High level description: *2

Generative models learn the probability distribution of features in a data set, and therefore calculate an image with the most likely set of features given the known features. (it can therefore be conceptualized as similar to a bayesian network)

Training inputs for the generator are random data, and the objective of the generator is to alter them to fit the distribution.

High level architecture:

The generative NN is trained on random noise within the range of values which it is possible for genuine input to be (known as the latent space), it outputs perturbed versions of its input (which it understands to lie on the distribution of the real data) and its loss function is the inverse of the loss function of the discriminator.

Whereas the classifier NN is trained on batches of images, which are a labeled mixture of real and generated data, and can be updated using back-propagation based on its loss function in the usual way.(ft.2)

Prompts:

Prompt based AI image generators will assign keyword labels to their images instead of just ‘real’ or ‘generated’ (ft.3)

When training in this manner the output for each image in a training batch will be a softmax distribution of the discriminator's confidence of the image corresponding to each keyword (thus the dimensionality of such networks is vast).

And when training the generator random keywords will be assigned to each random input image.

The loss function of the discriminator is now more complicated, it is the inverse of its accuracy in correctly identifying the keywords of real images, plus its accuracy in identifying the keywords of generated images, e.i the discriminator wants to match the keywords for real images and not match the keywords of generated images.

Then when a prompt is entered the trained generator is passed a random image and a non random, yet essentially arbitrary set of keywords.

Implementation for a simple example:

Here we are implementing a GAN which will generate new points along a sin curve, trained on a set of points along said curve.

1) Import the necessary python packages:

```
import torch # this library provides a framework for complex operations of matrices.*3
from torch import nn # this sub-library provides a variety of machine learning functions.*3

import math
import matplotlib.pyplot as plt # this library allows for data visualisation *3
```

2) Set up your training data:

```
train_data_length = 1024
train_data = torch.zeros((train_data_length, 2)) # this line sets up a two dimensional list of initially all zeros.
train_data[:, 0] = 2 * math.pi * torch.rand(train_data_length) # this line generates random x values for the training data (note that torch.rand requires a seed and will be the same seed used when initializing network weights later on)
train_data[:, 1] = torch.sin(train_data[:, 0]) # This line takes the random x values and fits them to corresponding y values so that the points lie along the sin curve.
train_labels = torch.zeros(train_data_length) # This initializes the labels that will identify these instances as training instances when calculating the loss function for the discriminator
train_set = [
    (train_data[i], train_labels[i]) for i in range(train_data_length)
 ]

# the following splits the data into batches using a Dataloader object, which can be created using the nn sub-library of torch.

batch_size = 32
train_loader = torch.utils.data.DataLoader(
    train_set, batch_size=batch_size, shuffle=True
)
```

3) Initialize your neural networks.

```
torch.manual_seed(111)# this provides a random seed for torch.random, it is important that we use a seeded randomiser, so that results can be reproduced.

# discriminator is instantiated as a sequential model with three hidden ReLu layers and then a Sigmoid layer, with dropout on every level, it input dimension is 256,2 to accommodate 2 batches of 32 labeled tuples (one batch from the real data and one batch from the generated data.


# generator is instantiated as a sequential model with three hidden ReLu layers, its input space is 2,16 to accommodate 32 random scalars, its output is 32, 2 to accommodate a single batch of estimated tuples.
```

4) Train your models.

initialize hyperparameters, epochs, learning rate, and type of loss function (binary cross entropy is suggested)

while using pytorch you will need to instantiate optimizer objects for each of your neural networks so that you can backpropagate once the loss function has been calculated.

```
optimizer_discriminator =
torch.optim.Adam(discriminator.parameters(), lr=lr)
 2optimizer_generator = torch.optim.Adam(generator.parameters(),
lr=lr)

# the following must take place in a loop bounded by the epoch hyperparameter
for epoch in range(num_epochs):

#for the discriminators training data take a labelled 32 batch of training data and labeled output for the generator on random input, and concatenate them into a 64 input.
```

```
# then train the discriminator in the standard way, and calculate
loss function based on the confidence of its output.

loss_discriminator = loss_function(
            output_discriminator, all_samples_labels)
        loss_discriminator.backward()
        optimizer_discriminator.step()


# for the generators training data simply generate 32 random values
within the acceptable x range (0,2pi)

# then train the discriminator as you would a classifier but using
the loss function of the discriminator as its loss function.
       output_discriminator_generated =
discriminator(generated_samples)
        loss_generator = loss_function(
            output_discriminator_generated, real_samples_labels
        )
        loss_generator.backward()
        optimizer_generator.step()
```

Generator starts with an input vector Z. This is a vector generated by a simple normal distribution (Normal = bell curve, simple = mean is 0 standard deviation is 1). The result is a random seed vector with weighting to values like a normal distribution. This vector is then turned into the image so as a consequence the generator will generate a large range of images but more normal ones produced more often. This vector doesn't have to be of the same size (but has to be the same shape) as the image and its size is actually a hyper parameter. The random seed Z then passes through the network transforming it to have the same shape as the images required i.e (3, 128, 128) would be an RGB image 128 x 128 pixels.

Looking only at convolutional layered design each “block” of layers is a conv-transpose layer, batch norm 2d and ReLU. The conv-transpose is similar to a CNN but in reverse. In a CNN convolutional layers are used to extract important features by reducing the size (individual pixels don't tell you much but by looking at more of them at once you can tell if there is a curve and if there's curves in the right places it might make the number 3.). In GANs conv-transpose layers increase the size of the image and introduce features learned by the weights. The batch norm 2d layer helps the flow of gradients during training, speeding up convergence. Finally the ReLU layer reduces the link between input and output allowing the generator to capture more complex patterns.

Z (batch_size, size_of_z, 1, 1) passes through 4 blocks with each block multiplying the H, W dimensions. The first block upscales the size of z while each subsequent block divides it by a factor of 2. The final block takes it to 3 as this corresponds to the number of channels in an RGB image.

Footnotes:
(ft.1) GNNs and classifier NNs may take the form of convolutional networks, or almost any other kind of network, as long as the input has the same dimension as the latent space (a compressed form of the former generated samples) and the output has the same dimensionality as the real data.
(ft.2) This architecture is known as a generative adversarial network or “GAN”.

(ft.3) Craiyon for example labels the images in its training dataset according to the google keywords linked to them during image searches.

Training the generator involves optimizing its parameters to minimize the discrepancy between the generated samples and the real data distribution. This optimization process is intricately linked to the dynamics of the GAN training loop, where the generator competes against the discriminator in a minimax game.

During training, the generator strives to produce samples that are indistinguishable from real data, while the discriminator aims to correctly differentiate between real and generated samples. As the generator learns to produce more realistic samples, the discriminator's task becomes increasingly challenging, leading to a dynamic equilibrium between the two networks.

One common challenge in GAN training is mode collapse, where the generator fails to explore the entire data distribution, resulting in limited diversity in the generated samples. Techniques such as minibatch discrimination, spectral normalization, and Wasserstein distance regularization have been proposed to mitigate mode collapse and improve training stability.

Reference:

1) DCGAN Tutorial — PyTorch Tutorials 2.2.0+cu121 documentation
2) Generative Adversarial Networks: Build Your First Models – Real Python
3) PyPI · The Python Package Index