Tutorial for conditional generative adversarial network

Xiaoyang Zheng [1,2], Ikumu Watanabe [1,2]

[1] Graduate School of Pure and Applied Sciences, University of Tsukuba, 1-1-1 Tennodai, Tsukuba 305-8573, Japan

[2] Research Center for Structural Materials, National Institute for Materials Science, 1-2-1 Sengen, Tsukuba 305-0047, Japan

Emails: ZHENG.Xiaoyang@nims.go.jp (X.Z.); WATANABE.Ikumu@nims.go.jp (I.W.)

Abstract

This tutorial aims to give an introduction of how to use a deep generative model, conditional generative adversarial network (CGAN). The CGAN can be used for the inverse design of 2D and 3D microstructures with target properties. The CGAN is trained with supervised learning using a labeled dataset. The dataset consists of a large number of geometries and their corresponding properties (e.g., elastic moduli). After training, the CGAN can generate a batch of geometries using target properties at inputs. In our previous two papers, we have demonstrated how to use the CGAN for the inverse design of 2D auxetic metamaterials and 3D architected materials [1,2]. We hope this tutorial can be useful for those who are interested in the inverse design problems of microstructures.

1. Introduction

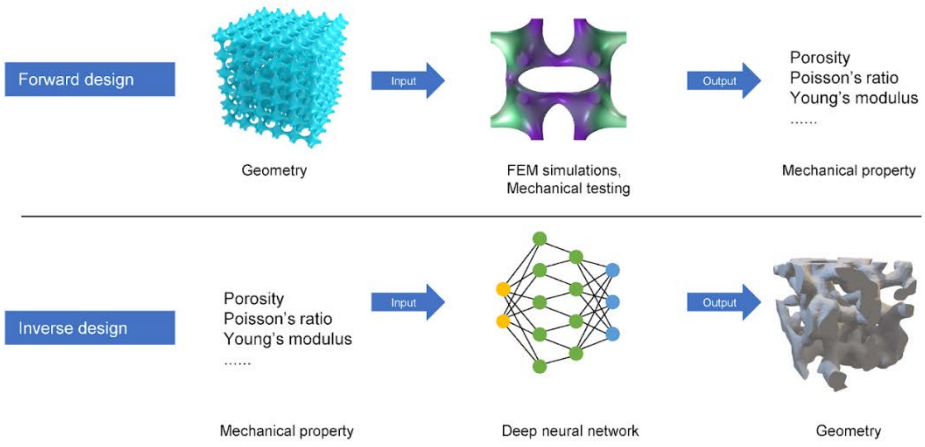

Figure 1. Forward design and inverse design

Forward design is a conventional approach to design microstructures, such as architected materials, mechanical metamaterials, lattices, etc. This forward design approach follows a general process: a structure is created firstly and then its mechanical properties are investigated by finite element simulation or mechanical testing (Figure 1). The mechanical properties of designed materials will be only known after time-consumingly simulations or experiments. In contrast, deep generative models, such as GAN, enable inverse design of microstructures. In inverse design, microstructures can be automatically generated by inputting target properties to a deep generative model, which outputs corresponding geometries of microstructures.

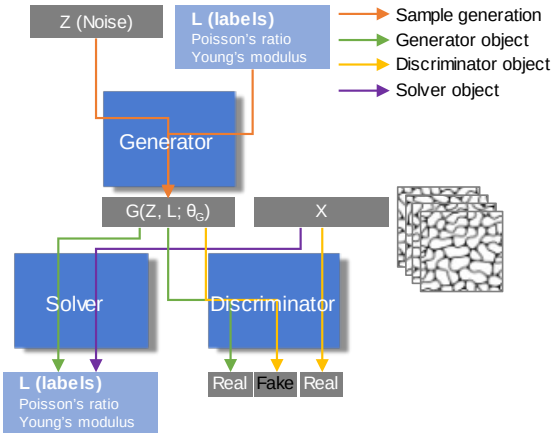

Figure 2. Architecture of CGAN

In our previous studies, we proposed an improved deep generative model, CGAN, for the inverse design of 2D auxetic metamaterials and 3D architected materials [1,2]. The CGAN is composed of three neural network structures: a generator, a discriminator, and a solver (Figure 2). The generator is trained to generate the realistic geometries from latent variables (multivariate normal distribution) and user-defined labels (target properties, e.g., elastic moduli), and simultaneously aims to deceive the discriminator and the solver. The discriminator is trained to distinguish geometries produced by the generator from the real dataset. The solver is trained to predict the properties of a given geometry. CGAN is optimized by a minimax game in the following equations:

$$\hat{\theta}_D = \arg\min_{\theta_D}\{L_D\big(t_D, D(\mathbf{X};\theta_D)\big) + L_D(u_D, D(G(\mathbf{Z},\mathbf{L};\theta_G);\theta_D))\} \quad (1)$$

$$\hat{\theta}_G = \arg\max_{\theta_D}\{L_D\big(u_D, D(G(\mathbf{Z},\mathbf{L};\theta_G);\theta_D)\big) - \alpha \cdot L_S(L, S(G(\mathbf{Z},\mathbf{L};\theta_G);\theta_S))\} \quad (2)$$

$$\hat{\theta}_S = \arg\min_{\theta_S}\{L_S\big(L, S(\mathbf{X};\theta_S)\big)\} \quad (3)$$

Where θ_D, θ_G, θ_S are sets of parameters of the discriminator, generator, and solver, respectively. D, G, and S denotes the discriminator, generator, and solver, respectively. $\mathbf{X} \in \mathbb{R}^{n\times p}$ is the training dataset (vectors of auxetic metamaterials), $\mathbf{L} \in \mathbb{R}^{n\times l}$ is the labels of the dataset $\mathbf{X}$ (i.e., Young's moduli and Poisson's ratios), and $\mathbf{Z} \in \mathbb{R}^{n\times l}$ is latent variables from a multivariate normal distribution during each iteration. L_D is a loss function (binary cross-entropy function) for the discriminator. t_D and u_D are target labels and generally set at as one and zero, respectively.

However, we apply label smoothing technique for the target labels: t_D is replaced with a random number between 0.7 and 1.2, and u_D is replaced with a random number between 0 and 0.3. The moderating weights, α, determines how much the generator focuses on the training of input labels, and is set to be 0.1 in our study.

The deep learning calculations are performed using TensorFlow. Adam optimizer with learning rate of 0.0001 and β_1 of 0.5 is used to train the model. The batch size for training is set to be 32. The detailed network structures used in this study are provided in Tables S1–S3. In short, the used layers include 2D convolutional layer, 2D transposed convolutional layer, 2D max pooling, fully connected layer, batch normalization, and dropout, and the used activation functions include leakyRELU and tanh. Note that circular padding is used in 2D convolutional layer and 2D transposed convolutional layer in order to keep and identify the periodicity of the patterns.

Note that before using the CGAN, you need to prepare a labeled dataset consisting of 2D geometries and their corresponding properties (e.g., elastic moduli). The 2D geometries are represented using 2D images with shape of [b, 256, 256] and the properties are represented by two values with shape of [b, 2]. The letter "b" means the total number of geometries. The total number of geometries varies case by case, depending on the complexity and type of the geometries. In generally, the bigger the dataset, the better performance the training result. In our previous studies, we used 100,000 datapoints for 2D case and 10,000 datapoints for 3D case. The most convenient way is to generate geometries using code-based CAD modeling and calculate their properties using finite element simulations.

2. Procedures of CGAN training

2.1 Solver training

As the solver is independent of the generator and discriminator, we firstly train the solver with supervised learning. Follow the steps below:

a. Open *solver.py* with, e.g., PyCharm
b. Change the source file in line 178 dataset_matrixes = np.load("yourdataset_geometries.npy"). The shape of "yourdataset_geometries.npy" should be [b,256,256]. b means the total number of geometries, and 256 means the height and width of a geometry image. "yourdataset_geometries.npy" consists of 0s and 1s, where 0 represents void part and 1 represents solid part.

Note that the dataset will be divided with 80% for training and 20% for testing. (lines 182-185)

The batch size is set to 32, and training epoch is 200.

The comparison between the real and predicted values is plotted using line 221.

The performance is investigated using the mean square error (MSE) of testing dataset using line 243.

The check points (trained weights and biases) are saved using line 251.

The check points will be used to train the generator and discriminator.

2.2 Generator and discriminator training

After training the solver, we can use the saved weights of the solver to train the generator and discriminator. Follow the steps below:

a. Open CGAN_main.py with, e.g., PyCharm
b. Change the source file in line 292 dataset_matrixes = np.load("yourdataset_geometries.npy"). The shape of “yourdataset_geometries.npy” should be [b,256,256]. b means the total number of geometries, and 256 means the height and width of a geometry image. “yourdataset_geometries.npy” consists of 0s and 1s, where 0 represents void part and 1 represents solid part.
c. Change the source file of saved weights in line 307 solver.load_weights(r"/ckpt/solver_199.ckpt")
d. Change the limit of your dataspace in lines 289 and 290. For example, if your dataspace covers a square of [0,1], you can use current one. If not, e.g., a circle or triangle shape, you need to change the way to sample a datapoint from your dataspace.

Note that the loss functions and MSE will be saved using lines 374-377.

The weighs of the generator and the discriminator will be saved using lines 380 and 381.

2.3 Using trained CGAN for inverse design

After training the solver, generator, and discriminator, we can use the saved weights of the solver and generator for the inverse design. Follow the steps below:

a. Open generate_geometies_using_trained_cgan.py with, e.g., PyCharm
b. Change the source file of saved weights of generator in line 145 generator.load_weights(r'ckpt/generator_199.ckpt')
c. Change the source file of saved weights of solver in line 147 solver.load_weights(r"/ckpt/solver_199.ckpt")
d. Change your target property for the condition in line 158

Note that the comparison between the input and output values will be save in the form of figures using line 171. The images of the generated 2D geometries will be saved using line 173.

3. Conclusions

We give a tutorial of how to use CGAN for the inverse design of 2D geometries. After suitable training, the CGAN can take target properties as inputs and output corresponding 2D geometries. If you would like to use CGAN for the inverse design of 3D geometries, you can refer our latest paper [2], in which we proposed a 3D-CGAN that has a similar architecture with CGAN. If you meet any problems or have any comments, just feel free to contact us via emails. Hope you can enjoy our codes.

Reference

1. Zheng X, Chen TT, Guo X, Samitsu S, Watanabe I. Controllable inverse design of auxetic metamaterials using deep learning. Materials & Design. 2021 Dec 1;211:110178.
2. Zheng X, Chen TT, Jiang X, Naito M, Watanabe I. Deep-learning-based inverse design of three-dimensional architected cellular materials with the target porosity and stiffness using voxelized Voronoi lattices. Science and Technology of Advanced Materials. 2023 Dec 31;24(1):2157682.

Table S1: Network architecture of generator.

Description	Kernel size	Resampling	Input shape	Output shape
Concatenate(Z, L)	-	-	128+2	130
Fully connected + Batch Normalization + Reshape	-	-	130	4×4×512
2D Transposed convolution + Batch Normalization + Leaky ReLU	4×4	Up	4×4×512	8×8×256
2D Transposed convolution + Batch Normalization + Leaky ReLU	4×4	Up	8×8×256	16×16×128
2D Transposed convolution + Batch Normalization + Leaky ReLU	4×4	Up	16×16×128	32×32×64
2D Transposed convolution + Batch Normalization + Leaky ReLU	4×4	Up	32×32×64	64×64×32
2D Transposed convolution + Batch Normalization + Leaky ReLU	4×4	Up	64×64×32	128×128×16
2D Transposed convolution	4×4	Up	128×128×16	256×256×1
Tanh	-	-	256×256×1	256×256×1

Table S2: Network architecture of discriminator.

Description	Kernel size	Resampling	Input shape	Output shape
2D convolution + Leaky ReLU + Dropout	4×4	Down	256×256×1	128×128×16
2D convolution + Leaky ReLU + Dropout	4×4	Down	128×128×16	64×64×32
2D convolution + Leaky ReLU + Dropout	4×4	Down	64×64×32	32×32×64
2D convolution + Leaky ReLU + Dropout	4×4	Down	32×32×64	16×16×128
2D convolution + Leaky ReLU + Dropout	4×4	Down	16×16×128	8×8×256
2D convolution + Leaky ReLU + Dropout	4×4	Down	8×8×256	4×4×512
Flatten	-	-	4×4×512	8192
Fully connected	-	-	8192	1

Table S3: Network architecture of solver.

	Description	Kernel size / pool size	Resampling	Input shape	Output shape
	2D convolution	3×3	-	256×256×1	256×256×16
Unit 1	2D convolution	3×3	-	256×256×16	256×256×16
	2D Max pooling	2×2	Down	256×256×1	128×128×16
	2D convolution	3×3	-	128×128×16	128×128×32
Unit 2	2D convolution	3×3	-	128×128×32	128×128×32
	2D Max pooling	2×2	Down	128×128×32	64×64×32
	2D convolution	3×3	-	64×64×32	64×64×64
Unit 3	2D convolution	3×3	-	64×64×64	64×64×64
	2D Max pooling	2×2	Down	64×64×64	32×32×64
	2D convolution	3×3	-	32×32×64	32×32×128
Unit 4	2D convolution	3×3	-	32×32×128	32×32×128
	2D Max pooling	2×2	Down	32×32×128	16×16×128
	2D convolution	3×3	-	16×16×128	16×16×256
Unit 5	2D convolution	3×3	-	16×16×256	16×16×256
	2D Max pooling	2×2	Down	16×16×256	8×8×256
	2D convolution	3×3	-	8×8×256	8×8×384
Unit 6	2D convolution	3×3	-	8×8×384	8×8×384
	2D Max pooling	2×2	Down	8×8×384	4×4×384
	2D convolution	3×3	-	4×4×384	4×4×512
Unit 7	2D convolution	3×3	-	4×4×512	4×4×512
	2D Max pooling	2×2	Down	4×4×512	2×2×512
	2D convolution	3×3	-	2×2×512	2×2×512
Unit 8	2D convolution	3×3	-	2×2×512	2×2×512
	2D Max pooling	2×2	Down	2×2×512	1×1×512
	Flatten + Fully connected	-	-	1×1×512	256
	Fully connected			256	128
	Fully connected	-	-	128	2