Estudos Sobre GANs

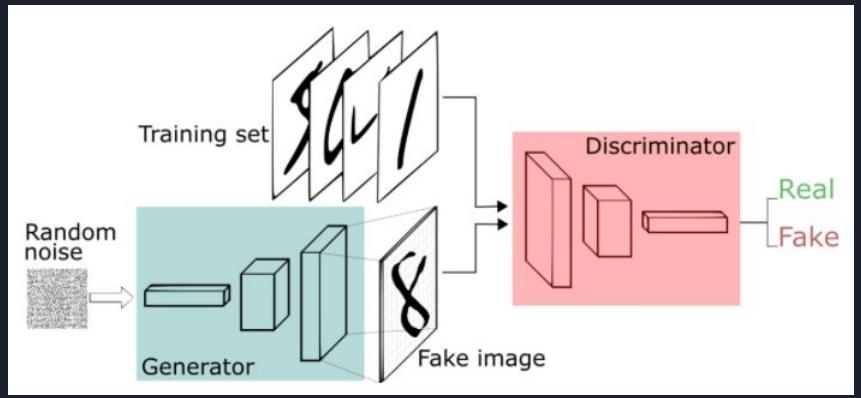
Enrique T. R. Pinto
Thiago Motta

O que é uma rede GAN

O que é uma rede GAN

A GAN (Generative Adversarial Network) é uma classe de sistema de aprendizado de máquinas que surgiu em 2014, inventada por Ian Goodfellow.

Ela é composta por duas redes neurais que concorrem entre si onde uma gera imagens (Generative Network) a partir de características extraídas de um dataset, e a outra tenta classificar as imagens geradas(Discriminative Network). O objetivo da Generative Network é aumentar a taxa de erros da Discriminative Network.

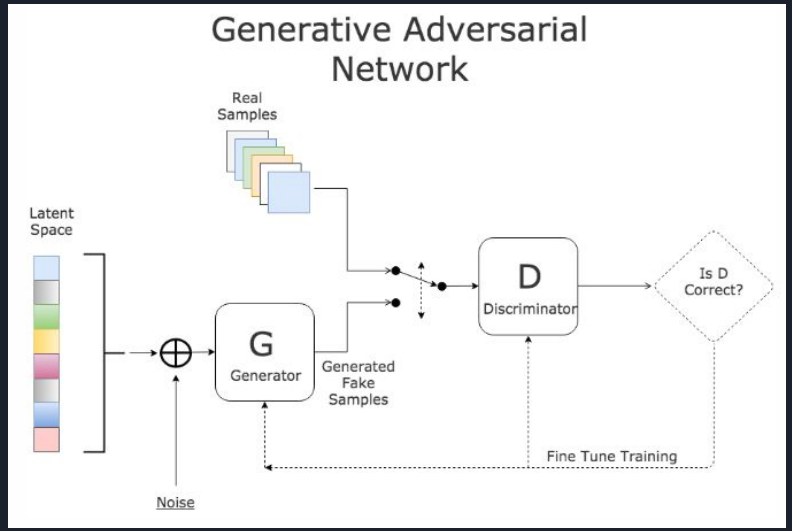


Como a GAN funciona

Um dos conceitos que define a GAN é a ideia de um jogo de soma zero, onde o ganho de uma rede representa necessariamente a perda a rede adversária.

Define-se uma distribuição de ruído Pz(z) de entrada e funções G(z,θg) e D(x,θd), onde o ruído “z” é entrada do gerador, que age como uma transformação do espaço do ruído para o “espaço de dados”, que no caso são imagens de 3 canais (RGB).

O discriminador possui apenas uma saída, que determina a probabilidade do sinal de entrada ter vindo de x em vez de Pg (distribuição de imagens geradas por G).



Como a GAN funciona

A função custo utilizada no jogo minmax é a função representada na figura abaixo.

Vale ressaltar que min max é diferente de max min. Maximizar no discriminador e minimizar no gerador implica em um discriminador com o melhor desempenho atingível e um gerador que exerce seu potencial máximo de simular a distribuição original.

Na prática, o treinamento consiste em treinar o discriminador com um batch de dados reais, com labels '1', um batch de dados gerados, com labels '0'; e então treinar o conjunto "Gerador + Discriminador"(com os parâmetros do discriminador "congelados") com labels '1'.

$$\min_G \max_D V(D,G)$$

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

Problemas Recorrentes

Problemas Recorrentes

Alguns dos dos problemas encontrados, durante o estudo, foram: Instabilidade, Colapso de modo e alta sensibilidade a parâmetros e hiperparâmetros.

A Rede

O Discriminador

Para a implementação do discriminador, utilizamos 4 blocos convolucionais com “Batch Normalization” e “Leaky ReLU” com a finalidade de evitar o “Gradient Vanish Problem”, seguido por uma camada de pooling para reduzir a dimensão.

para a saída, utilizamos uma camada de sigmoid com apenas uma saída, que representa a probabilidade do sinal de entrada ter vindo de x em vez de Pg.

Utilizamos o otimizador “Adam”com o loss “Binary Crossentropy”

```
# define the standalone discriminator model
def define_discriminator(in_shape=(128,128,3)):
  init = RandomNormal(mean=0.0, stddev=0.02)
  model = Sequential()
  # 128x128
  main_input = Input(shape=in_shape)
  model.add(Conv2D(64, (5,5), padding='same', input_shape=in_shape, kernel_initializer=init))
  model.add(BatchNormalization(momentum=0.8))
  model.add(LeakyReLU(alpha=0.2))
  model.add(AveragePooling2D()) # downsample to 64x64

  model.add(Conv2D(128, (5,5), padding='same', kernel_initializer=init))
  model.add(BatchNormalization(momentum=0.8))
  model.add(LeakyReLU(alpha=0.2))
  model.add(AveragePooling2D()) # downsample 32x32

  model.add(Conv2D(256, (5,5), padding='same', kernel_initializer=init))
  model.add(BatchNormalization(momentum=0.8))
  model.add(LeakyReLU(alpha=0.2))
  model.add(AveragePooling2D()) # downsample 16x16

  model.add(Conv2D(512, (5,5), padding='same', kernel_initializer=init))
  model.add(BatchNormalization(momentum=0.8))
  model.add(LeakyReLU(alpha=0.2))
  model.add(AveragePooling2D()) # downsample 8x8
  # dense classifier
  model.add(Flatten())
  model.add(Dense(1024))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dense(512))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dense(128))
  model.add(LeakyReLU(alpha=0.2))
  # output
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0001, beta_1=0.05) # low learning rate
  bce = tf.keras.losses.BinaryCrossentropy(from_logits=False)
  model.compile(loss=bce, optimizer=opt, metrics=['accuracy'])
  return model
```

O Gerador

Para o gerador, utilizamos uma estrutura similar a do discriminador, porém com apenas 3 blocos.

como camada de saída foi utilizado uma camada convolucional com ativação tangente hiperbólica

```
# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  init = RandomNormal(mean=0.0, stddev=0.02)
  # foundation for 16x16 image
  model.add(Dense(64, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dense(512, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dense(16384, input_dim=latent_dim))
  model.add(BatchNormalization(momentum=0.8))
  model.add(Reshape((16, 16, 64)))
  model.add(BatchNormalization(momentum=0.8))
    # upsample
  model.add(Conv2D(256, (5,5), padding='same', kernel_initializer=init))
  model.add(BatchNormalization(momentum=0.8))
  model.add(LeakyReLU(alpha=0.2))
  model.add(BatchNormalization(momentum=0.8))
  model.add(UpSampling2D())
  # upsample
  model.add(Conv2D(128, (5,5), padding='same', kernel_initializer=init))
  model.add(BatchNormalization(momentum=0.8))
  model.add(LeakyReLU(alpha=0.2))
  model.add(UpSampling2D())
  # upsample
  model.add(Conv2D(64, (5,5), padding='same', kernel_initializer=init))
  model.add(BatchNormalization(momentum=0.8))
  model.add(LeakyReLU(alpha=0.2))
  model.add(UpSampling2D())
  # output layer
  model.add(Conv2D(3, (5,5), activation='tanh', padding='same', kernel_initializer=init))
  return model
```