Investigate Deep Learning Methods for Scientific Computing Problems:

Neural Networks for Nonconvex Energy Minimization

Final Report

MATH 4999 Mathematics Project

(2020 – 2021)

Name:	Lee Chui Shan
University Number:	3035473000
Supervisor:	Dr. Zhiwen Zhang
Date of Submission:	26 April 2021

DEPARTMENT OF MATHEMATICS

THE UNIVERSITY OF HONG KONG

Abstract

In molecular modelling, the stable geometry of the molecule is associated with the minimum potential energy configurations of the atoms. This project studies the energy minimization problem of elastic crystals, which asks how the elastic crystals should be deformed to achieve the minimum energy under some Dirichlet boundary constraint. A neural network approach inspired by the physics-informed neural networks (PINNs) is introduced to solve a specific problem in the 1-dimensional case, and is shown to be capable of deducing the optimal deformation. This approach may be applied similarly to this type of problem in higher dimensions.

Acknowledgements

I would like to express my deepest gratitude to Dr. Zhang, my project supervisor, who gave me the precious opportunity to learn how to customize neural network models and apply it to solve a meaningful problem. His professional guidance and continuous support are greatly appreciated.

Contents

List of Figures

1 Introduction

Elastic bendable molecular crystals, which recently evoked tremendous interest from the material engineering community, are potentially applicable in electronics, sensors, actuators, and more due to their exceptional mechanical flexibility [1]. In designing new molecules, the minimization of potential energy is considered as it allows the identification of physically stable molecular geometry [2, 3]. It is widely accepted that the general formula of the potential energy of an elastic crystal is

$$E(\mathbf{u}) = \int_{\Omega} W(\nabla \mathbf{u}(\mathbf{x}))\, d\mathbf{x}, \tag{1}$$

where $\Omega \subset \mathbb{R}^3$ is the reference geometry of the solid, and $W : \mathbb{R}^{3\times 3} \to \mathbb{R}$ defines the potential energy density which depends on the deformation $\mathbf{u}$ experienced by the solid [4]. The corresponding energy minimization problem is to choose a vector-valued function $\mathbf{u} : \Omega \to \mathbb{R}^3$ such that the potential energy given by equation 1 is minimized. The problem can even be generalized to higher dimensions. To set up a foundation for solving the more general problems, this project considers a problem of 1 dimension under some Dirichlet boundary condition on $\partial\Omega$.

Inspired by the physics-informed neural networks (PINNs) that emerged recently for solving partial differential equations (PDEs) [5], deep neural networks are employed to approximate the energy minimum. PINNs concurrently uses the physics described in the PDEs and data collected from the real world as inputs to train a deep neural network that outputs the solution. The loss function involved is designed to enforce the initial and boundary conditions and to satisfy the PDE at the collocation points. By modifying the definition of the loss functions and also adopting the idea of placing self-adaptive weights in the loss function [6], this project will propose some methods which together solve the 1-dimensional problem studied.

As follows, Section 2 will introduce the general definition of the energy minimization problem in the context of elastic crystals, and then a specific 1-dimensional problem of the form along with the analysis of its theoretical solutions. Next, Section 3 will give the mathematical details of the four methods that are designed to solve the 1-dimensional problem. Then, based on the methods, neural networks are built using TensorFlow. The details and the performance of the models are shown in Section 4. Finally, Section 5 concludes the report.

2 Problem Setup

2.1 The General Problem

For a general n, let $\Omega \subset \mathbb{R}^n$. Suppose $W : \mathbb{R}^{n\times n} \to \mathbb{R}$ defines the energy density, and $\mathbf{u} : \Omega \to \mathbb{R}^n$ defines the deformation which is assumed to be continuous. An energy minimization problem can be defined as

$$\begin{aligned} \min \quad & E(\mathbf{u}) = \int_\Omega W(\nabla \mathbf{u}(\mathbf{x}))\, d\mathbf{x} \\ \text{s.t.} \quad & \mathbf{u} = \mathbf{g} \quad \text{on } \partial\Omega, \end{aligned} \tag{2}$$

where g : $\partial\Omega \to \mathbb{R}^n$ is a smooth function. Since $\partial\Omega = \overline{\Omega} \cap \overline{\mathbb{R}^n \setminus \Omega}$ is a closed subset in $\mathbb{R}^n$, the definition of g can be extended to the entire Ω so that it remains smooth by Whitney extension theorem [7]. Denote $\overline{\mathbf{u}} := \mathbf{u} - \mathbf{g}$. Then problem 2 is equivalent to

$$\begin{aligned} \min \quad & E(\overline{\mathbf{u}}) = \int_\Omega W((\nabla \overline{\mathbf{u}} + \nabla \mathbf{g})(\mathbf{x}))\, d\mathbf{x} \\ \text{s.t.} \quad & \overline{\mathbf{u}} = 0 \quad \text{on } \partial\Omega. \end{aligned} \tag{3}$$

2.2 The 1-dimensional Problem

Consider the 1-dimensional problem (i.e. $n = 1$) where $\Omega := [0, 1] \subset \mathbb{R}$. Define $W : \mathbb{R} \to \mathbb{R}$ by $W(z) = z^2(1 - z)^2$ for any $z \in \mathbb{R}$ so that W is a double well potential with minima at $z = 0$ and $z = 1$. Suppose $u : [0, 1] \to \mathbb{R}$ is a continuous function that satisfies the boundary conditions $u(0) = 0$ and $u(1) = \gamma$ for some constant $\gamma \in \mathbb{R}$. Then the energy minimization problem is

$$\begin{aligned} \min \quad & E(u) = \int_0^1 W(u'(x))\, dx \\ \text{s.t.} \quad & u(0) = 0 \text{ and } u(1) = \gamma. \end{aligned} \tag{4}$$

Consider $g : \Omega \to \mathbb{R}$ defined as $g(x) = \gamma x$ for $x \in \Omega$. By substituting $\overline{u} = u - g$, problem 4 can be restated equivalently as

$$\begin{aligned} \min \quad & E(\overline{u}) = \int_0^1 W_\gamma(\overline{u}'(x))\, dx \\ \text{s.t.} \quad & \overline{u}(0) = \overline{u}(1) = 0 \end{aligned} \tag{5}$$

where $W_\gamma(z) := W(z + \gamma)$ for any $z \in \mathbb{R}$.

The solution of the problem can be analyzed by considering the following two cases, namely **(1)** when $\gamma \notin (0, 1)$ and **(2)** when $\gamma \in (0, 1)$.

Case (1): $\gamma \notin (0,1)$. In this case, consider problem 5. We have

$$\begin{aligned}
E(\overline{u}) &= \int_0^1 W_\gamma(\overline{u}'(x))\,dx \\
&= \int_0^1 (\overline{u}'(x)+\gamma)^2(1-\overline{u}'(x)-\gamma)^2\,dx \\
&= \int_0^1 [\overline{u}'(x)]^4 + (4\gamma-2)[\overline{u}'(x)]^3 + (6\gamma^2-6\gamma+1)[\overline{u}'(x)]^2 + (4\gamma^3-6\gamma^2+2\gamma)\overline{u}'(x) + \gamma^2(1-\gamma)^2\,dx \\
&\geq \int_0^1 (4\gamma-2)[\overline{u}'(x)]^3 + (4\gamma^3-6\gamma^2+2\gamma)\overline{u}'(x) + \gamma^2(1-\gamma)^2\,dx \\
&= \gamma^2(1-\gamma)^2 + \int_0^1 (4\gamma-2)[\overline{u}'(x)]^2 + (4\gamma^3-6\gamma^2+2\gamma)\,d(\overline{u}(x))
\end{aligned}$$

If $\gamma \geq 1$, then

$$\begin{aligned}
E(\overline{u}) &\geq \gamma^2(1-\gamma)^2 + \int_0^1 4\gamma^3-6\gamma^2+2\gamma\,d(\overline{u}(x)) \\
&= \gamma^2(1-\gamma)^2 \ \text{ since } \overline{u}(0)=\overline{u}(1)=0
\end{aligned}$$

If $\gamma \leq 0$, then by substituting $\overline{v}(x) := \overline{u}(1-x)$, we have $\overline{v}(0)=\overline{v}(1)=0$ and thus

$$\begin{aligned}
E(\overline{u}) &\geq \gamma^2(1-\gamma)^2 - \int_0^1 (4\gamma-2)[\overline{v}'(x)]^2 + 4\gamma^3-6\gamma^2+2\gamma\,d(\overline{v}(x)) \\
&\geq \gamma^2(1-\gamma)^2 - \int_0^1 4\gamma^3-6\gamma^2+2\gamma\,d(\overline{v}(x)) \\
&= \gamma^2(1-\gamma)^2 \ \text{ since } \overline{v}(0)=\overline{v}(1)=0
\end{aligned}$$

In both cases, the lower bound $E(\overline{u}) = \gamma^2(1-\gamma^2)$ can be achieved by the trivial minimizer

$$\overline{u} \equiv 0,$$

since $\overline{u} \equiv 0$ implies $\overline{u}' \equiv 0$, which gives $E(\overline{u}) = \int_0^1 W_\gamma(0)\,dx = \int_0^1 W(\gamma)\,dx = \int_0^1 \gamma^2(1-\gamma)^2\,dx = \gamma^2(1-\gamma)^2$. Thus, when $\gamma \notin (0,1)$, a minimizer is $\overline{u}(x) \equiv 0$ for problem 5, or $u(x) = \gamma x$ for problem 4.

Case (2): $\gamma \in (0,1)$. In this case, a minimizer of problem 4 is the piecewise linear function

$$u(x) = \begin{cases} 0 & \text{if } x \in [0, 1-\gamma] \\ x-1+\gamma & \text{if } x \in (1-\gamma, 1] \end{cases}$$

because it gives $u'(x) = \begin{cases} 0 & \text{if } x \in [0,1-\gamma] \\ 1 & \text{if } x \in (1-\gamma,1] \end{cases}$ and hence $E(u) = \int_0^{1-\gamma} W(0)\,dx + \int_{1-\gamma}^1 W(1)\,dx = 0$.

In particular, when $\gamma = 0.5$, it is easy to observe that both $u(x) = \begin{cases} 0 & \text{if } x \in [0,0.5] \\ x-0.5 & \text{if } x \in (0.5,1] \end{cases}$ and $u(x) = \begin{cases} x & \text{if } x \in [0,0.5] \\ 0.5 & \text{if } x \in (0.5,1] \end{cases}$ are minimizers of the problem.

3 Methods

To solve the 1-dimensional energy minimization problem, four different forms of neural networks are devised for different values of γ under the PINNs framework [5].

3.1 Method 1: For $\gamma \geq 1$

Let $\overline{u}(x)$ be approximated by the output $\overline{u}_\theta(x)$ of a neural network with input x. Define the approximation network $\overline{u}(x; \mathbf{w})$, where $\mathbf{w}$ represents the network weights. The weights are trained by minimizing the loss function

$$\mathcal{L}(\mathbf{w}, \lambda_0, \lambda_1) = \mathcal{L}_\text{e}(\mathbf{w}) + \mathcal{L}_\text{b}(\mathbf{w}, \lambda_0, \lambda_1), \tag{6}$$

where λ_0 and λ_1 are trainable self-adaptation weights for the boundary points $x = 0$ and $x = 1$ respectively. The first partial loss in 6

$$\mathcal{L}_\text{e}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} \left|W_\gamma(\overline{u}'(x_i; \mathbf{w}))\right|, \tag{7}$$

where $\{x_i\}_{i=1}^N$ are N evenly-spaced sample points on $[0, 1]$, enforces the minimization of the integral, while the second partial loss in 6

$$\mathcal{L}_\text{b}(\mathbf{w}, \lambda_0, \lambda_1) = \frac{1}{2}(\lambda_0[\overline{u}(0; \mathbf{w})]^2 + \lambda_1[\overline{u}(1; \mathbf{w})]^2) \tag{8}$$

corresponds to the boundary conditions $\overline{u}(0) = \overline{u}(1) = 0$. Note that from Section 2.2, we know that when $\gamma \notin (0, 1)$, the partial loss stated in equation 7 is expected to be nonzero because any minimizer $\overline{u}$ of problem 5 must satisfy $E(\overline{u}) \geq \gamma^2(1-\gamma)^2$. Hence, the trainable penalty coefficients λ_0 and λ_1 are introduced to the neural network so that the network can avoid the type of minimizers where $E(\overline{u}) = 0$ and the boundary conditions of $\overline{u}$ are not satisfied.

Theoretically speaking, the method works for any $\gamma \in \mathbb{R}$; however, in practice, it is found that this method does not work well when $\gamma < 1$. Therefore, other methods are designed to handle the cases when $\gamma < 1$.

3.2 Method 2: For $\gamma \leq 0$

In view of the symmetrical property of problem 5 over $x = 0.5$, the transformation $\hat{u}(x) := \overline{u}(1-x)$ is introduced to handle the case when $\gamma \leq 0$. The entire method is almost the same as method 1 except that the neural network $\hat{u}(x; \mathbf{w})$ is now defined to approximate the function $\hat{u}(x)$. The two partial losses now become

$$\mathcal{L}_\text{e}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} \left|W_\gamma(-\hat{u}'(x_i; \mathbf{w}))\right|, \tag{9}$$

and

$$\mathcal{L}_\text{b}(\mathbf{w}, \lambda_0, \lambda_1) = \frac{1}{2}(\lambda_0[\hat{u}(0; \mathbf{w})]^2 + \lambda_1[\hat{u}(1; \mathbf{w})]^2). \tag{10}$$

3.3 Method 3: For $\gamma \in [0.5, 1]$

Let $u(x)$ be approximated by the output $u_\theta(x)$ of a neural network with input x. Define the approximation network $u(x; \mathbf{w})$, where $\mathbf{w}$ represents the network weights. The weights are trained by minimizing the loss function

$$\mathcal{L}(\mathbf{w}) = \mathcal{L}_{\mathrm{e}}(\mathbf{w}) + \mathcal{L}_{\mathrm{b}}(\mathbf{w}). \tag{11}$$

The first partial loss in 11

$$\mathcal{L}_{\mathrm{e}}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} \left|W(u'(x_i; \mathbf{w}))\right| = \frac{1}{N}\sum_{i=1}^{N} \left|W_\gamma(u'(x_i; \mathbf{w}) - \gamma)\right|, \tag{12}$$

where $\{x_i\}_{i=1}^N$ are N evenly-spaced sample points on $[0, 1]$, enforces the minimization of the integral, while the second partial loss in 11

$$\mathcal{L}_{\mathrm{b}}(\mathbf{w}) = \frac{1}{2}([u(0; \mathbf{w})]^2 + [u(1; \mathbf{w}) - \gamma]^2) \tag{13}$$

corresponds to the boundary conditions $u(0) = 0$ and $u(1) = \gamma$. Note that from Section 2.2, we know that when $\gamma \in [0, 1]$, a minimizer of problem 4 must satisfy $E(u) = 0$. This is true in particular when $\gamma \in [0.5, 1]$. Hence, the partial loss in equation 12 is expected to be zero, and thus no trainable penalty coefficients are introduced in this method. It is found empirically that this method works well when $\gamma \in [0.5, 1]$.

3.4 Method 4: For $\gamma \in [0, 0.5]$

Based on the symmetrical property of the problem again, another equivalent problem of 4 is introduced to handle the remaining cases when $\gamma \in [0, 0.5]$. Let $v(x) = u(1 - x) + x - \gamma$. Then we have $\overline{u}(x) = u(x) - \gamma x = v(1 - x) - (1 - \gamma)(1 - x)$, it is equivalent to consider the following problem:

$$\begin{aligned} \min \quad & E(v) = \int_0^1 W_\gamma(-v'(1 - x) + 1 - \gamma)\, dx \\ \text{s.t.} \quad & v(0) = 0 \text{ and } v(1) = 1 - \gamma. \end{aligned} \tag{14}$$

Then, the entire method is almost the same as method 3 except that a neural network $v(x; \mathbf{w})$ is now defined to approximate the function $v(x)$. The two partial losses are now

$$\mathcal{L}_{\mathrm{e}}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} \left|W_\gamma(-v'(1 - x_i; \mathbf{w}) + 1 - \gamma)\right| \tag{15}$$

and

$$\mathcal{L}_{\mathrm{b}}(\mathbf{w}) = \frac{1}{2}([u(0; \mathbf{w})]^2 + [u(1; \mathbf{w}) - 1 + \gamma]^2). \tag{16}$$

4 Results

Based on the methods discussed in Section 4, four forms of neural networks are implemented to solve problem 4. The adopted architectures and the hyperparameters of the neural networks will be introduced. Then, the accuracy of the methods proposed will be evaluated. After that, this section will investigate how the outputs of the neural networks change when the network architecture is modified. The TensorFlow implementation of the methods is available at the Appendix.

4.1 Network Architecture and Hyperparameters

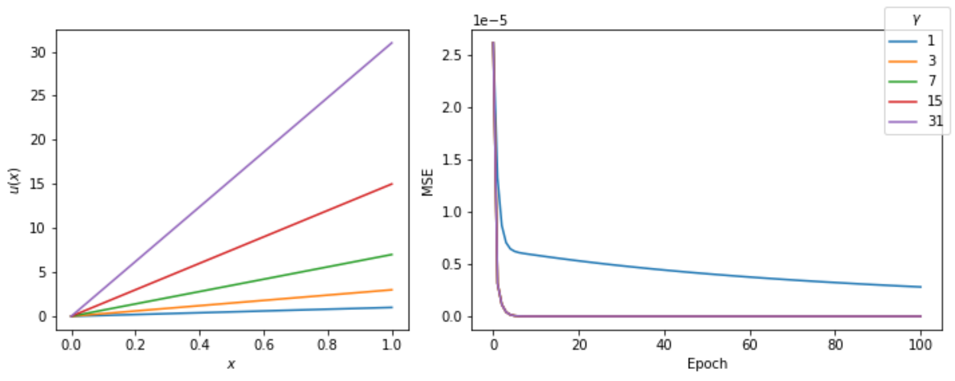
Each of the four methods is implemented with the same following network architecture: Given a set of $N = 128$ evenly-spaced sample points on $[0, 1]$, the specific target function is learned by a neural network with 1 hidden layer using the corresponding loss function. Each hidden layer contains 1 neuron and adopts the Rectified Linear Unit (ReLU) activation function, while the last layer adopts the linear activation function. The Stochastic Gradient Descent (SGD) optimizer with a learning rate of 0.1 is selected to train the network.

To evaluate the accuracy of the network in each epoch, the predicted output of the network first undergoes a suitable transformation to give the function u, then the mean squared error (MSE) between the output function and the expected function (discussed in Section 2.2) is approximated by comparing the function values at 10000 evenly-spaced sample points on $[0, 1]$. Finally, 20000 evenly-spaced sample points on $[0, 1]$ are used to plot the output function generated by the neural network.

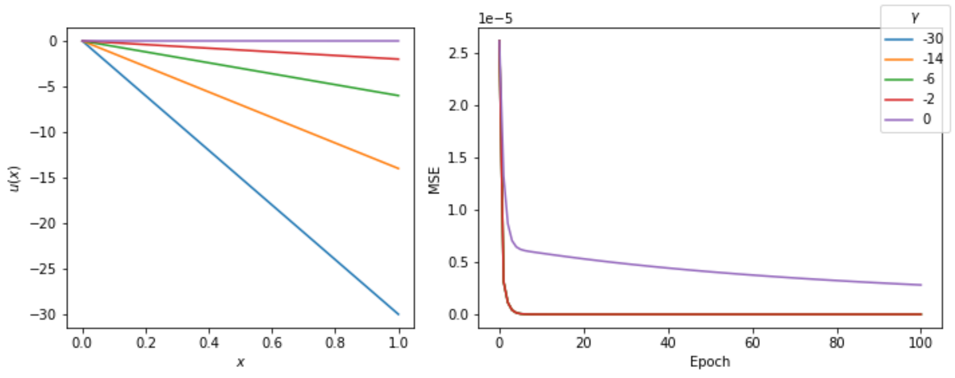
4.2 Network Performance

To demonstrate the performance of the four methods, various γ's ranging from -30 to 31 are selected for experiments. The plots of the output functions generated by the four neural networks built at different γ's, as well as the changes of the MSE during the training process, are shown in Figure 1. As depicted in the figure, the four methods together work for all the selected γ's as the graph of each γ matches the expected function and the MSE converges to 0.

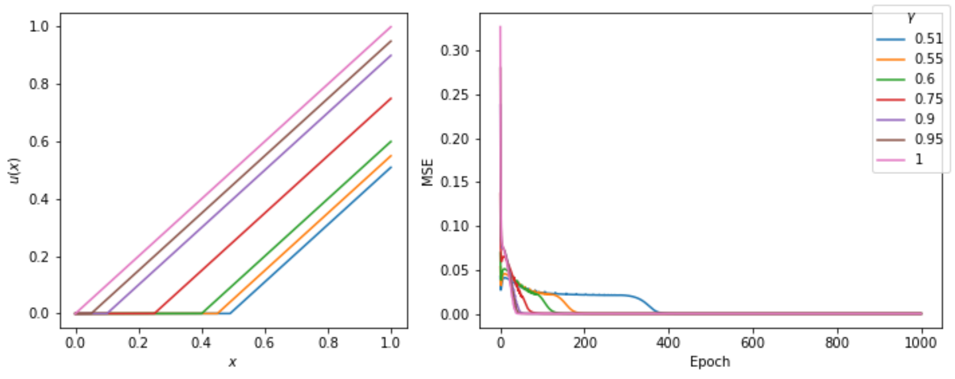
Among all methods, Method 1 and 2 converge faster in general. As shown in Figures 1a and 1b, the MSE of each network using Method 1 or 2 remains within -5 order of magnitude and monotonically decreases during the 100 training epochs. In Method 1 and 2, it is more challenging to learn the function for $\gamma = 0$ and $\gamma = 1$, since the MSEs that correspond to the two γ values decrease at a significantly slower rate. As for Method 3 and 4, the convergence is slower. Moreover, the closer to 0.5 γ is, the slower the convergence is. In particular, for $\gamma = 0.51$ and $\gamma = 0.49$, the MSEs remain at about 0.025 from 100^{th} to 300^{th} epoch, before a significant decrease to a negligible value around the 400^{th} epoch. Note that the case when $\gamma = 0.5$ is not included in the experiment because the minimizer of this problem is not unique, as proved in Section 2.2.



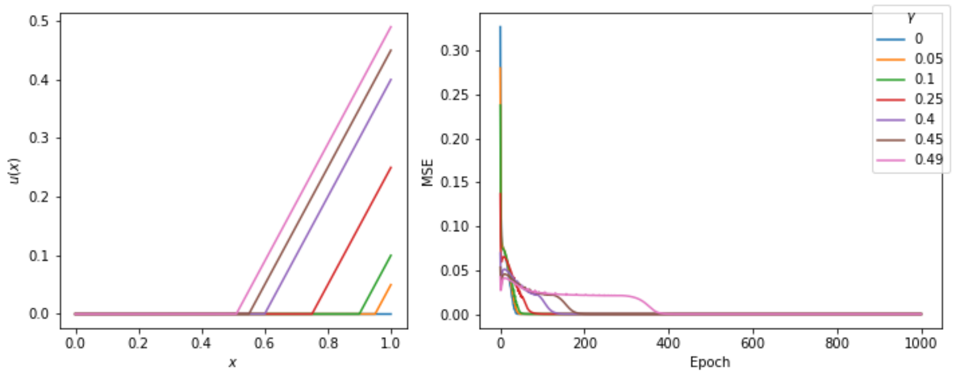
(**a**) Method 1: For $\gamma \geq 1$



(**b**) Method 2: For $\gamma \leq 0$



(**c**) Method 3: For $\gamma \in [0.5, 1]$



(**d**) Method 4: For $\gamma \in [0, 0.5]$

Figure 1: Outputs of the four methods for different γ's.

4.3 Evaluation of the Choice of Hyperparameters

To assess and study the importance of the choice of hyperparameters of the four neural networks, a total of 5 experiments are conducted on each neural network. These experiments investigate how

(1) the number of hidden layers,

(2) the size of the hidden layer,

(3) the type of activation function used,

(4) the type of optimizer employed to train the networks, and

(5) the learning rate of the SGD optimizer

affect the output function of the networks, respectively. In each experiment, all other parameters are taken to be the same as the adopted ones described in Section 4.1 except the variable under testing. In the experiments, the value of γ is set to be 3, -6, 0.75, and 0.25 for Method 1, 2, 3, and 4 respectively.

4.3.1 Number of hidden layers

The current choice is to include 1 hidden layer with 1 neuron. To assess how the number of hidden layers affect the output functions, two other neural networks, which contain 2 and 3 hidden layers respectively with 1 neuron each, are constructed. Then, the output functions are compared after the same training. The result of the experiment is shown in Figure 2. It is found that in each method, the output function of the network converges to the corresponding expected function regardless of the number of hidden layers. Moreover, in Method 3 and 4, the number of epochs required for convergence is fewer when there is only 1 hidden layer. Hence, it is sufficient to include only 1 hidden layer.

4.3.2 Size of hidden layer

To evaluate the adequacy of including only 1 neuron in the only hidden layer, two other neural networks containing 1 hidden layer with 2 and 3 neurons inside the hidden layer respectively are constructed, and the corresponding output functions after the same training are then compared. As shown in Figure 3, the output functions converge at similar rates regardless of the method and the size of the hidden layer concerned. This indicates that it suffices to include 1 neuron in the hidden layer.

4.3.3 Activation function

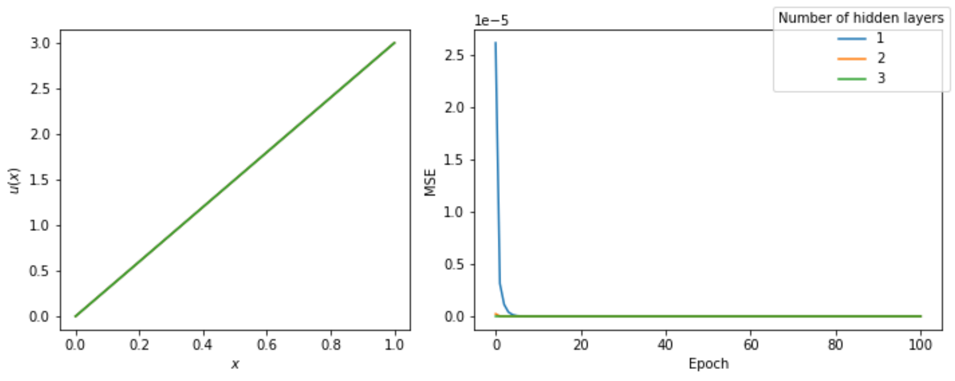
While the ReLU function is adopted in our networks, it is interesting to investigate the performance of the networks when other activation functions are used. The ReLU function aside, the sigmoid, the hyperbolic tangent (tanh), the leaky ReLU, and the ELU functions, which are some commonly used activation functions, are also considered in the experiment. As shown in Figure 4, the ReLU function leads to the expected functions in all methods. The tanh, the leaky ReLU, and the ELU functions do not work in any methods, while the sigmoid function appears to work in Method 2 and no other method. In Method 1 and 2, among the activation functions which fail to give the expected function, the MSE diverges to infinity. Such divergence does not happen in Method 3 or 4, but the outputs given by the sigmoid, the tanh, and the ELU functions do not satisfy the boundary conditions, whereas the output functions corresponding to the leaky ReLU function still differ from the expected functions although the boundary conditions are satisfied. Hence, the ReLU function is the best among the five activation functions for this problem.

4.3.4 Type of optimizer

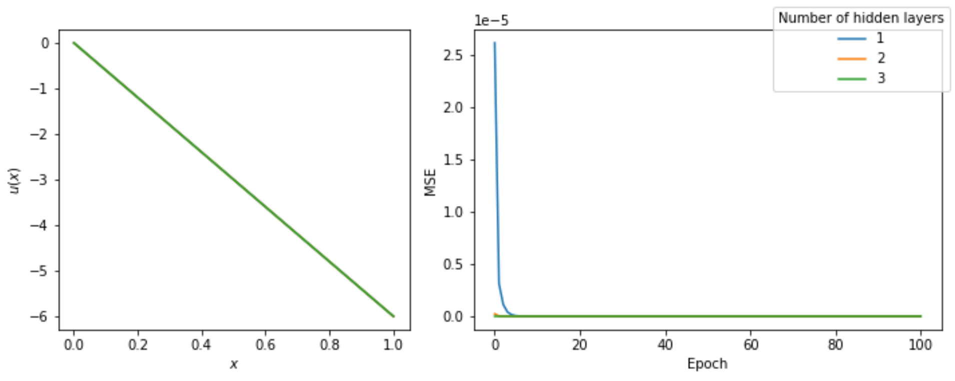
The SGD optimizer with a learning rate of 0.1 in TensorFlow is selected in the four neural networks. To evaluate the performance of the optimizer, three other popular optimizers, including Adagrad, Adam, and RMSprop, are considered in the experiment. In the experiment, the learning rates of Adagrad, Adam, and RMSprop are all set to be 0.001. Figure 5 shows the performance of the four optimizers. In Method 1 and 2, although all four methods lead to the expected output functions, SGD gives a much faster convergence than the other three optimizers do. In addition, in Method 3 and 4, SGD also outperforms the other optimizers. The convergence rate yielded by RMSprop ranks second while that yielded by Adam comes third. The shapes of the output function given by Adagrad in Method 3 and 4 do not resemble those of the expected function, so it is believed that Adagrad would not work for this problem.

4.3.5 Learning rate of SGD optimizer

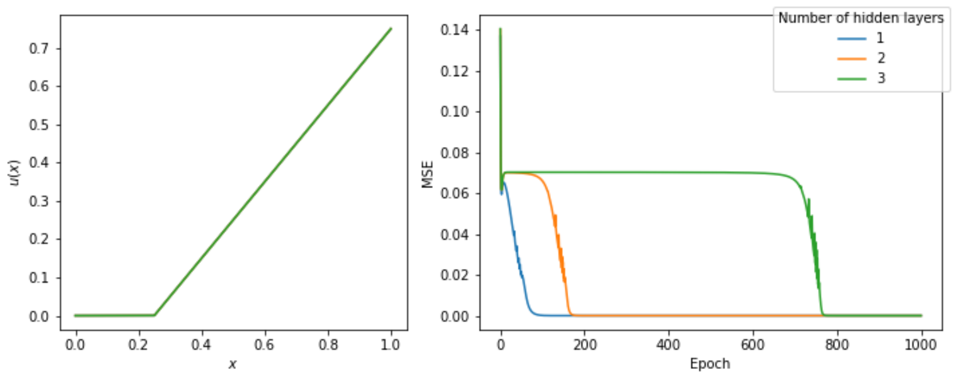
In the last experiment, the effect of the learning rate of the SGD optimizer is assessed. The adopted learning rate is 0.1, while the learning rates 0.05 and 0.2 are considered for comparison. Figure 6 illustrates the result of the experiment. The effect of the learning rate is not significant in Method 1 and 2, while the convergence rate increases with the learning rate in Method 3 and 4. Yet, the learning rate of 0.1 is still an acceptable choice. Indeed, the output function converges to the expected function regardless of the learning rate and the method concerned.



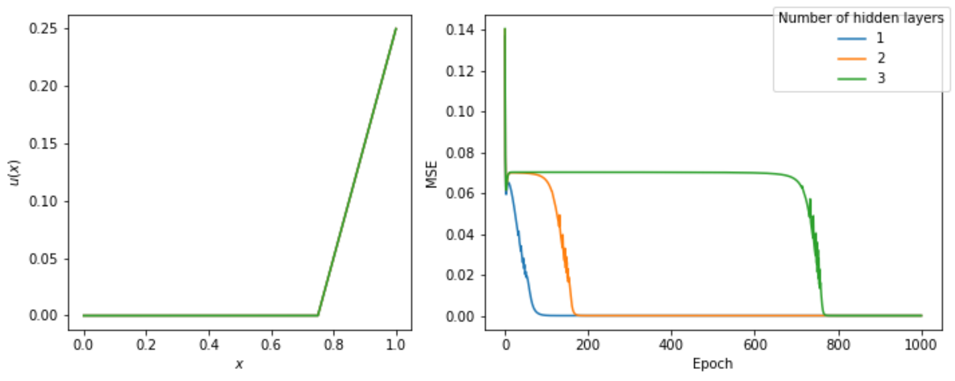
(**a**) Method 1: For $\gamma = 3$



(**b**) Method 2: For $\gamma = -6$

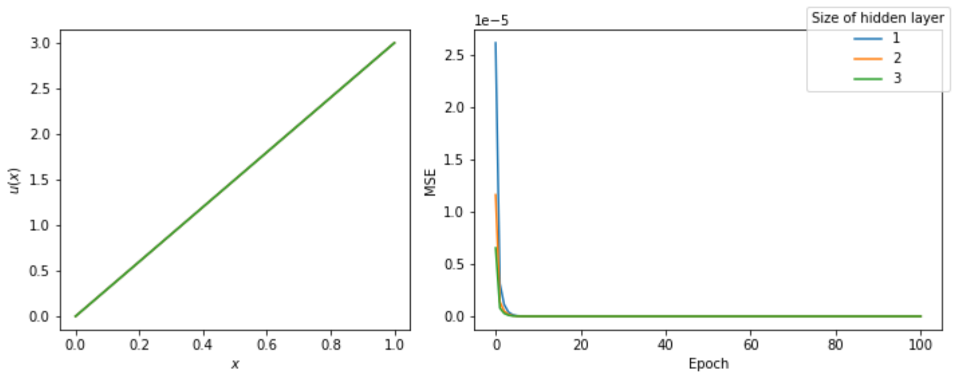


(**c**) Method 3: For $\gamma = 0.75$

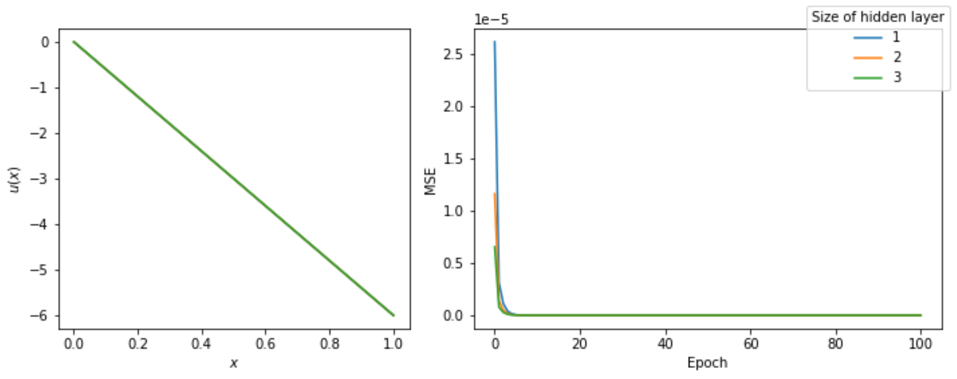


(**d**) Method 4: For $\gamma = 0.25$

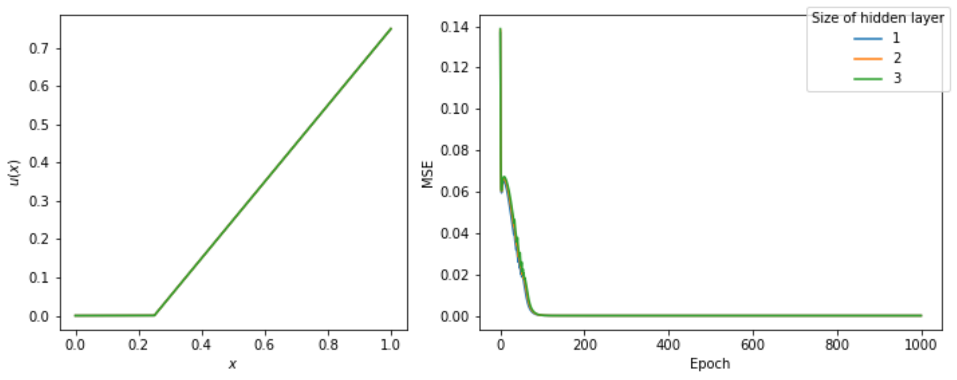
Figure 2: Outputs of the four methods using different number of hidden layers.



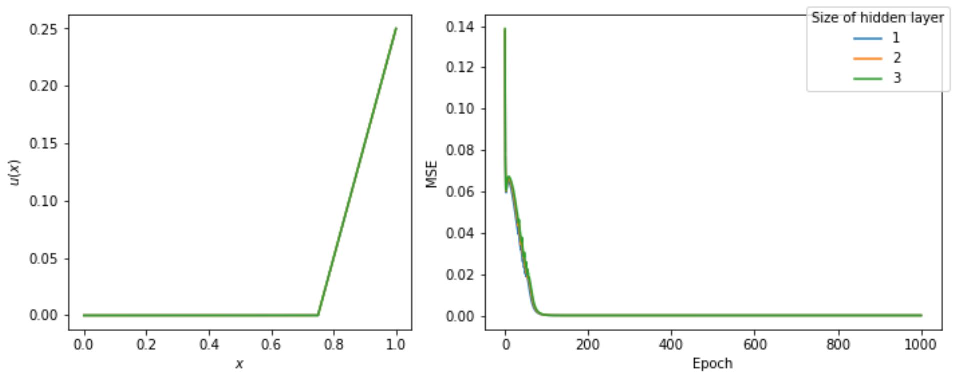
(**a**) Method 1: For $\gamma = 3$



(**b**) Method 2: For $\gamma = -6$

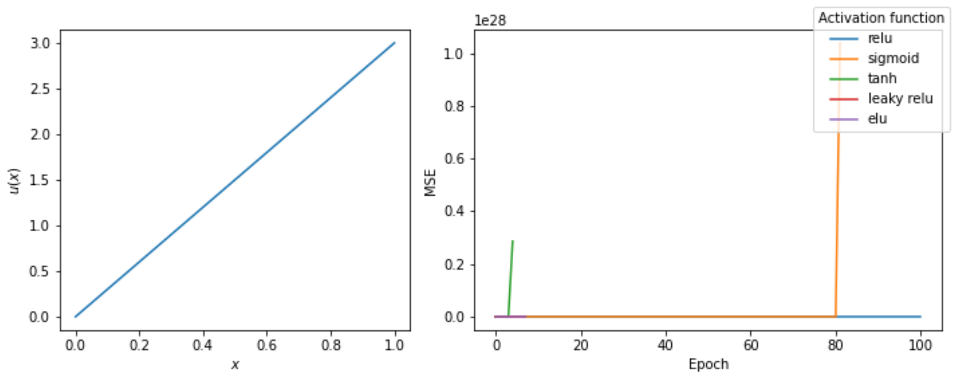


(**c**) Method 3: For $\gamma = 0.75$

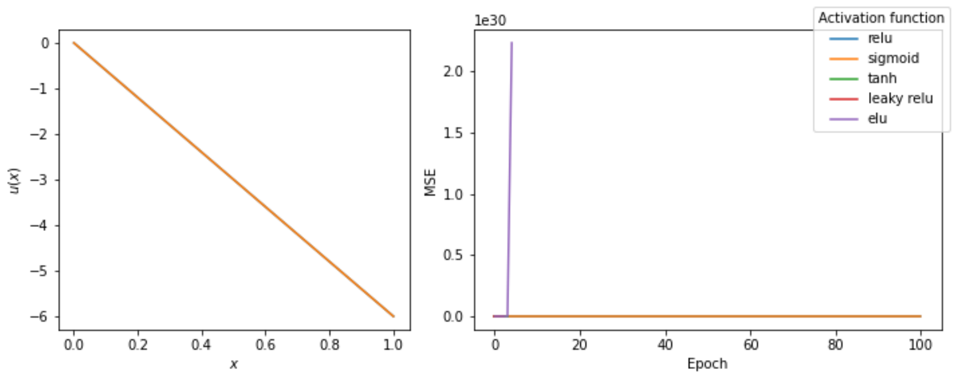


(**d**) Method 4: For $\gamma = 0.25$

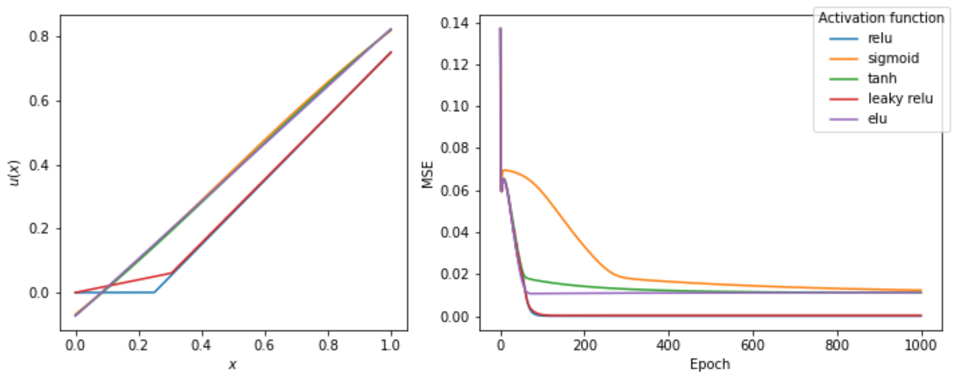
Figure 3: Outputs of the four methods using different size of hidden layer.



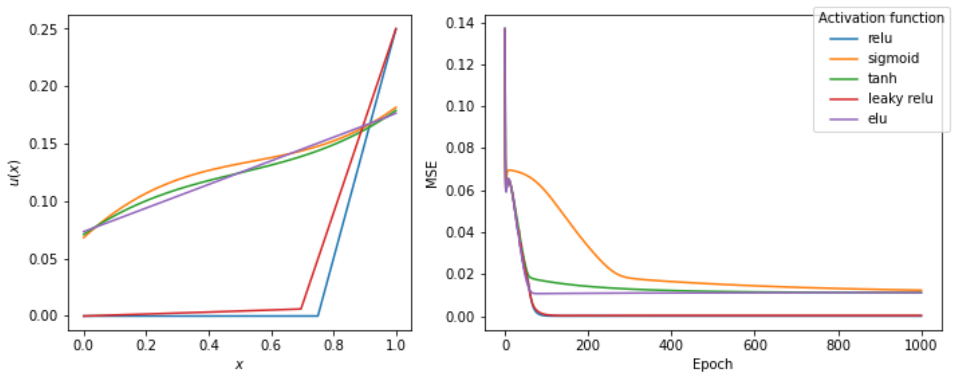
(**a**) Method 1: For $\gamma = 3$



(**b**) Method 2: For $\gamma = -6$

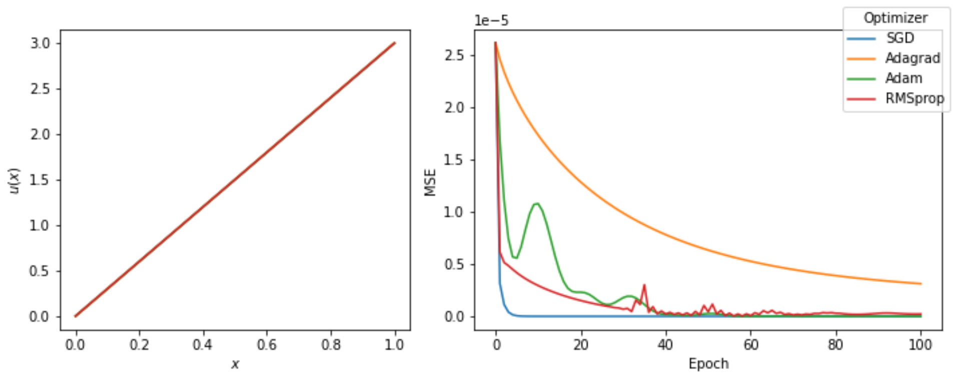


(**c**) Method 3: For $\gamma = 0.75$

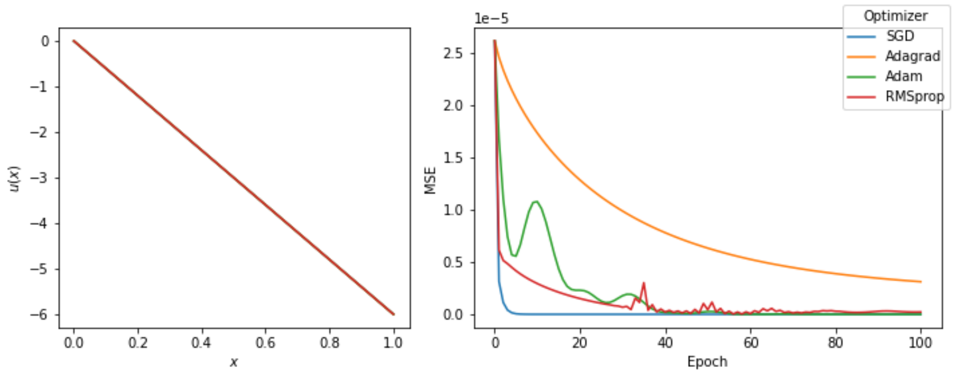


(**d**) Method 4: For $\gamma = 0.25$

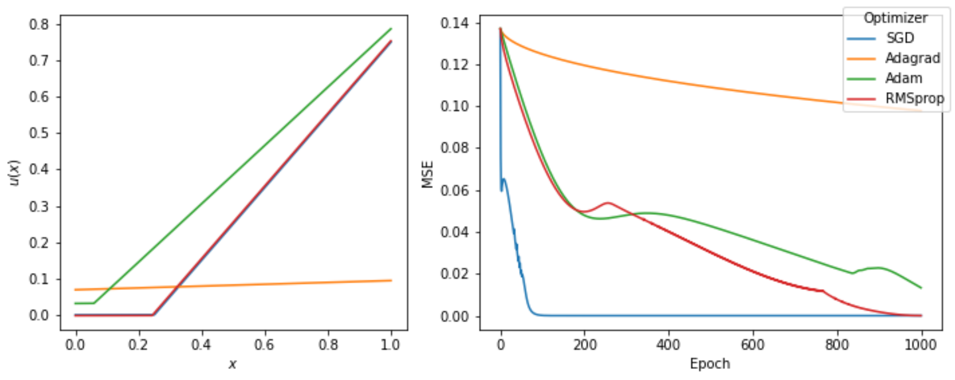
Figure 4: Outputs of the four methods using different activation functions.



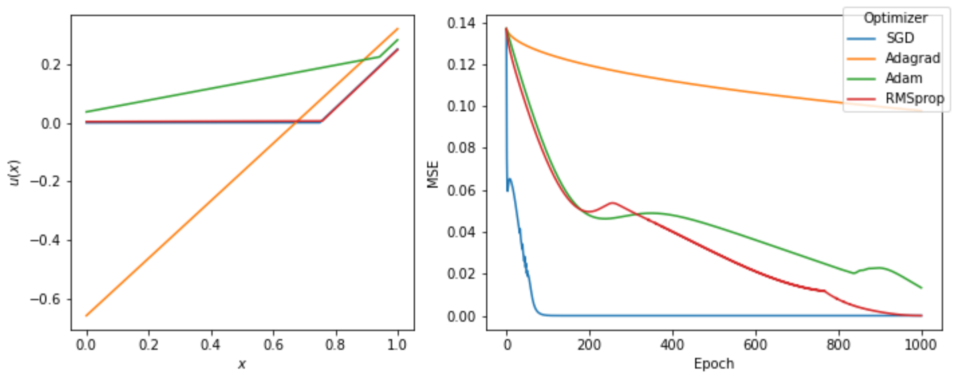
(**a**) Method 1: For $\gamma = 3$



(**b**) Method 2: For $\gamma = -6$

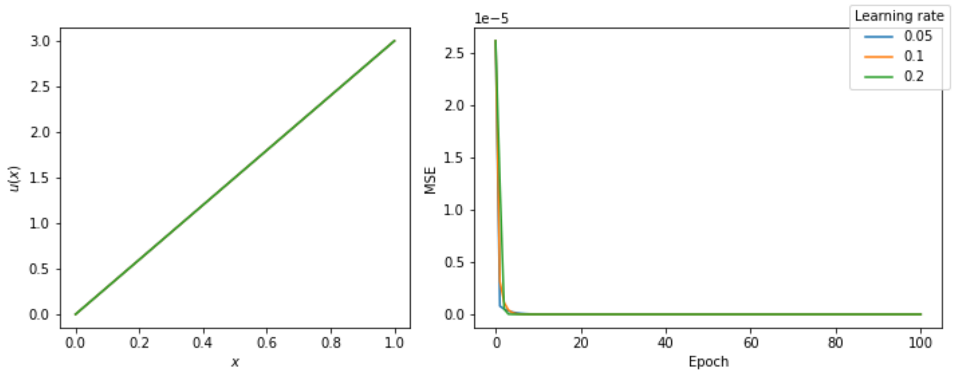


(**c**) Method 3: For $\gamma = 0.75$

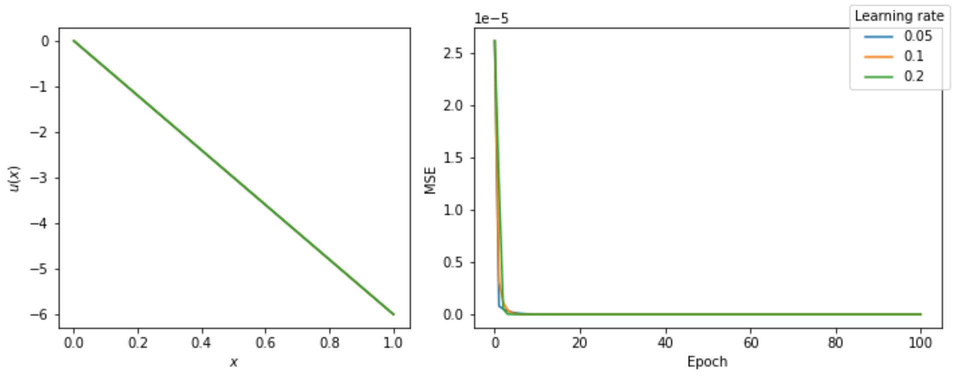


(**d**) Method 4: For $\gamma = 0.25$

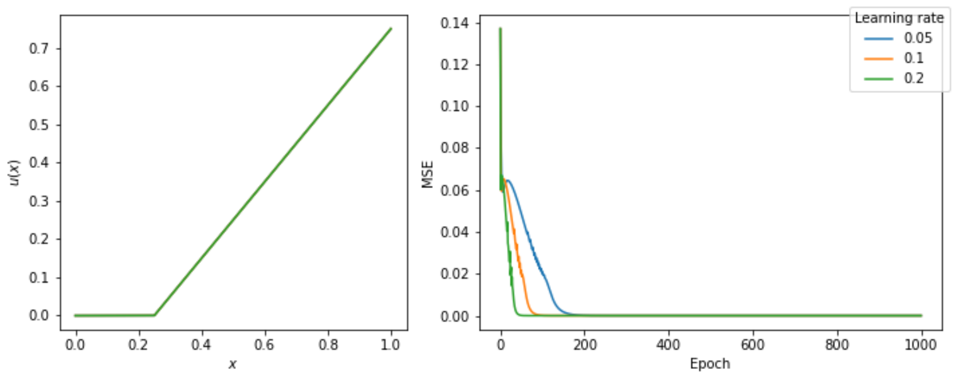
Figure 5: Outputs of the four methods using different optimizers.



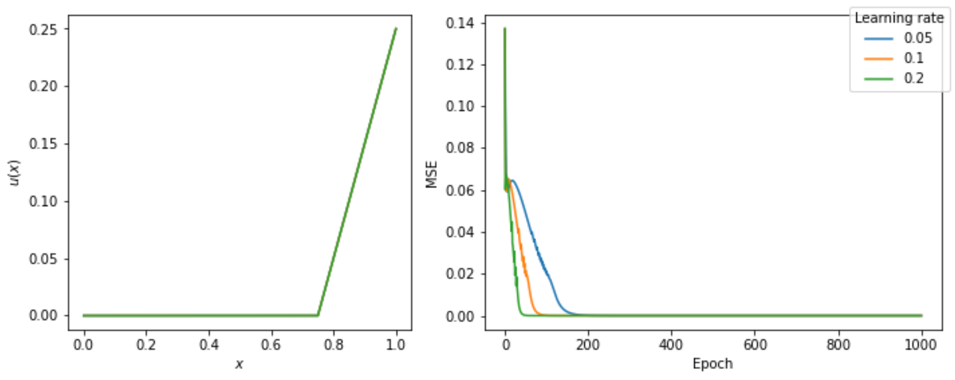
(**a**) Method 1: For $\gamma = 3$



(**b**) Method 2: For $\gamma = -6$



(**c**) Method 3: For $\gamma = 0.75$



(**d**) Method 4: For $\gamma = 0.25$

Figure 6: Outputs of the four methods using different optimizers.

5 Conclusion

Potential energy minimization is an essential component in designing new molecules, including elastic crystals. Based on the generally adopted integral formula of the potential energy of elastic crystals, an energy minimization problem is defined. The objective of the problem is to find a deformation function subject to some Dirichlet boundary conditions such that the resulting potential energy of the crystals is minimized.

To investigate the possibility of employing neural networks to solve the problem, a particular family of problems in 1 dimension is defined in 4. As the PINNs framework has been a promising application of deep neural networks to numerically solve PDEs [5], this project takes a step forward to solve the set of 1-dimensional energy minimization problems using the Self Adaptive PINNs framework proposed in [6]. The modified loss functions are designed to enforce both the energy minimization and the boundary conditions of the problem.

Four neural networks, which differ by their target output functions and hence the loss functions, are designed to solve the problems. Through experiments of the methods in TensorFlow, it is then found that the four networks together can deduce the optimal deformation of the problems in the family if the network parameters are selected appropriately. Future work is thus to take this approach to solve more complex energy minimization problems in higher dimensions.

References

[1] P. Naumov, D. P. Karothu, E. Ahmed, L. Catalano, P. Commins, J. Mahmoud Halabi, M. B. Al-Handawi, and L. Li, "The rise of the dynamic crystals," *Journal of the American Chemical Society*, vol. 142, no. 31, pp. 13 256–13 272, 2020, pMID: 32559073. [Online]. Available: https://doi.org/10.1021/jacs.0c05440

[2] E. Lewars, *Computational Chemistry: Introduction to the Theory and Applications of Molecular and Quantum Mechanics*. Springer, 2011. [Online]. Available: https://doi.org/10.1007/978-90-481-3862-3

[3] D. H. J. Mackay, A. J. Cross, and A. T. Hagler, *The Role of Energy Minimization in Simulation Strategies of Biomolecular Systems*. Boston, MA: Springer US, 1989, pp. 317–358. [Online]. Available: https://doi.org/10.1007/978-1-4613-1571-1_7

[4] X. Blanc, C. Bris, and P.-L. Lions, "From molecular models to continuum mechanics," *Archive for Rational Mechanics and Analysis*, vol. 164, pp. 341–381, 2002. [Online]. Available: https://doi.org/10.1007/s00205-002-0218-5

[5] M. Raissi, P. Perdikaris, and G. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational Physics*, vol. 378, 2018. [Online]. Available: https://doi.org/10.1016/j.jcp.2018.10.045

[6] L. McClenny and U. Braga-Neto, "Self-adaptive physics-informed neural networks using a soft attention mechanism," 2020. [Online]. Available: https://arxiv.org/abs/2009.04544

[7] H. Whitney, "Analytic extensions of differentiable functions defined in closed sets," *Transactions of the American Mathematical Society*, vol. 36, no. 1, pp. 63–89, 1934. [Online]. Available: http://www.jstor.org/stable/1989708

Appendix: TensorFlow Implementation of the Four Methods

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import gridspec
import time
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers, initializers

def init_model(num_layers, layer_size, N_train, activation):
    layer_sizes = [1] + [layer_size for i in range(num_layers)] + [1]
    model = neural_net(layer_sizes, activation)
    lb = np.array([0.0])
    ub = np.array([1.0])
    x_lb = tf.convert_to_tensor(lb, dtype=tf.float32)
    x_ub = tf.convert_to_tensor(ub, dtype=tf.float32)
    x = lb + (ub-lb)*np.reshape(np.linspace(0,1,N_train),(N_train,1))
    x = tf.convert_to_tensor(x, dtype=tf.float32)
    weights = tf.Variable([[1.0],[1.0]])
    return model, x, x_lb, x_ub, weights

def u_exact(gamma, x):
    u = np.zeros(x.shape)
    if (gamma <= 0 or gamma >= 1):
        for i in range(u.shape[0]):
            u[i,0] = x[i,0] * gamma
    else:
        for i in range(u.shape[0]):
            if (x[i,0] > 1 - gamma):
                u[i,0] = x[i,0] - 1 + gamma
            else:
                u[i,0] = 0
    return u

def neural_net(layer_sizes, activation):
    model = Sequential()
    model.add(layers.InputLayer(input_shape=(layer_sizes[0],)))
    for width in layer_sizes[1:-1]:
        model.add(layers.Dense(width, activation=activation,
                               kernel_initializer=initializers.glorot_normal(seed=21)))
    model.add(layers.Dense(layer_sizes[-1], activation=None,
                           kernel_initializer=initializers.glorot_normal(seed=21)))
    return model
```

```
def fit(model, gamma, method, x, x_lb, x_ub, weights, tf_optimizer, N_test, epochs):
    x_eva = np.linspace(x_lb, x_ub, N_test)
    exact = u_exact(gamma, x_eva)
    mse = np.zeros(epochs+1)
    MSE = tf.keras.losses.MeanSquaredError()
    start_time = time.time()
    print("Starting␣training")

    for epoch in range(epochs):
        pred = predict(model, gamma, x_eva)
        u_pred = output_transform(model, gamma, method, x_eva, pred)
        mse[epoch] = MSE(exact, u_pred)
        if (epoch%100 == 0):
            elapsed = time.time() - start_time
            print('Iter:␣%d,␣Time:␣%.2f,␣mse:␣%.10f' % (epoch, elapsed, mse[epoch]))
            start_time = time.time()

        with tf.GradientTape(persistent=True) as tape:
            loss_value = loss(model, gamma, method, x, x_lb, x_ub, weights)
            grads = tape.gradient(loss_value, model.trainable_variables)
            grads_weights = tape.gradient(loss_value, weights)
        tf_optimizer.apply_gradients(zip(grads, model.trainable_variables))
        if gamma < 0 or gamma > 1:
            tf_optimizer.apply_gradients(zip([-grads_weights], [weights]))
        del tape

    pred = predict(model, gamma, x_eva)
    u_pred = output_transform(model, gamma, method, x_eva, pred)
    mse[epochs] = MSE(exact, u_pred)
    elapsed = time.time() - start_time
    print('Iter:␣%d,␣Time:␣%.2f,␣mse:␣%.10f' % (epochs, elapsed, mse[epoch]))

    return model, mse

def predict(model, gamma, x):
    x = tf.convert_to_tensor(x, dtype=tf.float32)
    u = model(x)
    return u.numpy()

def output_transform(model, gamma, method, x, pred):
    if (method == 1):           # output transform u(x) = u_bar(x) + gamma*x
        return pred + gamma*x
    elif (method == 2):         # output transform u(x) = u_bar_hat(1-x) + gamma*x
        pred_rev = np.flip(pred)
        return pred_rev + gamma*x
    elif (method == 3):         # no transform required
        return pred
    elif (method == 4):         # output transform u(x) = v(1-x) + x - 1 + gamma
        pred_rev = np.flip(pred)
        return pred_rev + x - (1-gamma)*np.ones((x.shape[0],1))

def loss(model, gamma, method, x, x_lb, x_ub, weights):
    energy = E_u(model, gamma, method, x)
    pred = model(np.array([x_lb,x_ub]))
    if (method == 1 or method == 2):
        loss_b = tf.reduce_mean(weights[0]*tf.square(pred[0])+
                                weights[1]*tf.square(pred[1]))
    elif (method == 3):
        loss_b = tf.reduce_mean(weights[0]*tf.square(pred[0])+
                                weights[1]*tf.square(pred[1]-gamma))
    elif (method == 4):
        loss_b = tf.reduce_mean(weights[0]*tf.square(pred[0])+
                                weights[1]*tf.square(pred[1]-1+gamma))
    loss_e = tf.reduce_mean(energy)
    return  loss_e + loss_b

def E_u(model, gamma, method, x):
    with tf.GradientTape(persistent=True) as tape:
        tape.watch(x)
        if (method == 1):
            u_bar = model(x)
        elif (method == 2):
            u_bar = model(1-x)
        elif (method == 3):
            u_bar = model(x) - gamma*x
        elif (method == 4):
            u_bar = model(1-x) - (1-gamma)*(1-x)
        u_bar_x = tape.gradient(u_bar, x)
    ones = tf.convert_to_tensor(np.ones(u_bar_x.shape), dtype=tf.float32)
    w = (u_bar_x+gamma*ones)**2 * ((1-gamma)*ones-u_bar_x)**2
    del tape
    return w

def plot(models, gamma, method, num_pts, mses, labels, legend_title, epochs, variable=''):
    fig = plt.figure(figsize=(10,4))
    spec = gridspec.GridSpec(ncols=2, nrows=1, width_ratios=[3, 4])
    ax1 = fig.add_subplot(spec[0])
    ax2 = fig.add_subplot(spec[1])
    x_plt = np.linspace(0,1,num_pts)
    x_plt_ = np.reshape(x_plt,(num_pts,1))
    for i in range(len(models)):
        if (variable=='gamma'):
            u = output_transform(models[i], gamma[i], method, x_plt_,
                                 predict(models[i], gamma[i], x_plt_))
        else:
            u = output_transform(models[i], gamma, method, x_plt_,
                                 predict(models[i], gamma, x_plt_))
        ax1.plot(x_plt, u)
    ax1.set_xlabel(r'$x$')
    ax1.set_ylabel(r'$u(x)$')

    t = epochs
    for i in range(len(mses)):
        if (labels is not None):
            ax2.plot(np.linspace(0,t,t+1), mses[i], label=str(labels[i]))
        else:
            ax2.plot(np.linspace(0,t,t+1), mses[i])
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('MSE')

    if (legend_title=='gamma'):
        fig.legend(loc='upper␣right', title=r'$\gamma$')
    elif (legend_title is not None):
        fig.legend(loc='upper␣right', title=legend_title)
    fig.tight_layout()
    fig.show()
    if (legend_title is not None):
        fig.savefig(str(method)+str(legend_title))
    else:
        fig.savefig(str(method))

# Example runs (Method 1, gamma = 3)
method, gamma = 1, 3
N_train, N_test = 128, 10000
num_layers, layer_size = 1, 1
activation_func = 'relu'
optimizer = tf.keras.optimizers.SGD(lr=0.1)
epochs = 1000
model, x, x_lb, x_ub, weights = init_model(num_layers, layer_size, N_train, activation_func)
model, mse = fit(model, gamma, method, x, x_lb, x_ub, weights, optimizer, N_test, epochs)
plot([model], [gamma], method, 20000, [mse], None, None, epochs, variable='gamma')
```