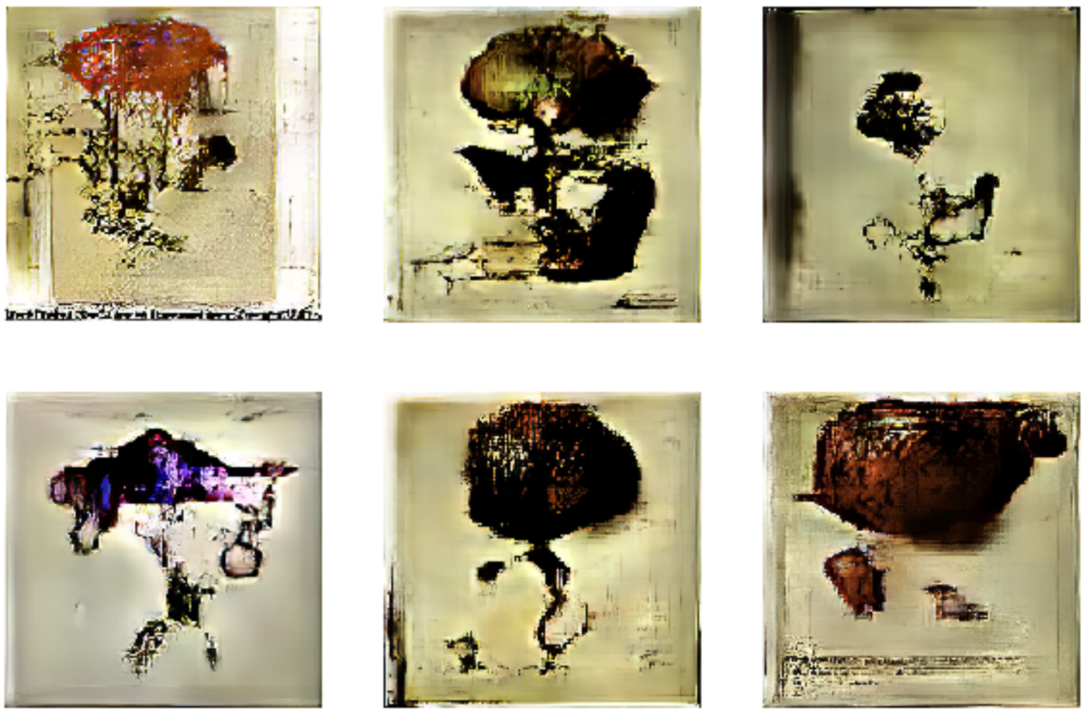
Build a Super Simple GAN in PyTorch

GANs can seem scary but the ideas and basic implementation are super simple, like ~50 lines of code simple.

Nicolas Bertagnolli Mar 9 · 7 min read ★


Botanical drawings from a GAN trained on the USDA pomological watercolor collection.

Introduction

Generative Adversarial Networks (GANs) are a model framework where two models are trained together: one learns to generate synthetic data from the same distribution as the training set and the other learns to distinguish true data from generated data. When I was first learning about them, I remember being kind of overwhelmed with how to construct the joint training. I haven't seen a tutorial yet that focuses on building a trivial GAN so I'm going to try and do that here. No image generation, no fancy deep fried conv nets. We are going to train a model capable of learning to generate even numbers in about 50 lines of Python code. All of the code for this project can be found in the github repository here.

What is a GAN?

GANs are composed of two models trained in unison. The first model, the generator, takes in some random input and tries to output something that looks like our training data. The second model, the discriminator, takes in training data and generated data and tries to distinguish the fake generated data from the real training data. What makes this framework interesting is that these models are trained together. As the discriminator gets better at recognizing fake images this learning is passed to the generator and the generator gets better at generating fake images. To overuse an analogy the generator is to the discriminator as a counterfeiter is to FBI investigators. One tries to forge data, the other tries to distinguish forgeries from the real deal.

This framework has produced a ton of super interesting results in the last few years from translating horses to zebras, to creating deep fakes, to imagining up wholly new images. In this tutorial we aren't going to do anything as interesting as those but this should give you all of the background you need in order to successfully implement a GAN of your own from scratch :). Let's get started.

Problem Definition

Imagine that we have a data set of all even numbers between zero and 128. This is a subset of a much bigger distribution of data, the integers, with some specific properties, much like how human faces are a subset of all images of living things. Our generator is going to take in random noise as an integer in that same range and learn to produce only even numbers.

Before getting into the actual model let's build out our data set. We are going to represent each integer as it's unsigned seven bit binary representation. So the number 56 is 0111000. We do this because:

1. It is very natural to pass in a binary vector to a machine learning algorithm, in this case, a neural network.
2. It is easy to see if the model is generating even numbers by looking at the lowest bit. If it's a one the number is odd, if it's a zero the number is even.

To start let's write a function which converts any positive integer to its binary form as a list.

```python
from typing import List

def create_binary_list_from_int(number: int) -> List[int]:
    if number < 0 or type(number) is not int:
        raise ValueError("Only Positive integers are allowed")

    return [int(x) for x in list(bin(number))[2:]]
```

create_binary_list_from_positive_int.py hosted with ❤ by **GitHub** **view raw**

Convert a positive integer to a binary list

With this we can make a function which will generate random training data for us on the fly.

```python
def generate_even_data(max_int: int, batch_size: int=16) -> Tuple[List[int], List[Li
    # Get the number of binary places needed to represent the maximum number
    max_length = int(math.log(max_int, 2))

    # Sample batch_size number of integers in range 0-max_int
    sampled_integers = np.random.randint(0, int(max_int / 2), batch_size)

    # create a list of labels all ones because all numbers are even
    labels = [1] * batch_size

    # Generate a list of binary numbers for training.
    data = [create_binary_list_from_int(int(x * 2)) for x in sampled_integers]
    data = [([0] * (max_length - len(x))) + x for x in data]

    return labels, data
```

generate_even_data.py hosted with ❤ by **GitHub** **view raw**

Function for generating GAN training data.

This function will produce two outputs the first is a list of ones representing that this data is even and comes from our true distribution. The second output is a random even number in binary list form. That's all we need to start building and training our models!

Building the Generator and Discriminator

Generator

Building the Generator and Discriminator is a snap! Let's start with the Generator. We need something capable of mapping random seven digit binary input to seven digit binary input that is even. The simplest possible thing here is a single seven neuron layer.

```python
class Generator(nn.Module):

    def __init__(self, input_length: int):
        super(Generator, self).__init__()
        self.dense_layer = nn.Linear(int(input_length), int(input_length))
        self.activation = nn.Sigmoid()

    def forward(self, x):
        return self.activation(self.dense_layer(x))
```

even_gan.py hosted with ❤ by **GitHub** **view raw**

Generator Architecture

If we were building a GAN to do something more complicated on say images we would probably train it using random noise generated from a normal distribution and gradually upsample and reshape it until it's the same size as the data we are trying to copy. Since our example is so simple, a single linear layer with a logistic (sigmoid) activation should be enough to map ones and zeros in seven positions to other ones and zeros in seven positions.

Discriminator

The Discriminator is no more complicated than the Generator. Here we need a model to take in a seven digit binary input and output whether or not it is from our real data distribution (is even) or not (is odd or not a number). To accomplish this we use a single neuron model (logistic regression) with a logistic activation (Sigmoid).

```python
class Discriminator(nn.Module):
    def __init__(self, input_length: int):
        super(Discriminator, self).__init__()
        self.dense = nn.Linear(int(input_length), 1);
        self.activation = nn.Sigmoid()

    def forward(self, x):
        return self.activation(self.dense(x))
```

even_discriminator.py hosted with ❤ by **GitHub** **view raw**

That's it, we've built the two models which we will train in unison. Now for the tricky part of GAN training, the training. We need to link these models up in a way that can propagate the gradients around correctly.

Training the Model

Training GANs can seem a bit confusing at first because we need to update two models with every bit of input and we need to be careful about how we do that. So to break it down, we pass two batches of data to our model at every training step. One batch is random noise which will cause the generator to create some generated data, and the second batch is composed solely of data from our true distribution. Throughout the training description, I will reference line numbers in the final training code gist below, not the Github repository.

```python
import math

import torch
import torch.nn as nn


def train(max_int: int = 128, batch_size: int = 16, training_steps: int = 500):
    input_length = int(math.log(max_int, 2))

    # Models
    generator = Generator(input_length)
    discriminator = Discriminator(input_length)

    # Optimizers
    generator_optimizer = torch.optim.Adam(generator.parameters(), lr=0.001)
    discriminator_optimizer = torch.optim.Adam(discriminator.parameters(), lr=0.001)

    # loss
    loss = nn.BCELoss()

    for i in range(training_steps):
        # zero the gradients on each iteration
        generator_optimizer.zero_grad()

        # Create noisy input for generator
        # Need float type instead of int
        noise = torch.randint(0, 2, size=(batch_size, input_length)).float()
        generated_data = generator(noise)

        # Generate examples of even real data
        true_labels, true_data = generate_even_data(max_int, batch_size=batch_size)
        true_labels = torch.tensor(true_labels).float()
        true_data = torch.tensor(true_data).float()

        # Train the generator
        # We invert the labels here and don't train the discriminator because we wan
        # to make things the discriminator classifies as true.
        generator_discriminator_out = discriminator(generated_data)
        generator_loss = loss(generator_discriminator_out, true_labels)
        generator_loss.backward()
        generator_optimizer.step()

        # Train the discriminator on the true/generated data
        discriminator_optimizer.zero_grad()
        true_discriminator_out = discriminator(true_data)
        true_discriminator_loss = loss(true_discriminator_out, true_labels)
```

Final Training Code

Train the Generator

Let's start by training the generator. This consists of:

1. Creating random noise. (Line 27)
2. Generating new "fake" data by passing the noise to the generator (Line 28)
3. Get the predictions from the discriminator on the "fake" data (Line 38)
4. Calculate the loss from the discriminator's output using labels as if the data were "real" instead of fake. (Line 39)
5. Backpropagate the error through just the generator. (Lines 40–41)

Notice how in step four we use true labels instead of fake labels for calculating the loss. This is because we are training the generator. The generator should be trying to fool the discriminator so when the discriminator makes a mistake and says the generated output is real (predicts 1) then the gradients should be small, when the discriminator acts correctly and predicts that the output is generated (predicts 0) the gradients should be big. This is why we only propagate the gradients through the generator at this step, because we inverted the labels. If we trained the entire model like this either the generator would learn the wrong thing or the discriminator would.

Train the Discriminator

Now it's time to update the weights in our discriminator. We do that in a few steps:

1. Pass in a batch of only data from the true data set with a vector of all one labels. (Lines 44–46)
2. Pass our generated data into the discriminator, with detached weights, and zero labels. (Lines 49–50)
3. Average the loss from steps one and two. (Line 51)
4. Backpropagate the gradients through just the discriminator. (Lines 52–53)

The discriminator is trying to learn to distinguish real data from "fake" generated data. The labels while training the discriminator need to represent that, i.e. one when our data comes from the real data set and zero when it is generated by our generator. We pass in those two batches in steps (1) and (2) above and then average the loss from the two batches. It's important to note that when passing in the generated data we want to deta the gradients. We do this because we are not training the generator we are just focused on tl discriminator. Once all of that is done we backpropagate the gradients in only the discriminator and we are done.

Wrapping Up

That's it! We've built our entire GAN. Wrap that in a training loop with some gradient zeroir at each step and we're ready to roll. If we look at the output of our generator at various training steps we can see it converging to only creating even numbers which is exactly what we wanted!

```
0   : [47, 3, 35, 1, 16, 56, 39, 16, 3, 1]
50  : [2, 35, 34, 34, 38, 2, 34, 43, 3, 43]
100 : [42, 43, 106, 38, 35, 42, 35, 42, 43, 106]
200 : [108, 106, 106, 42, 106, 42, 106, 106, 42, 96]
```

At step zero we have 7/10 odd numbers in our sample and at step 200 10/10 of our samples are even numbers! That's a successful generator and it only took ~50 lines of real Python code!

What Next?

As you've probably guessed there are some other tricks for training a GAN which generates non trivial output. Some immediate things to try if you want to make this model work on rea data like images are:

1. The Generator will probably need to be a bit deeper and scale up the noise to the size of the real data. You can do this using transposed convolutions or upsampling layers.
2. Change the noise input to the generator to be Gaussian
3. Increase the depth of the discriminator so that its capacity for prediction is better.
4. Train for much much longer and monitor the loss.

As a good next step try and implement the DCGAN architecture. This code will get you 90% the way there. Once you've done that and made some fun images like those in the introduction, try and improve them by playing around with training hyper parameters. A go list of things to try when training real GANs can be found here.