uses the Softmax function. The skip connection is applied in the last hidden layer. The ADAM (adaptive moment estimation) optimization algorithm is used to optimize the weights and biases to minimize the loss function which measured by the MSE. The learning rate was defined as 0.001, momentum as 0.9, the number of epochs as 100, and batch size as 32. The Quasi_Newton method is used to optimize with the value and derivative of functions that are predicted by DL model.

4.1. Constrained benchmark problems.

In this section, DL is performed on the 5 well-known benchmark constrained functions taken from [17], [19] (see Appendix A). The parameters of neural architecture presented in Table 2. The neural network will stop training when the error remains constant. The result obtained is excellent with the maximum error of 0.43% and the minimum error of 0.0004% of the objective function value in Table 3. We can see which optimal results on the approximation

function are quite close to the original function. Therefore, the DL model is a good approximation for the objective and constraints function. As shown in Figure 8, the shape of the objective function is the same approximation function.

Table 2. Network architecture

Fun.	Data pairs		Neurons in the layers	Time training(s)	Error MSE
	Training	Testing			
A.1	2000	200	2-32-32-3	21	2.04E-6
A.2	2500	250	2-48-48-3	24	2.72E-6
A.3	3200	320	3-64-64-3	35	4.07E-6
A.4	3500	350	3-64-64-4	41	4.82E-6
A.5	2500	200	2-32-32-3	24	7.82E-6

Table 3. The optimal result for constrained benchmarks

Fun.	Optimal	Quasi_Newton	Present	Error (%)
A.1	-7.2	-7.199	-7.231	0.43
A.2	-225	-224.999	-224.773	0.0004
A.3	-40/9	-4.444	-4.449	0.001
A.4	1	0.999	1.0002	0.01
A.5	-0.998673	-0.998	-0.995	0.36

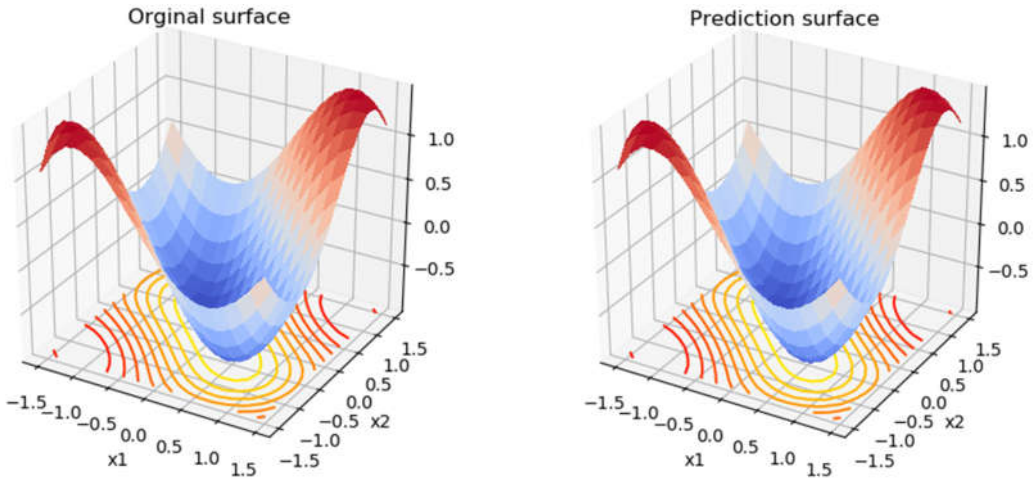

Figure 8. Objective function approximation

4.2.Engineering design problems.

In this example, we must minimize the weight and subject to the stress of the following three-bar truss problem (see Appendix B). The model uses 8000 data pairs to training and 800 data pairs to test, 128 neurons in the hidden layers. The training accuracy obtained 99%, and MSE 2.58E-7. The comparison of the solution obtained from DL, Quasi_Newton, backtracking search (BSA), hybrid particle swarm optimization and differential evolution, a hybrid evolutionary algorithm is shown in Table 4. Figure 9 shows the weight convergence histories obtained by the approximation function from DL, Quasi_Newton for this structure. Again, the DL model shows its effectiveness to solve the optimization problem based on the approximation function.

Table 4. Comparison of solution for three-bar truss design problem

Method	HEAA	PSO-DE	BSA	Quasi_Newton	Present
x_1	0.78868	0.788675	0.788675	0.788693	0.785695
x_2	0.408234	0.408248	0.408248	0.408196	0.404468
$g_1(x)$	NA	-5.29E-11	-3.23E-12	2.56E-6	8.84E-5
$g_2(x)$	NA	-1.463748	-1.464102	-1.463891	-1.469825
$g_3(x)$	NA	-0.536252	-0.535898	-0.536105	-0.528274
f(x)	263.895843	263.895843	263.895843	263.895838	263.582674

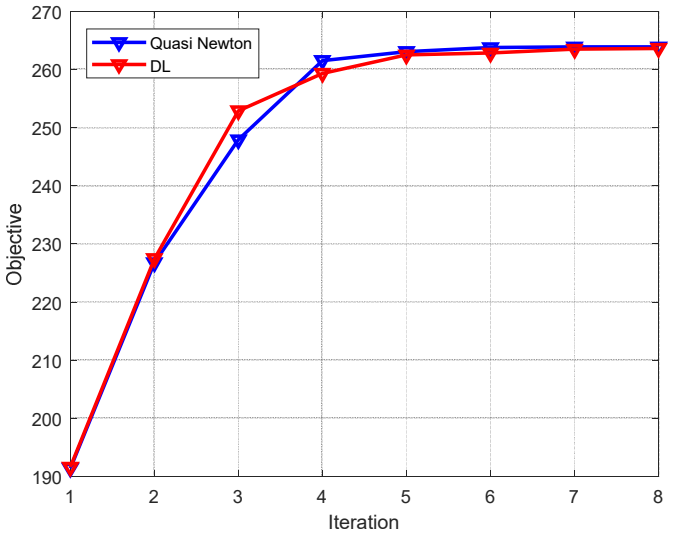

Figure 9. The weight convergence histories of three-bar truss obtained using the DL, Quasi_Newton

5. Conclusions

In this paper, we have successfully implemented optimization with the value and derivative of the function by the DL model. Six numerical examples with convex properties to verify the effectiveness of the proposed approach. It is shown that the method yield converges globally to a unique optimal solution of problems.

Appendix

A. Constrained benchmark problems

A.1. Constrained problem 01 [19]

$$\begin{aligned} \min \quad & f(x) = x_1^2 + 2x_2^2 - 2x_1x_2 - 2x_1 - 6x_2 \\ \text{subject to}: \quad & g_1(x) = x_1 + x_2 - 2 \le 0 \\ & g_2(x) = -x_1 + 2x_2 - 2 \le 0 \\ & x_i \ge 0 \quad i = 1, 2 \end{aligned}$$

A.2. Constrained problem 02 [19]

$$\begin{aligned} \min \quad & f(x) = x_1^2 + 2x_2^2 + x_1x_2 - 30x_1 - 30x_2 \\ \text{subject to}: \quad & g_1(x) = \frac{5}{12}x_1 - x_2 - \frac{35}{12} \le 0 \\ & g_2(x) = \frac{5}{2}x_1 + x_2 - \frac{35}{2} \le 0 \\ & x_1 \ge -5, \;\; x_2 \le 5 \end{aligned}$$

A.3. Constrained problem 03 [19]

$$\begin{aligned} \min \quad & f(x) = x_1^2 + 2x_2^2 + 0.5x_3^2 + x_1x_2 + x_1x_3 - 4x_1 - 3x_2 - 2x_3 \\ \text{subject to}: \quad & g_1(x) = x_1 + x_2 + 2x_3 - 3 \le 0 \\ & g_2(x) = 3x_1 - 9x_2 + 9x_3 = 1 \\ & 0 \le x_i \le \frac{4}{3}, \quad i = 1,\ldots,3 \end{aligned}$$

A.4. Constrained problem 04 [19]

$$\begin{aligned} \min \quad & f(x) = 10x_1^2 + 2x_2^2 + 2x_3^2 - 2*(x_1x_2 + 3*x_1x_3 - x_2x_3) \\ & -1 \le -x_1 + x_2 \le 0 \\ \text{subject to}: \quad & -1 \le x_3 - 3x_1 \le 1 \\ & 1 \le x_2 + x_3 \le 2 \\ & 0 \le x_i \le \frac{4}{3}, \quad i = 1,\ldots,3 \end{aligned}$$

A.5. Constrained problem 05 [17]

$$\begin{aligned} \min \quad & f(x) = -\cos(x_1 x_2) + \frac{x_1 x_2}{100} + \sin\left(x_1 + x_2\right)\left(x_1 + x_2\right) \\ \text{subject to}: \quad & g_1(x) = -x_1 + x_2 + 0.5 \le 0 \\ & g_2(x) = x_1 x_2 - 15 \le 0 \\ & 0 \le x_1 \le 1.5, \qquad -1 \le x_2 \le 1 \end{aligned}$$

B Three-Bar Truss Design Problem [20]

$$\begin{aligned} \min \quad & f(x) = (2\sqrt{2}x_1 + x_2)\ell \\ \text{subject to}: \quad & g_1(x) = \frac{P\left(\sqrt{2}x_1 + x_2\right)}{\sqrt{2}x_1^2 + 2x_1 x_2} - \sigma \le 0 \\ & g_2(x) = \frac{Px_2}{\sqrt{2}x_1^2 + 2x_1 x_2} - \sigma \le 0 \\ & g_3(x) = \frac{P}{\sqrt{2}x_2 + x_1} - \sigma \le 0 \\ & 0 \le x_i \le 1 \quad i = 1, 2 \\ & \ell = 100\text{cm}, \quad P = 2\text{kN}, \quad \sigma = 2\text{kN/cm}^2 \end{aligned}$$

where x_1, x_2 are the cross-sectional areas of the bars, σ is the maximum permissible stress in tension, P is force

Acknowledgements

This research was supported by grants (NRF-2018R1A2A1A05018287, NRF-2017R1A4A1015660) from NRF (National Research Foundation of Korea) funded by MEST (Ministry of Education and Science Technology) of Korean government.

References

[1] G. Venter(2010), "Review of Optimization Techniques," in *Encyclopedia of Aerospace Engineering*.
[2] W. E, J. Han, and A. Jentzen, (2017) "Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations," *ArXiv170604702,*
[3] J. Schmidhuber (2015), "Deep learning in neural networks: An overview," *Neural Netw.*, 85–117.
[4] Y. Guo, Y. Liu, A. Oerlemans, S. Lao, S. Wu, and M. S. Lew (2016), "Deep learning for visual understanding: A review," *Neurocomputing*, 27–48.
[5] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, Apr. 2017.
[6] Liang Liang, Liu Minliang, Martin Caitlin, and Sun Wei (2018), "A deep learning approach to estimate stress distribution: a fast and accurate surrogate of finite-element analysis," *J. R. Soc. Interface*, vol. 15.
[7] G. Zhou, X. Yang, C. Zhang, Z. Li, and Z. Xiao (2019), "Deep learning enabled cutting tool selection for special-shaped machining features of complex products," *Adv. Eng. Softw.*, 1–11.
[8] Y.-J. Cha, W. Choi, and O. Büyüköztürk (2017), "Deep Learning-Based Crack Damage Detection Using Convolutional Neural Networks," *Comput.-Aided Civ. Infrastruct. Eng.*, 361–378.
[9] A. Oishi and G. Yagawa (2017), "Computational mechanics enhanced by deep learning," *Comput. Methods Appl. Mech. Eng.*, 327–351.
[10] Y. Zhang, A. Chen, B. Peng, X. Zhou, and D. Wang, "A deep Convolutional Neural Network for topology optimization with strong generalization ability," 13.
[11] L. Deng and D. Yu, "Deep Learning: Methods and Applications," *Deep Learn.*, p. 197.
[12] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall (2018), "Activation Functions: Comparison of trends in Practice and Research for Deep Learning," *ArXiv181103378 Cs*.
[13] C. J. Hyeong-Taek Kang, "Neural Network Approaches to Aid Simple Truss Design Problems."

[14] A. Abobakr, M. Hossny, and S. Nahavandi (2018), “SSIMLayer: Towards Robust Deep Representation Learning via Nonlinear Structural Similarity,” *ArXiv180609152 Cs*
[15] K. He, X. Zhang, S. Ren, and J. Sun (2015), “Deep Residual Learning for Image Recognition,” *ArXiv151203385.*
[16] T. Nguyen-Thien and T. Tran-Cong (1999), “Approximation of functions and their derivatives: A neural network implementation with applications,” *Appl. Math. Model.*, 687–704.
[17] G. Villarrubia, J. F. De Paz, P. Chamoso, and F. D. la Prieta (2018), “Artificial neural networks used in optimization problems,” *Neurocomputing*, 10–16.
[18] S. Liang and R. Srikant (2017), “Why deep neural networks for function ap- proximation?,” 17.
[19] “A New Neural Network for Solving Linear and Quadratic Programming Problems.”
[20] H. Wang, Z. Hu, Y. Sun, Q. Su, and X. Xia (2018), “Modified Backtracking Search Optimization Algorithm Inspired by Simulated Annealing for Constrained Engineering Optimization Problems,” *Computational Intelligence and Neuroscience*.