# **Generative Adversarial Networks: Build Your First Models**

by [Renato Candido](https://realpython.com/generative-adversarial-networks/#author) [11 Comments](https://realpython.com/generative-adversarial-networks/#reader-comments) [**advanced**](https://realpython.com/tutorials/advanced/) [**machine-learning**](https://realpython.com/tutorials/machine-learning/)

Mark as Completed

[**Tweet**](https://twitter.com/intent/tweet/?text=Check%20out%20this%20%23Python%20tutorial:%20Generative%20Adversarial%20Networks%3A%20Build%20Your%20First%20Models%20by%20@realpython&url=https%3A//realpython.com/generative-adversarial-networks/) [**Share**](https://facebook.com/sharer/sharer.php?u=https%3A//realpython.com/generative-adversarial-networks/) **Email**

Table of Contents

* [What Are Generative Adversarial Networks?](https://realpython.com/generative-adversarial-networks/#what-are-generative-adversarial-networks)
* [Discriminative vs Generative Models](https://realpython.com/generative-adversarial-networks/#discriminative-vs-generative-models)
* [The Architecture of Generative Adversarial Networks](https://realpython.com/generative-adversarial-networks/#the-architecture-of-generative-adversarial-networks)
* [Your First GAN](https://realpython.com/generative-adversarial-networks/#your-first-gan)
  + [Preparing the Training Data](https://realpython.com/generative-adversarial-networks/#preparing-the-training-data)
  + [Implementing the Discriminator](https://realpython.com/generative-adversarial-networks/#implementing-the-discriminator)
  + [Implementing the Generator](https://realpython.com/generative-adversarial-networks/#implementing-the-generator)
  + [Training the Models](https://realpython.com/generative-adversarial-networks/#training-the-models)
  + [Checking the Samples Generated by the GAN](https://realpython.com/generative-adversarial-networks/#checking-the-samples-generated-by-the-gan)
* [Handwritten Digits Generator With a GAN](https://realpython.com/generative-adversarial-networks/#handwritten-digits-generator-with-a-gan)
  + [Preparing the Training Data](https://realpython.com/generative-adversarial-networks/#preparing-the-training-data_1)
  + [Implementing the Discriminator and the Generator](https://realpython.com/generative-adversarial-networks/#implementing-the-discriminator-and-the-generator)
  + [Training the Models](https://realpython.com/generative-adversarial-networks/#training-the-models_1)
  + [Checking the Samples Generated by the GAN](https://realpython.com/generative-adversarial-networks/#checking-the-samples-generated-by-the-gan_1)
* [Conclusion](https://realpython.com/generative-adversarial-networks/#conclusion)
* [Further Reading](https://realpython.com/generative-adversarial-networks/#further-reading)



[Remove ads](https://realpython.com/account/join/)

Generative adversarial networks (GANs) are [neural networks](https://en.wikipedia.org/wiki/Artificial_neural_network) that generate material, such as images, music, speech, or text, that is similar to what humans produce.

GANs have been an active topic of research in recent years. Facebook’s AI research director Yann LeCun called adversarial training [“the most interesting idea in the last 10 years”](https://www.quora.com/What-are-some-recent-and-potentially-upcoming-breakthroughs-in-deep-learning) in the field of machine learning. Below, you’ll learn how GANs work before implementing two generative models of your own.

In this tutorial, you’ll learn:

* What a generative model is and how it differs from a discriminative model
* How GANs are structured and trained
* How to build your own GAN using [PyTorch](https://pytorch.org/)
* How to train your GAN for practical applications using a GPU and PyTorch

Let’s get started!

Free Bonus: [**5 Thoughts On Python Mastery**](https://realpython.com/bonus/python-mastery-course/), a free course for Python developers that shows you the roadmap and the mindset you’ll need to take your Python skills to the next level.

## **What Are Generative Adversarial Networks?**

Generative adversarial networks are machine learning systems that can learn to mimic a given distribution of data. They were first proposed in a 2014 [NeurIPS paper](https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf) by deep learning expert Ian Goodfellow and his colleagues.

GANs consist of two neural networks, one trained to generate data and the other trained to distinguish fake data from real data (hence the “adversarial” nature of the model). Although the idea of a structure to generate data isn’t new, when it comes to image and video generation, GANs have provided impressive results such as:

* Style transfer using [CycleGAN](https://github.com/junyanz/CycleGAN/), which can perform a number of convincing style transformations on images
* Generation of human faces with [StyleGAN](https://en.wikipedia.org/wiki/StyleGAN), as demonstrated on the website [This Person Does Not Exist](https://www.thispersondoesnotexist.com/)

Structures that generate data, including GANs, are considered generative models in contrast to the more widely studied discriminative models. Before diving into GANs, you’ll look at the differences between these two kinds of models.



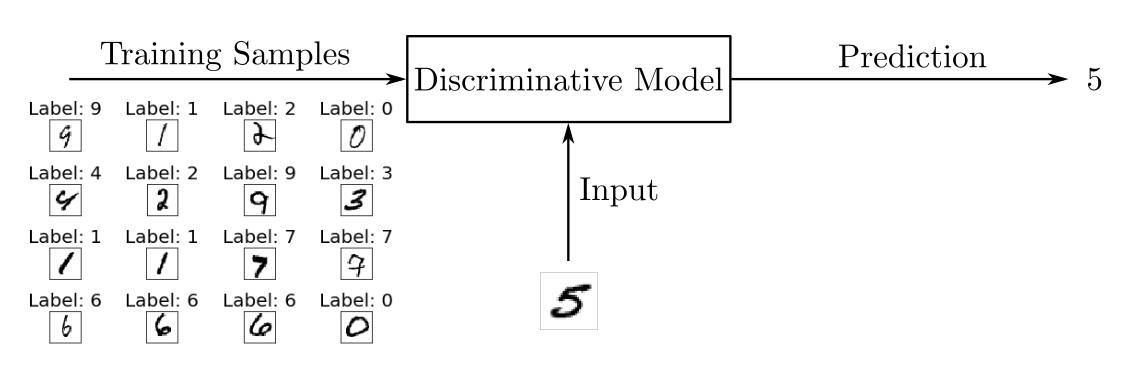
[Remove ads](https://realpython.com/account/join/)

## **Discriminative vs Generative Models**

If you’ve studied neural networks, then most of the applications you’ve come across were likely implemented using [discriminative models](https://en.wikipedia.org/wiki/Discriminative_model). Generative adversarial networks, on the other hand, are part of a different class of models known as [generative models](https://en.wikipedia.org/wiki/Generative_model).

Discriminative models are those used for most [supervised](https://en.wikipedia.org/wiki/Supervised_learning) classification or regression problems. As an example of a classification problem, suppose you’d like to train a model to classify images of handwritten digits from 0 to 9. For that, you could use a labeled dataset containing images of handwritten digits and their associated labels indicating which digit each image represents.

During the training process, you’d use an algorithm to adjust the model’s parameters. The goal would be to minimize a [loss function](https://en.wikipedia.org/wiki/Loss_function) so that the model learns the probability distribution of the output given the input. After the training phase, you could use the model to classify a new handwritten digit image by estimating the most probable digit the input corresponds to, as illustrated in the figure below:

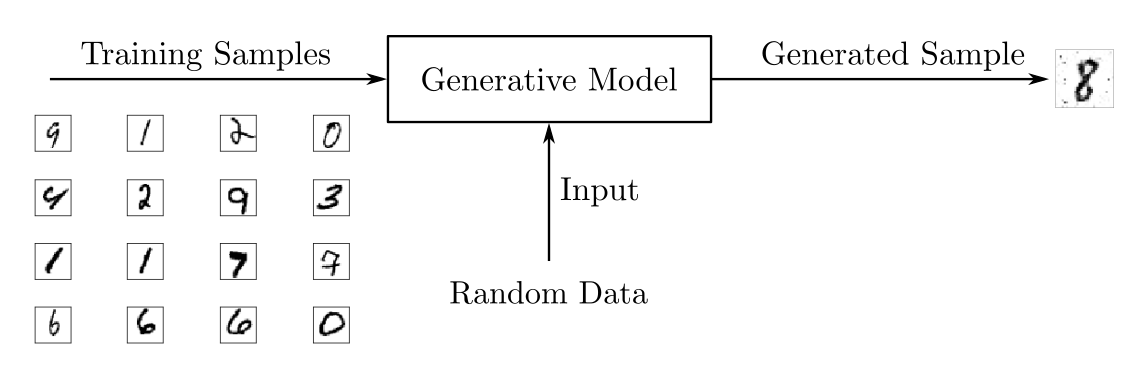


You can picture discriminative models for classification problems as blocks that use the training data to learn the boundaries between classes. They then use these boundaries to discriminate an input and predict its class. In mathematical terms, discriminative models learn the conditional probability *P*(*y*|*x*) of the output *y* given the input *x*.

Besides neural networks, other structures can be used as discriminative models such as [logistic regression](https://realpython.com/logistic-regression-python/) models and [support vector machines](https://en.wikipedia.org/wiki/Support_vector_machine) (SVMs).

Generative models like GANs, however, are trained to describe how a dataset is generated in terms of a probabilistic model. By sampling from a generative model, you’re able to generate new data. While discriminative models are used for supervised learning, generative models are often used with unlabeled datasets and can be seen as a form of [unsupervised learning](https://en.wikipedia.org/wiki/Unsupervised_learning).

Using the dataset of handwritten digits, you could train a generative model to generate new digits. During the training phase, you’d use some algorithm to adjust the model’s parameters to minimize a loss function and learn the probability distribution of the training set. Then, with the model trained, you could generate new samples, as illustrated in the following figure:



To output new samples, generative models usually consider a stochastic, or random, element that influences the samples generated by the model. The random samples used to drive the generator are obtained from a latent space in which the vectors represent a kind of compressed form of the generated samples.

Unlike discriminative models, generative models learn the probability *P*(*x*) of the input data *x*, and by having the distribution of the input data, they’re able to generate new data instances.

Note: Generative models can also be used with labeled datasets. When they are, they’re trained to learn the probability *P*(*x*|*y*) of the input *x* given the output *y*. They can also be used for classification tasks, but in general, discriminative models perform better when it comes to classification.

You can find more information on the relative strengths and weaknesses of discriminative and generative classifiers in the article [On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes](https://papers.nips.cc/paper/2020-on-discriminative-vs-generative-classifiers-a-comparison-of-logistic-regression-and-naive-bayes).

Although GANs have received a lot of attention in recent years, they’re not the only architecture that can be used as a generative model. Besides GANs, there are various other generative model architectures such as:

* [Boltzmann machines](https://en.wikipedia.org/wiki/Boltzmann_machine)
* [Variational autoencoders](https://en.wikipedia.org/wiki/Autoencoder)
* [Hidden Markov models](https://en.wikipedia.org/wiki/Hidden_Markov_model)
* Models that predict the next word in a sequence, like [GPT-2](https://en.wikipedia.org/wiki/OpenAI#GPT-2)

However, GANs have attracted the most public interest of late due to the [exciting results](https://www.christies.com/features/A-collaboration-between-two-artists-one-human-one-a-machine-9332-1.aspx) in image and video generation.

Now that you know the basics of generative models, you’ll see how GANs work and how to train them.

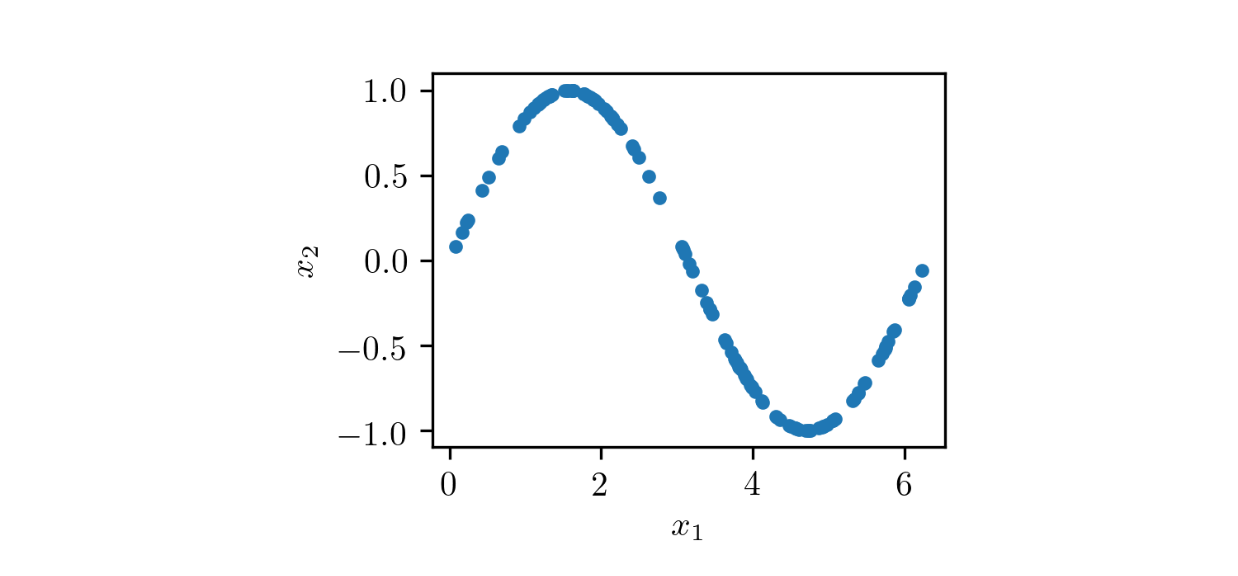
## **The Architecture of Generative Adversarial Networks**

Generative adversarial networks consist of an overall structure composed of two neural networks, one called the generator and the other called the discriminator.

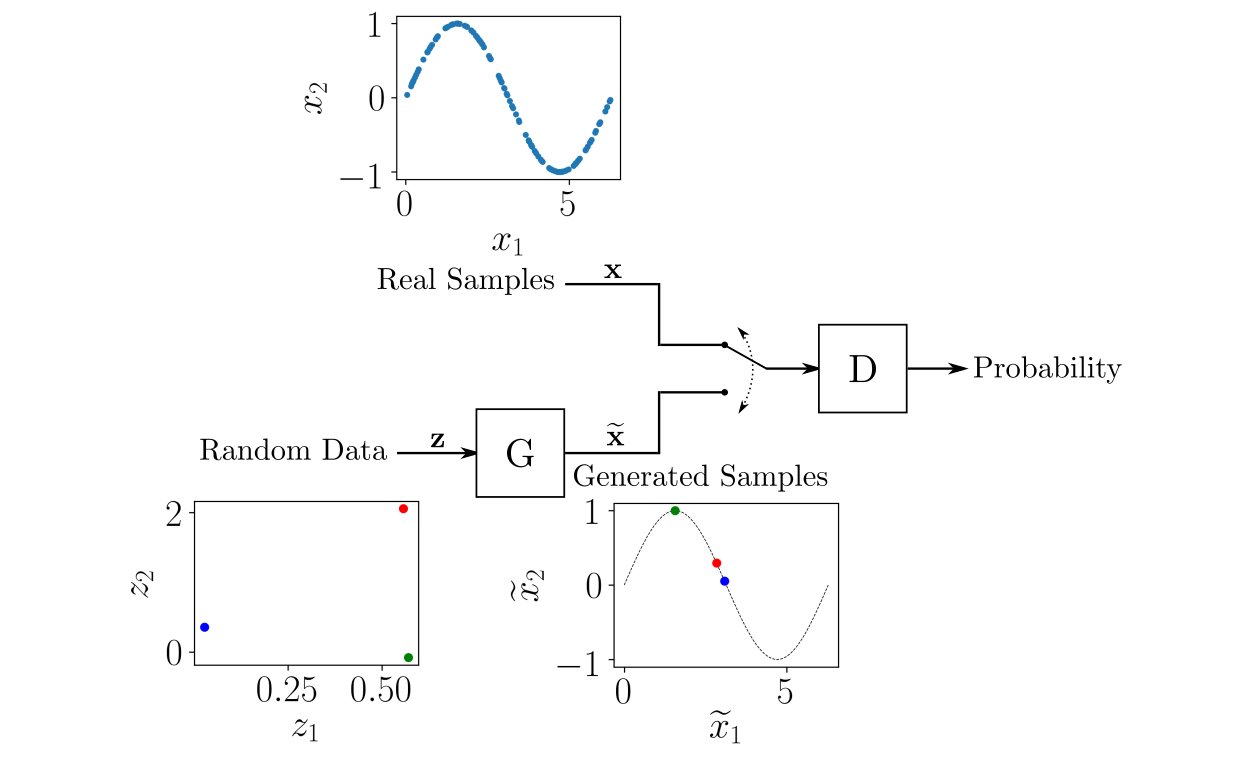
The role of the generator is to estimate the probability distribution of the real samples in order to provide generated samples resembling real data. The discriminator, in turn, is trained to estimate the probability that a given sample came from the real data rather than being provided by the generator.

These structures are called generative adversarial networks because the generator and discriminator are trained to compete with each other: the generator tries to get better at fooling the discriminator, while the discriminator tries to get better at identifying generated samples.

To understand how GAN training works, consider a toy example with a dataset composed of two-dimensional samples (*x*₁, *x*₂), with *x*₁ in the interval from 0 to 2π and *x*₂ = sin(*x*₁), as illustrated in the following figure:



As you can see, this dataset consists of points (*x*₁, *x*₂) located over a sine curve, having a very particular distribution. The overall structure of a GAN to generate pairs (*x̃*₁, *x̃*₂) resembling the samples of the dataset is shown in the following figure:



The generator *G* is fed with [random data](https://realpython.com/python-random/) from a latent space, and its role is to generate data resembling the real samples. In this example, you have a two-dimensional latent space, so that the generator is fed with random (*z*₁, *z*₂) pairs and is required to transform them so that they resemble the real samples.

The structure of the neural network *G* can be arbitrary, allowing you to use neural networks as a [multilayer perceptron](https://en.wikipedia.org/wiki/Multilayer_perceptron) (MLP), a [convolutional neural network](https://en.wikipedia.org/wiki/Convolutional_neural_network) (CNN), or any other structure as long as the dimensions of the input and output match the dimensions of the latent space and the real data.

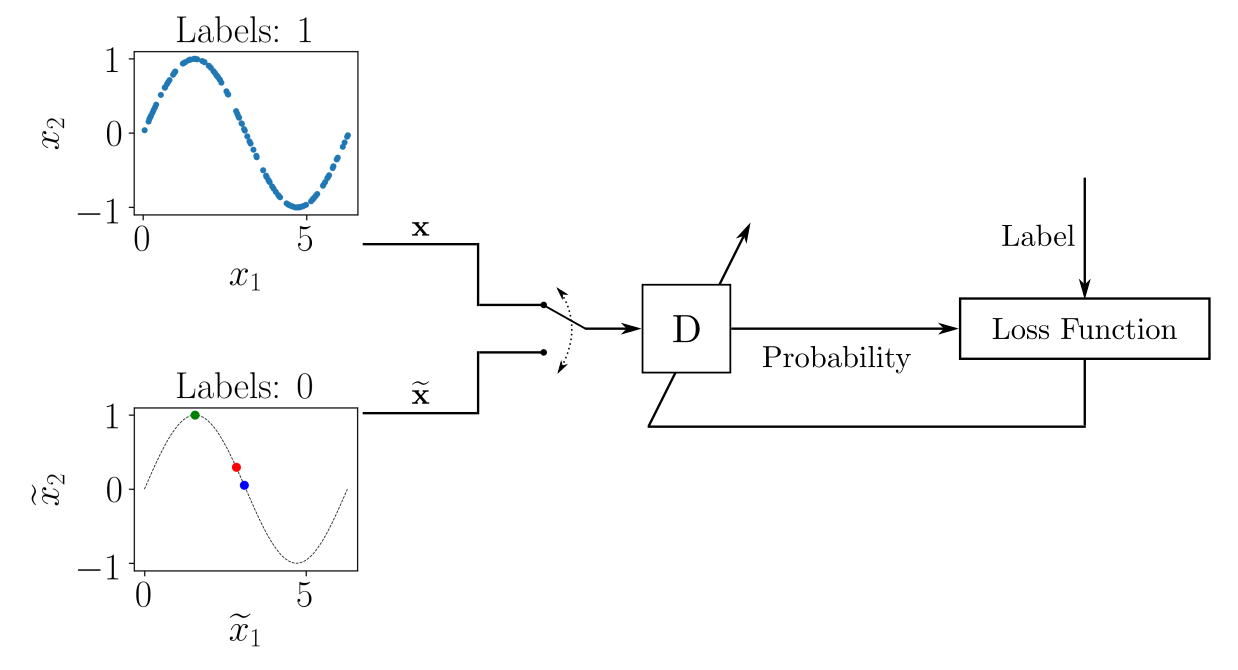
The discriminator *D* is fed with either real samples from the training dataset or generated samples provided by *G*. Its role is to estimate the probability that the input belongs to the real dataset. The training is performed so that *D* outputs 1 when it’s fed a real sample and 0 when it’s fed a generated sample.

As with *G*, you can choose an arbitrary neural network structure for *D* as long as it respects the necessary input and output dimensions. In this example, the input is two-dimensional. For a binary discriminator, the output may be a [scalar](https://en.wikipedia.org/wiki/Scalar_(mathematics)) ranging from 0 to 1.

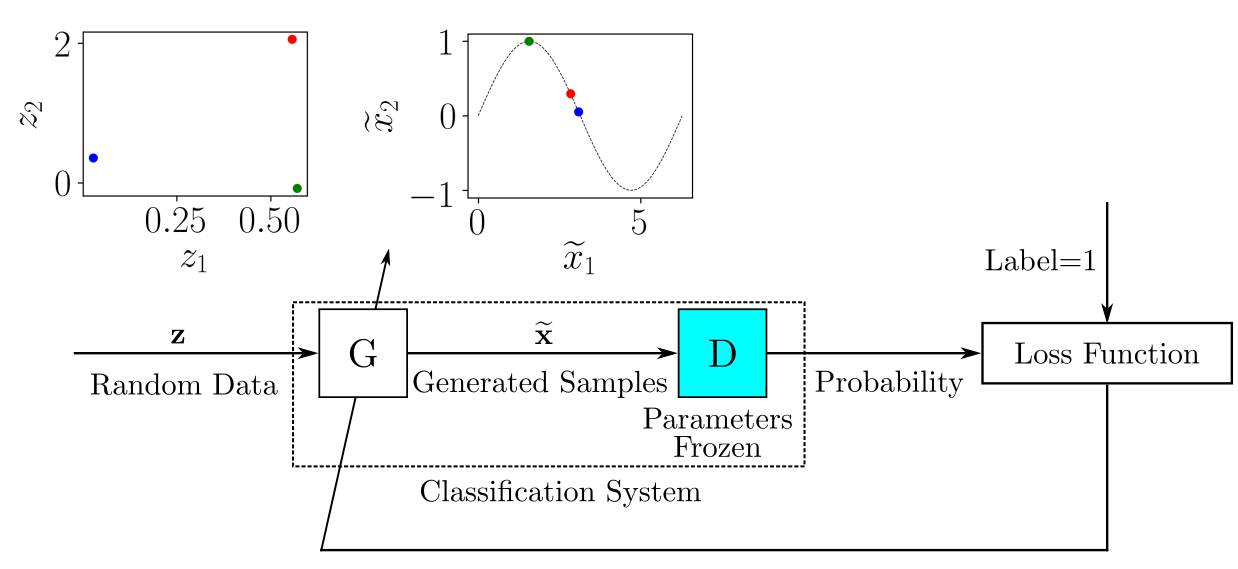
The GAN training process consists of a two-player [minimax](https://en.wikipedia.org/wiki/Minimax) game in which *D* is adapted to minimize the discrimination error between real and generated samples, and *G* is adapted to maximize the probability of *D* making a mistake.

Although the dataset containing the real data isn’t labeled, the training processes for *D* and *G* are performed in a supervised way. At each step in the training, *D* and *G* have their parameters updated. In fact, in the [original GAN proposal](https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf), the parameters of *D* are updated *k* times, while the parameters of *G* are updated only once for each training step. However, to make the training simpler, you can consider *k* equal to 1.

To train *D*, at each iteration you label some real samples taken from the training data as 1 and some generated samples provided by *G* as 0. This way, you can use a conventional supervised training framework to update the parameters of *D* in order to minimize a loss function, as shown in the following scheme:



For each batch of training data containing labeled real and generated samples, you update the parameters of *D* to minimize a loss function. After the parameters of *D* are updated, you train *G* to produce better generated samples. The output of *G* is connected to *D*, whose parameters are kept frozen, as depicted here:



You can imagine the system composed of *G* and *D* as a single classification system that receives random samples as input and outputs the classification, which in this case can be interpreted as a probability.

When *G* does a good enough job to fool *D*, the output probability should be close to 1. You could also use a conventional supervised training framework here: the dataset to train the classification system composed of *G* and *D* would be provided by random input samples, and the label associated with each input sample would be 1.

During training, as the parameters of *D* and *G* are updated, it’s expected that the generated samples given by *G* will more closely resemble the real data, and *D* will have more trouble distinguishing between real and generated data.

Now that you know how GANs work, you’re ready to implement your own using PyTorch.



[Remove ads](https://realpython.com/account/join/)

## **Your First GAN**

As a first experiment with generative adversarial networks, you’ll implement the example described in the previous section.

To run the example, you’re going to use the [PyTorch](https://pytorch.org/) library, which you can install using the [Anaconda](https://www.anaconda.com/products/individual) Python distribution and the [conda](https://docs.conda.io/projects/conda/en/latest/user-guide/getting-started.html) package and environment management system. To learn more about Anaconda and conda, check out the tutorial on [Setting Up Python for Machine Learning on Windows](https://realpython.com/python-windows-machine-learning-setup/).

To begin, create a conda environment and activate it:

$ conda create --name gan

$ conda activate gan

After you activate the conda environment, your prompt will show its name, gan. Then you can install the necessary packages inside the environment:

$ conda install -c pytorch pytorch=1.4.0

$ conda install matplotlib jupyter

Since [PyTorch](https://realpython.com/pytorch-vs-tensorflow/) is a very actively developed framework, the API may change on new releases. To ensure the example code will run, you install the specific version 1.4.0.

Besides PyTorch, you’re going to use [Matplotlib](https://matplotlib.org/) to work with plots and a [Jupyter Notebook](https://jupyter.org/) to run the code in an interactive environment. Doing so isn’t mandatory, but it facilitates working on machine learning projects.

For a refresher on working with Matplotlib and Jupyter Notebooks, take a look at [Python Plotting With Matplotlib (Guide)](https://realpython.com/python-matplotlib-guide/) and [Jupyter Notebook: An Introduction](https://realpython.com/jupyter-notebook-introduction/).

Before opening Jupyter Notebook, you need to register the conda gan environment so that you can create Notebooks using it as the kernel. To do that, with the gan environment activated, run the following command:

$ python -m ipykernel install --user --name gan

Now you can open Jupyter Notebook by running jupyter notebook. Create a new Notebook by clicking *New* and then selecting *gan*.

Inside the Notebook, begin by importing the necessary libraries:

import torch

from torch import nn

import math

import matplotlib.pyplot as plt

Here, you import the PyTorch library with torch. You also import nn just to be able to set up the neural networks in a less verbose way. Then you import math to obtain the value of the pi constant, and you import the Matplotlib plotting tools as plt as usual.

It’s a good practice to set up a random generator seed so that the experiment can be replicated identically on any machine. To do that in PyTorch, run the following code:

torch.manual\_seed(111)

The number 111 represents the [random seed](https://en.wikipedia.org/wiki/Random_seed) used to initialize the random number generator, which is used to initialize the neural network’s [weights](https://deepai.org/machine-learning-glossary-and-terms/weight-artificial-neural-network). Despite the random nature of the experiment, it must provide the same results as long as the same seed is used.

Now that the environment is set, you can prepare the training data.



[Remove ads](https://realpython.com/account/join/)

### Preparing the Training Data

The training data is composed of pairs (*x*₁, *x*₂) so that *x*₂ consists of the value of the sine of *x*₁ for *x*₁ in the interval from 0 to 2π. You can implement it as follows:

1train\_data\_length = 1024

2train\_data = torch.zeros((train\_data\_length, 2))

3train\_data[:, 0] = 2 \* math.pi \* torch.rand(train\_data\_length)

4train\_data[:, 1] = torch.sin(train\_data[:, 0])

5train\_labels = torch.zeros(train\_data\_length)

6train\_set = [

7 (train\_data[i], train\_labels[i]) for i in range(train\_data\_length)

8]

Here, you compose a training set with 1024 pairs (*x*₁, *x*₂). In line 2, you initialize train\_data, a tensor with dimensions of 1024 rows and 2 columns, all containing zeros. A tensor is a multidimensional array similar to a [NumPy array](https://realpython.com/numpy-array-programming/).

In line 3, you use the first column of train\_data to store random values in the interval from 0 to 2π. Then, in line 4, you calculate the second column of the tensor as the sine of the first column.

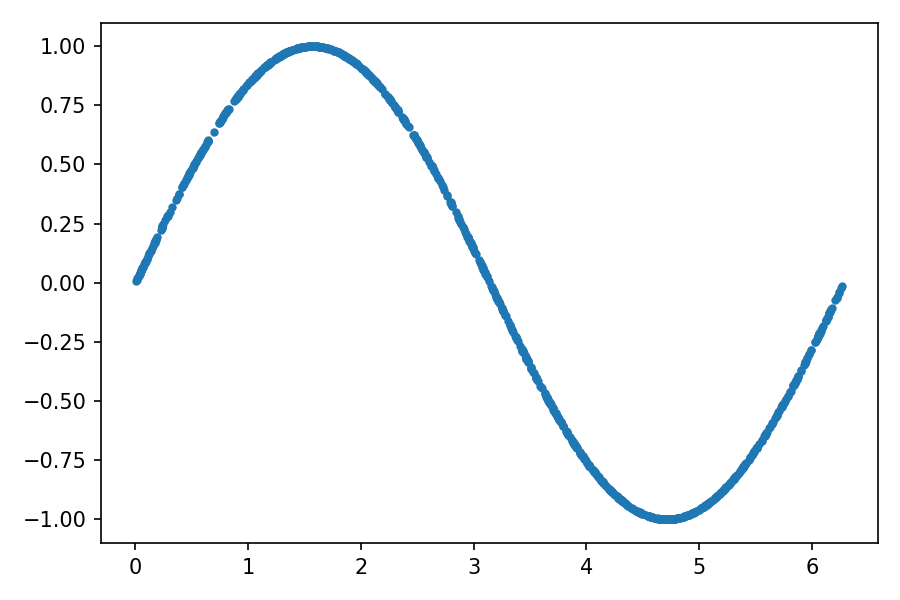
Next, you’ll need a tensor of labels, which are required by PyTorch’s data loader. Since GANs make use of unsupervised learning techniques, the labels can be anything. They won’t be used, after all.

In line 5, you create train\_labels, a tensor filled with zeros. Finally, in lines 6 to 8, you create train\_set as a list of tuples, with each row of train\_data and train\_labels represented in each tuple as expected by PyTorch’s data loader.

You can examine the training data by plotting each point (*x*₁, *x*₂):

plt.plot(train\_data[:, 0], train\_data[:, 1], ".")

The output should be something similar to the following figure:



With train\_set, you can create a PyTorch data loader:

batch\_size = 32

train\_loader = torch.utils.data.DataLoader(

train\_set, batch\_size=batch\_size, shuffle=True

)

Here, you create a data loader called train\_loader, which will shuffle the data from train\_set and return batches of 32 samples that you’ll use to train the neural networks.

After setting up the training data, you need to create the neural networks for the discriminator and generator that will compose the GAN. In the following section, you’ll implement the discriminator.

### Implementing the Discriminator

In PyTorch, the neural network models are represented by classes that inherit from nn.Module, so you’ll have to define a class to create the discriminator. For more information on defining classes, take a look at [Object-Oriented Programming (OOP) in Python 3](https://realpython.com/python3-object-oriented-programming/).

The discriminator is a model with a two-dimensional input and a one-dimensional output. It’ll receive a sample from the real data or from the generator and will provide the probability that the sample belongs to the real training data. The code below shows how to create a discriminator:

1class Discriminator(nn.Module):

2 def \_\_init\_\_(self):

3 super().\_\_init\_\_()

4 self.model = nn.Sequential(

5 nn.Linear(2, 256),

6 nn.ReLU(),

7 nn.Dropout(0.3),

8 nn.Linear(256, 128),

9 nn.ReLU(),

10 nn.Dropout(0.3),

11 nn.Linear(128, 64),

12 nn.ReLU(),

13 nn.Dropout(0.3),

14 nn.Linear(64, 1),

15 nn.Sigmoid(),

16 )

17

18 def forward(self, x):

19 output = self.model(x)

20 return output

You use .\_\_init\_\_() to build the model. First, you need to call super().\_\_init\_\_() to run .\_\_init\_\_() from nn.Module. The discriminator you’re using is an MLP neural network defined in a sequential way using nn.Sequential(). It has the following characteristics:

* Lines 5 and 6: The input is two-dimensional, and the first hidden layer is composed of 256 neurons with [ReLU](https://en.wikipedia.org/wiki/Rectifier_(neural_networks)) activation.
* Lines 8, 9, 11, and 12: The second and third hidden layers are composed of 128 and 64 neurons, respectively, with ReLU activation.
* Lines 14 and 15: The output is composed of a single neuron with [sigmoidal](https://en.wikipedia.org/wiki/Sigmoid_function) activation to represent a probability.
* Lines 7, 10, and 13: After the first, second, and third hidden layers, you use [dropout](https://en.wikipedia.org/wiki/Dropout_(neural_networks)) to avoid [overfitting](https://en.wikipedia.org/wiki/Overfitting).

Finally, you use .forward() to describe how the output of the model is calculated. Here, x represents the input of the model, which is a two-dimensional tensor. In this implementation, the output is obtained by feeding the input x to the model you’ve defined without any other processing.

After declaring the discriminator class, you should instantiate a Discriminator object:

discriminator = Discriminator()

discriminator represents an instance of the neural network you’ve defined and is ready to be trained. However, before you implement the training loop, your GAN also needs a generator. You’ll implement one in the next section.



[Remove ads](https://realpython.com/account/join/)

### Implementing the Generator

In generative adversarial networks, the generator is the model that takes samples from a latent space as its input and generates data resembling the data in the training set. In this case, it’s a model with a two-dimensional input, which will receive random points (*z*₁, *z*₂), and a two-dimensional output that must provide (*x̃*₁, *x̃*₂) points resembling those from the training data.

The implementation is similar to what you did for the discriminator. First, you have to create a Generator class that inherits from nn.Module, defining the neural network architecture, and then you need instantiate a Generator object:

1class Generator(nn.Module):

2 def \_\_init\_\_(self):

3 super().\_\_init\_\_()

4 self.model = nn.Sequential(

5 nn.Linear(2, 16),

6 nn.ReLU(),

7 nn.Linear(16, 32),

8 nn.ReLU(),

9 nn.Linear(32, 2),

10 )

11

12 def forward(self, x):

13 output = self.model(x)

14 return output

15

16generator = Generator()

Here, generator represents the generator neural network. It’s composed of two hidden layers with 16 and 32 neurons, both with ReLU activation, and a linear activation layer with 2 neurons in the output. This way, the output will consist of a vector with two elements that can be any value ranging from negative infinity to infinity, which will represent (*x̃*₁, *x̃*₂).

Now that you’ve defined the models for the discriminator and generator, you’re ready to perform the training!

### Training the Models

Before training the models, you need to set up some parameters to use during training:

1lr = 0.001

2num\_epochs = 300

3loss\_function = nn.BCELoss()

Here you set up the following parameters:

* Line 1 sets the learning rate (lr), which you’ll use to adapt the network weights.
* Line 2 sets the number of epochs (num\_epochs), which defines how many repetitions of training using the whole training set will be performed.
* Line 3 assigns the variable loss\_function to the [binary cross-entropy](https://en.wikipedia.org/wiki/Cross_entropy) function BCELoss(), which is the loss function that you’ll use to train the models.

The binary cross-entropy function is a suitable loss function for training the discriminator because it considers a binary classification task. It’s also suitable for training the generator since it feeds its output to the discriminator, which provides a binary observable output.

PyTorch implements various weight update rules for model training in torch.optim. You’ll use the [Adam algorithm](https://en.wikipedia.org/wiki/Stochastic_gradient_descent#Adam) to train the discriminator and generator models. To create the optimizers using torch.optim, run the following lines:

1optimizer\_discriminator = torch.optim.Adam(discriminator.parameters(), lr=lr)

2optimizer\_generator = torch.optim.Adam(generator.parameters(), lr=lr)

Finally, you need to implement a training loop in which training samples are fed to the models, and their weights are updated to minimize the loss function:

1for epoch in range(num\_epochs):

2 for n, (real\_samples, \_) in enumerate(train\_loader):

3 # Data for training the discriminator

4 real\_samples\_labels = torch.ones((batch\_size, 1))

5 latent\_space\_samples = torch.randn((batch\_size, 2))

6 generated\_samples = generator(latent\_space\_samples)

7 generated\_samples\_labels = torch.zeros((batch\_size, 1))

8 all\_samples = torch.cat((real\_samples, generated\_samples))

9 all\_samples\_labels = torch.cat(

10 (real\_samples\_labels, generated\_samples\_labels)

11 )

12

13 # Training the discriminator

14 discriminator.zero\_grad()

15 output\_discriminator = discriminator(all\_samples)

16 loss\_discriminator = loss\_function(

17 output\_discriminator, all\_samples\_labels)

18 loss\_discriminator.backward()

19 optimizer\_discriminator.step()

20

21 # Data for training the generator

22 latent\_space\_samples = torch.randn((batch\_size, 2))

23

24 # Training the generator

25 generator.zero\_grad()

26 generated\_samples = generator(latent\_space\_samples)

27 output\_discriminator\_generated = discriminator(generated\_samples)

28 loss\_generator = loss\_function(

29 output\_discriminator\_generated, real\_samples\_labels

30 )

31 loss\_generator.backward()

32 optimizer\_generator.step()

33

34 # Show loss

35 if epoch % 10 == 0 and n == batch\_size - 1:

36 print(f"Epoch: {epoch} Loss D.: {loss\_discriminator}")

37 print(f"Epoch: {epoch} Loss G.: {loss\_generator}")

For GANs, you update the parameters of the discriminator and the generator at each training iteration. As is generally done for all neural networks, the training process consists of two loops, one for the training epochs and the other for the batches for each epoch. Inside the inner loop, you begin preparing the data to train the discriminator:

* Line 2: You get the real samples of the current batch from the data loader and assign them to real\_samples. Notice that the first dimension of the tensor has the number of elements equal to batch\_size. This is the standard way of organizing data in PyTorch, with each line of the tensor representing one sample from the batch.
* Line 4: You use torch.ones() to create labels with the value 1 for the real samples, and then you assign the labels to real\_samples\_labels.
* Lines 5 and 6: You create the generated samples by storing random data in latent\_space\_samples, which you then feed to the generator to obtain generated\_samples.
* Line 7: You use torch.zeros() to assign the value 0 to the labels for the generated samples, and then you store the labels in generated\_samples\_labels.
* Lines 8 to 11: You concatenate the real and generated samples and labels and store them in all\_samples and all\_samples\_labels, which you’ll use to train the discriminator.

Next, in lines 14 to 19, you train the discriminator:

* Line 14: In PyTorch, it’s necessary to clear the gradients at each training step to avoid accumulating them. You do this using .zero\_grad().
* Line 15: You calculate the output of the discriminator using the training data in all\_samples.
* Lines 16 and 17: You calculate the loss function using the output from the model in output\_discriminator and the labels in all\_samples\_labels.
* Line 18: You calculate the gradients to update the weights with loss\_discriminator.backward().
* Line 19: You update the discriminator weights by calling optimizer\_discriminator.step().

Next, in line 22, you prepare the data to train the generator. You store random data in latent\_space\_samples, with a number of lines equal to batch\_size. You use two columns since you’re providing two-dimensional data as input to the generator.

You train the generator in lines 25 to 32:

* Line 25: You clear the gradients with .zero\_grad().
* Line 26: You feed the generator with latent\_space\_samples and store its output in generated\_samples.
* Line 27: You feed the generator’s output into the discriminator and store its output in output\_discriminator\_generated, which you’ll use as the output of the whole model.
* Lines 28 to 30: You calculate the loss function using the output of the classification system stored in output\_discriminator\_generated and the labels in real\_samples\_labels, which are all equal to 1.
* Lines 31 and 32: You calculate the gradients and update the generator weights. Remember that when you trained the generator, you kept the discriminator weights frozen since you created optimizer\_generator with its first argument equal to generator.parameters().

Finally, on lines 35 to 37, you display the values of the discriminator and generator loss functions at the end of each ten epochs.

Since the models used in this example have few parameters, the training will be complete in a few minutes. In the following section, you’ll use the trained GAN to generate some samples.



[Remove ads](https://realpython.com/account/join/)

### Checking the Samples Generated by the GAN

Generative adversarial networks are designed to generate data. So, after the training process is finished, you can get some random samples from the latent space and feed them to the generator to obtain some generated samples:

latent\_space\_samples = torch.randn(100, 2)

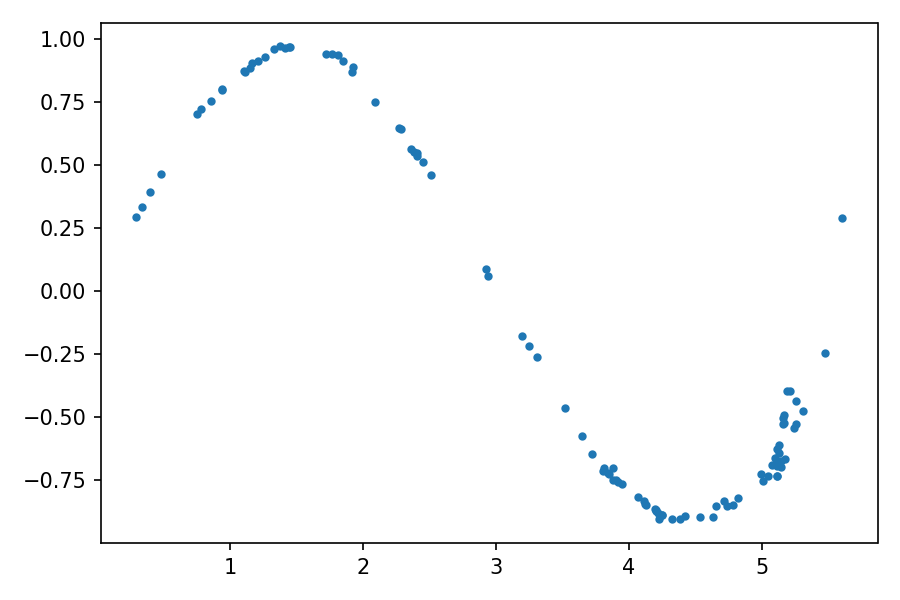
generated\_samples = generator(latent\_space\_samples)

Then you can plot the generated samples and check if they resemble the training data. Before plotting the generated\_samples data, you’ll need to use .detach() to return a tensor from the PyTorch computational graph, which you’ll then use to calculate the gradients:

generated\_samples = generated\_samples.detach()

plt.plot(generated\_samples[:, 0], generated\_samples[:, 1], ".")

The output should be similar to the following figure:



You can see the distribution of the generated data resembles the one from the real data. By using a fixed latent space samples tensor and feeding it to the generator at the end of each epoch during the training process, you can visualize the evolution of the training:

Evolution of the generator

Note that at the beginning of the training process, the generated data distribution is very different from the real data. However, as the training progresses, the generator learns the real data distribution.

Now that you’ve done your first implementation of a generative adversarial network, you’ll go through a more practical application using images.