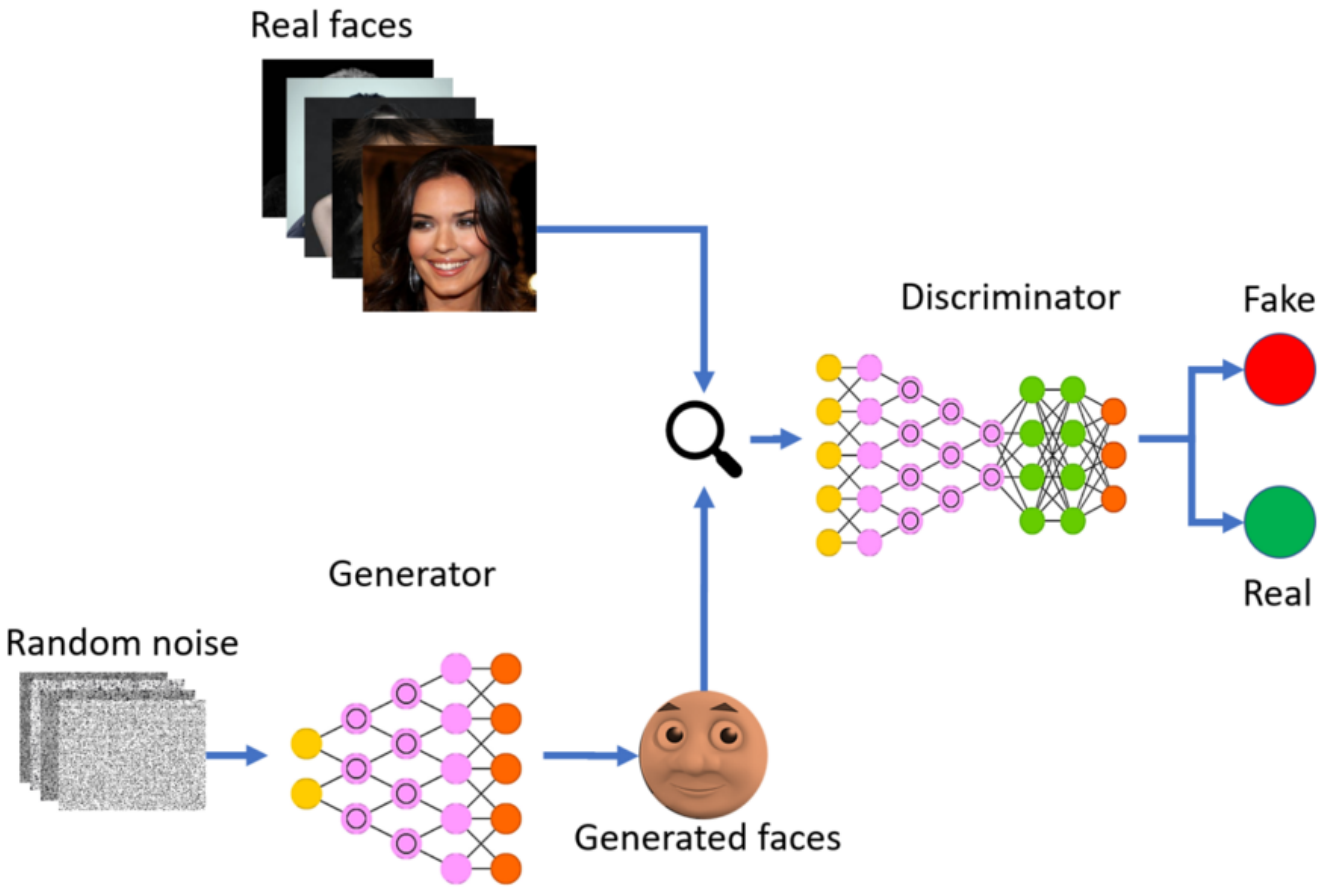
A Brief Introduction To GANs

With explanations of the math and code

Sarvasv Kulpati Follow
Apr 22, 2019 · 5 min read ★



GANs, or Generative Adversarial Networks, are a type of neural network architecture that allow neural networks to generate data. In the past few years, they've become one of the hottest subfields in deep learning, going from generating fuzzy images of digits to photorealistic images of faces.

Before: fuzzy digits, After: photorealistic faces

Variants of GANs have now done insane stuff, like converting images of zebras to horses and vice versa.

I found GANs fascinating, and in an effort to understand them better, I thought that I'd write this article, and in the process of explaining the math and code behind them, understand them better myself.

Here's a link to a github repo I made for GAN resources:

sarvasvkulpati/Awesome-GAN-Resources

Contribute to sarvasvkulpati/Awesome-GAN-Resources development by creating an account on GitHub.

github.com

So how do GANs work?

GANs learn a probability distribution of a dataset by putting two neural networks against each other.

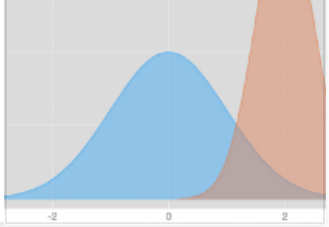
Here's a great article that explains probability distributions and other concepts for those who aren't familiar with them:

Probability concepts explained: probability distributions

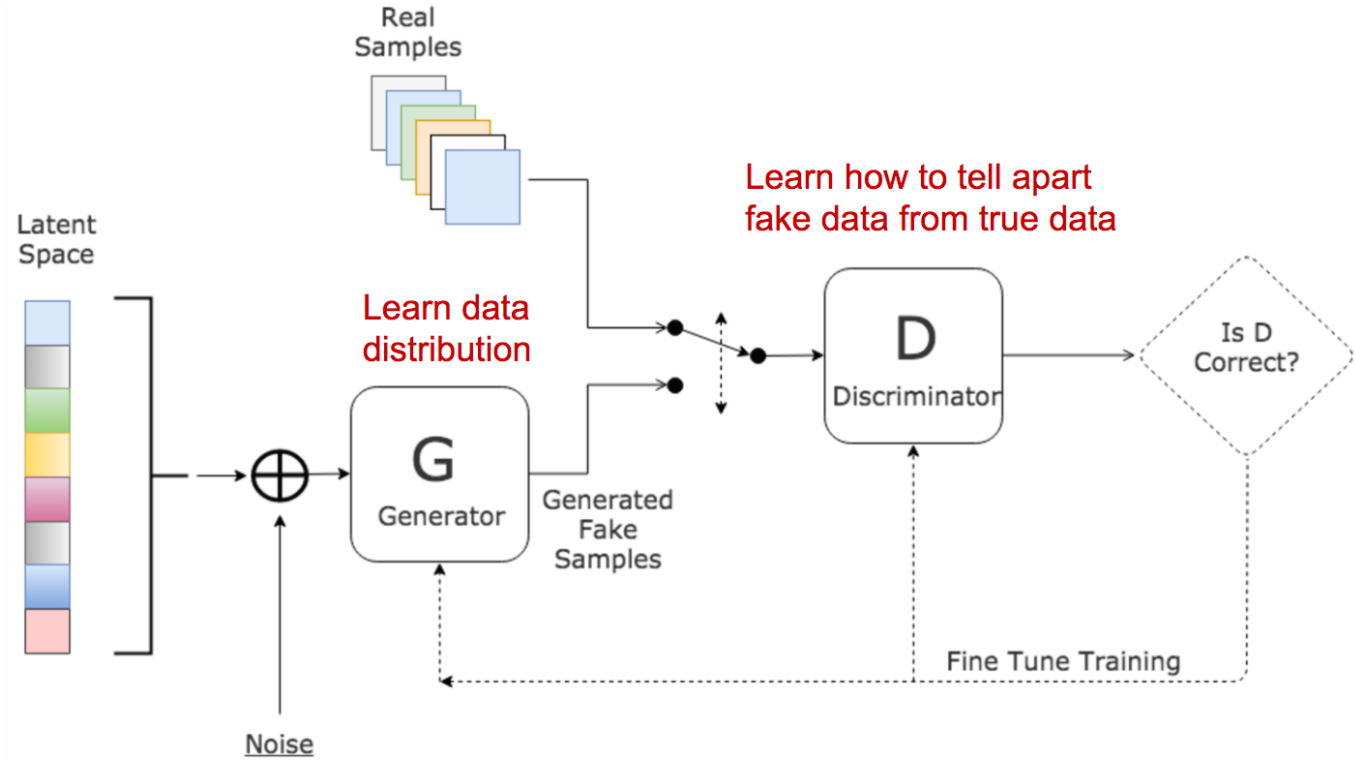
(introduction part 3)

Explaining the fundamentals of probability distributions

towardsdatascience.com



One model, the generator, acts akin to a painting forger. It tries to create images that look very similar to the dataset. The other model, the discriminator, acts like the police, and tries to detect whether the images generated were fake or not.



What basically happens, is that the forger keeps getting better at making fakes, while the police keep getting better at detecting fakes. Effectively, these two models keep trying to beat each other, until after many iterations, the generator creates images indistinguishable from the real dataset.

Training generative adversarial networks involve two objectives:

1. **The discriminator maximizes the probability of assigning the correct label to both training examples and images generated by the generator.** *I.e the policeman becomes better at differentiating between fakes and real paintings.*
2. **The generator minimizes the probability that the discriminator can predict that what it generates is fake.** *I.e the generator becomes better at creating fakes*

Let's try and encode these two ideas into a program.

We'll be following this code in this tutorial

sarvasvkulpati/intro_to_gans

Personal project to understand GANs better. Contribute to sarvasvkulpati/intro_to_gans development by creating an...

github.com

The Data

GANs need a dataset to use, so for this tutorial, we'll be using the classic hello world to machine learning — MNIST, a dataset of handwritten digits.

```
def load_minst_data():

  (x_train, y_train), (x_test, y_test) = mnist.load_data()  #1
  x_train = (x_train.astype(np.float32) - 127.5)/127.5      #2
  x_train = x_train.reshape(60000, 784)                     #3

  return (x_train, y_train, x_test, y_test)
```

load_mnist_data.py hosted with ❤ by GitHub — view raw

The generator also needs random input vectors to generate images, and for this, we'll be using numpy

```
np.random.seed(10)

random_dim = 100
```

initialise_numpy.py hosted with ❤ by GitHub — view raw

The GAN Function

The GAN plays a minimax game, where the entire network attempts to optimize the function V(D,G). This is the equation that defines what a GAN is doing:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

Now to anyone who isn't well versed in the math behind it, it looks terrifying, but the idea it represents is simple, yet powerful. It's just a mathematical representation of the two objectives as defined above.

The generator is defined by G(z), which converts some noise z we input into some data, like images.

The discriminator is defined by D(x), which outputs the probability that the input x came from the real dataset or not.

The discriminator acts like the police

We want the predictions on the dataset by the discriminator to be as close to 1 as possible, and on the generator to be as close to 0 as possible. To achieve this, we use the log-likelihood of D(x) and 1-D(z) in the objective function.

The log just makes sure that the closer it is to an incorrect value, the more it is penalized.

Here's an explanation for log loss if you aren't sure what it does:

Understanding binary cross-entropy / log loss: a visual explanation

Have you ever thought about what exactly does it mean to use this loss function?

towardsdatascience.com

Coding the Generator

```python
def get_generator(optimizer):
    generator = Sequential()
    generator.add(Dense(256, input_dim=random_dim, kernel_initializer=initializers.Randc
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(512))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(1024))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(784, activation='tanh'))
    generator.compile(loss='binary_crossentropy', optimizer=optimizer)
    return generator
```

get_generator.py hosted with ❤ by GitHub view raw

The generator is just a vanilla neural network model that takes a random input vector and outputs a 784-dim vector, which, when reshaped, becomes a 28*28 pixel image.

Coding the Discriminator

```python
def get_discriminator(optimizer):
    discriminator = Sequential()
    discriminator.add(Dense(1024, input_dim=784, kernel_initializer=initializers.RandomN
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))
```

```
    discriminator.add(Dense(512))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(256))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(1, activation='sigmoid'))
    discriminator.compile(loss='binary_crossentropy', optimizer=optimizer)
    return discriminator
```

get_discriminator.py hosted with ❤ by GitHub view raw

The discriminator is another neural network that takes the output of the previous network, a 784-dimensional vector, and outputs a probability between 0 and 1 that it came from the training dataset.

Compiling it into a GAN

```
def get_gan_network(discriminator, random_dim, generator, optimizer):

    discriminator.trainable = False

    gan_input = Input(shape=(random_dim,))
    x = generator(gan_input)
    gan_output = discriminator(x)

    gan = Model(inputs=gan_input, outputs=gan_output)
    gan.compile(loss='binary_crossentropy', optimizer=optimizer)
    return gan
```

get_gan_network.py hosted with ❤ by GitHub view raw

We now compile both models into a single adversarial network, setting the input as a 100-dimensional vector, and the output as the output of the discriminator.

Training the GAN

```
def train(epochs=1, batch_size=128):

    #1
    x_train, y_train, x_test, y_test = load_minst_data()
    batch_count = x_train.shape[0] / batch_size

    #2
```

```
    adam = get_optimizer()
    generator = get_generator(adam)
    discriminator = get_discriminator(adam)
    gan = get_gan_network(discriminator, random_dim, generator, adam)


    #3
    for e in range(1, epochs+1):
        print('-'*15, 'Epoch %d' % e, '-'*15)
        for _ in tqdm(range(int(batch_count))):

            #4
            noise = np.random.normal(0, 1, size=[batch_size, random_dim])
            image_batch = x_train[np.random.randint(0, x_train.shape[0], size=batch_size

            #5
            generated_images = generator.predict(noise)
            X = np.concatenate([image_batch, generated_images])

            #6
            y_dis = np.zeros(2*batch_size)
            y_dis[:batch_size] = 0.9

            #7
            discriminator.trainable = True
            discriminator.train_on_batch(X, y_dis)

            #8
            noise = np.random.normal(0, 1, size=[batch_size, random_dim])
            y_gen = np.ones(batch_size)
            discriminator.trainable = False
            gan.train_on_batch(noise, y_gen)

        if e == 1 or e % 20 == 0:
            plot_generated_images(e, generator)
```

train.py hosted with ❤ by GitHub view raw

1. First, we load the data and split the data into several batches to feed into our model
2. Here we just initialize our GAN network based on the methods defined above
3. This is our training loop, where we run for the specified number of epochs.
4. We generate some random noise and take out some images from our dataset
5. We generate some images using the generator and create a vector X that has some fake images and some real images

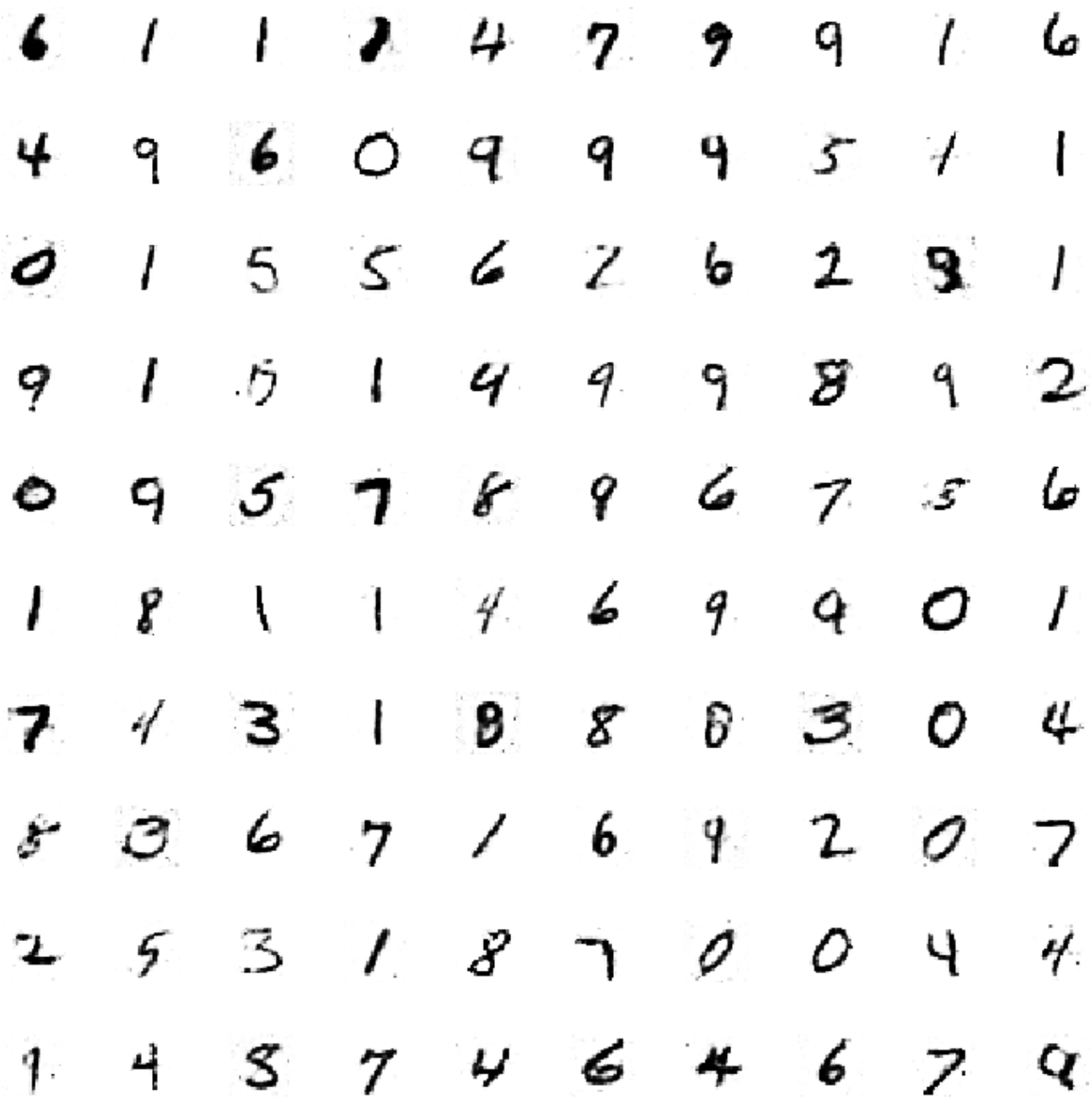
6. We create a vector Y which has the “correct answers” that corresponds to X, with the fake images labeled 0 and the real images labeled 0.9. They’re labeled 0.9 instead of 1 because it helps the GAN train better, a method called one-sided label smoothing.

7. We need to alternate the training between the discriminator and generator, so over here, we update the discriminator

8. Finally, we update the discriminator.

Our first GAN

Once we run the code above, we’ve effectively created our first GAN that generates digits from scratch!



Images generated from the GAN we trained!