[PSI3472-2023. Aula 11 parte 2. Início]

Introdução à Generative Adversarial Network (GAN)

1 Introdução

Vamos seguir as explicações dos posts [Brownlee2019a, Brownlee2019b] para entender como funciona GAN (Generative Adversarial Network). A arquitetura GAN foi descrita pela primeira vez no artigo "Generative Adversarial Networks" [Goodfellow2014]. Uma lista de aplicações de GAN pode ser encontrada em [https://en.wikipedia.org/wiki/Generative_adversarial_network]. O site [https://www.thispersondoesnotexist.com/] mostra fotos de pessoas e [https://thiscatdoesnotexist.com/] mostra fotos de gatos que foram geradas pelo GAN.

GANs são abordagens para gerar amostras (no nosso caso, imagens) usando métodos de aprendizagem com retro-propagação. GANs enxergam o problema de gerar novas amostras como um problema com dois submodelos: o "gerador" que é uma rede treinada para gerar novas amostras "falsas" a partir de vetores aleatórios e o "discriminador" que é uma rede que classifica se uma amostra é real ou falsa. Os dois modelos são treinados juntos num jogo de soma zero, adversarial, até que o discriminador seja enganado metade das vezes, o que significa que o gerador está gerando amostras plausíveis.

Nota: Em teoria dos jogos e em teoria econômica, um jogo de soma zero se refere a jogos em que o ganho de um jogador representa necessariamente a perda para o outro jogador.

A ideia fica mais clara com alguns exemplos. A figura 1 mostra dígitos "7" gerados artificialmente (a partir de vetores aleatórios) por GAN usando dígitos "7" de MNIST como exemplos de treinamento. A figura 2 mostra pseudo quadros artísticos de flores gerados por GAN. A figura 3 mostra faces artificiais (de pessoas que não existem) geradas por GAN.

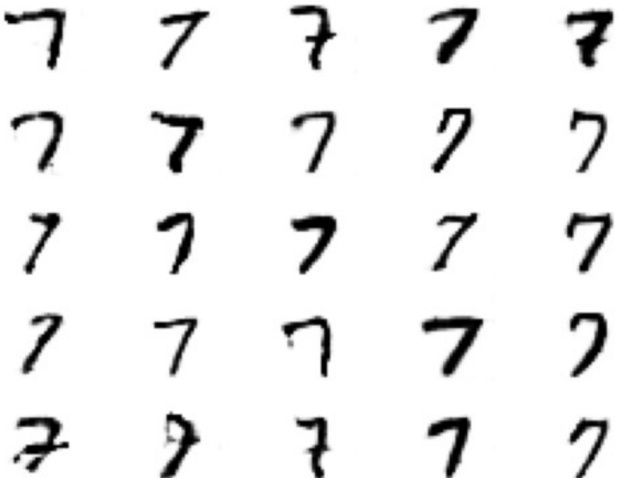


Figura 1: Dígitos "7" gerados artficialmente por uma GAN a partir do MNIST.

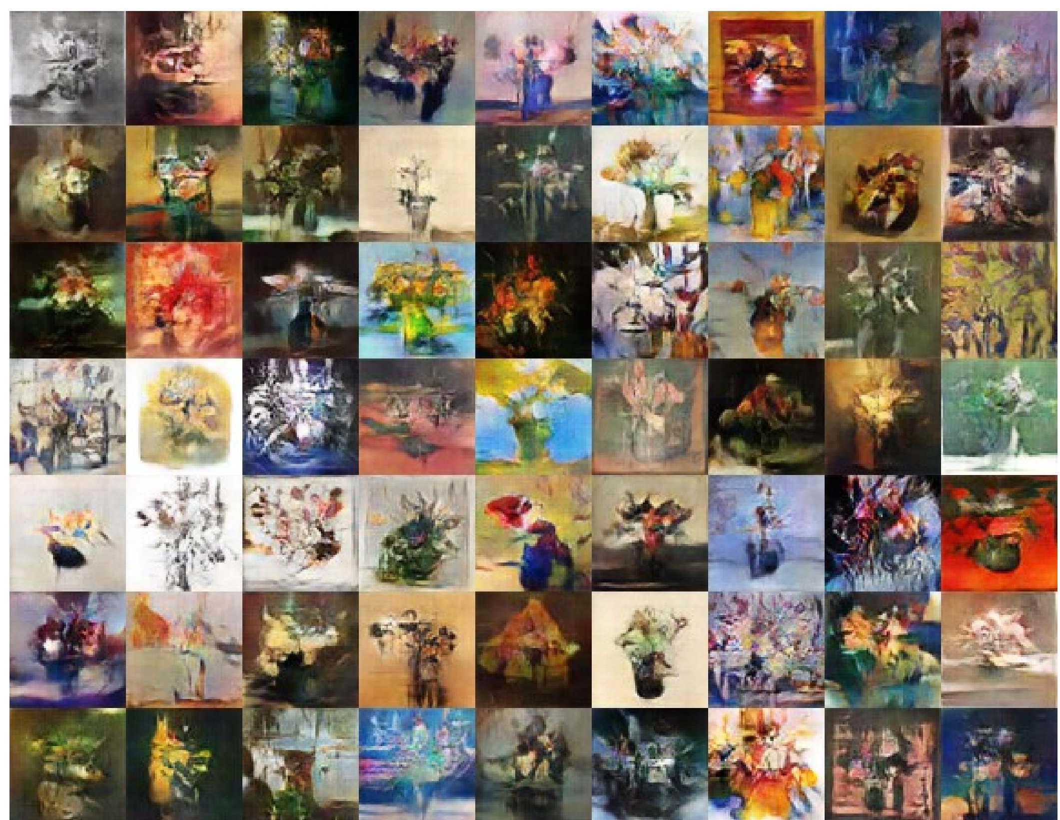


Figura 2: Quadros artísticos de flores gerados artificialmente por uma GAN [https://techcrunch.com/2017/06/20/gangogh/].

Figura 3: Faces artificiais geradas por GAN [https://techxplore.com/news/2018-12-nvidia-face-making-approach-genuinely-gan-tastic.html].

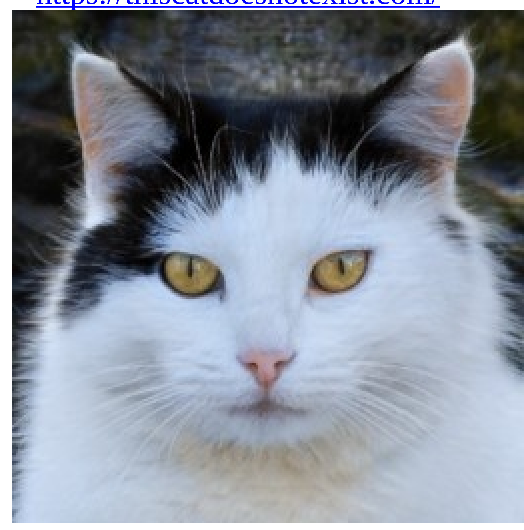
O site:

https://thisxdoesnotexist.com/

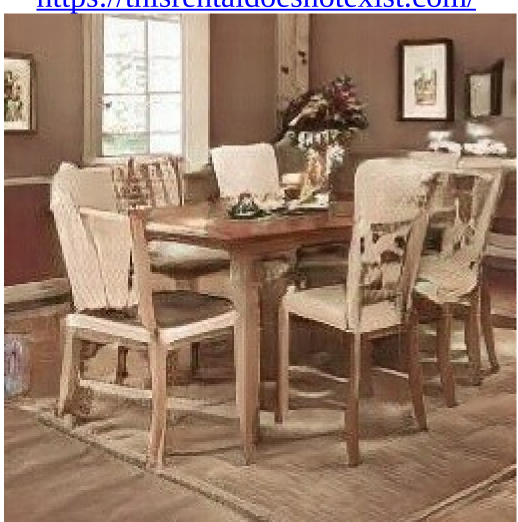
traz uma lista de outros sites que geram versões fake de praticamente qualquer coisa, por exemplo:

https://thispersondoesnotexist.com/

https://thiscatdoesnotexist.com/



https://thisrentaldoesnotexist.com/



https://thishorsedoesnotexist.com/

https://www.thisautomobiledoesnotexist.com/

O modelo gerador

O gerador pega um vetor aleatório de comprimento fixo como entrada e o converte em uma amostra. Esse espaço vetorial é conhecido como espaço latente. Variáveis latentes são aquelas que não são diretamente observáveis. Após o treinamento, o gerador é usado para gerar novas amostras falsas (mas que parecem reais) a partir de outros vetores aleatórios. Modelar imagem significa que o espaço latente, a entrada do gerador, fornece uma representação compacta do conjunto de imagens usadas para treinar o modelo.

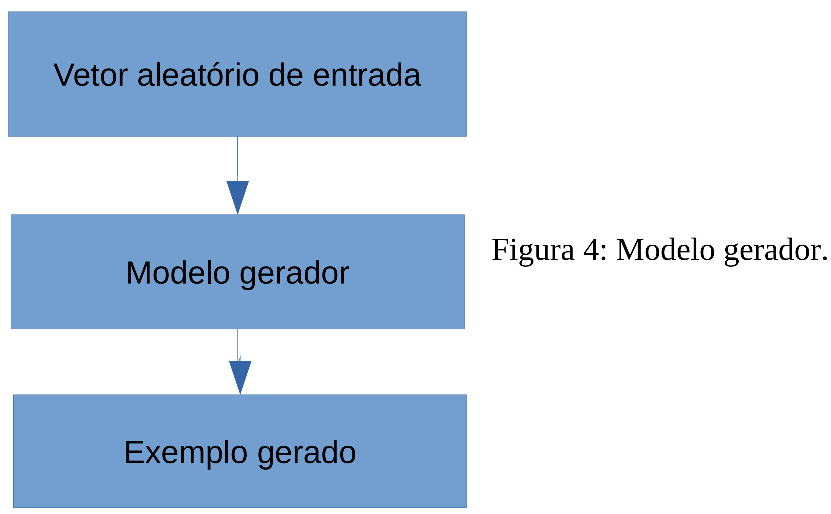


Figura 4: Modelo gerador.

O modelo discriminador

O discriminador pega uma amostra (real ou falsa) do domínio de entrada e tenta classificá-la em real ou falsa. A amostra real vem do conjunto de dados de treinamento. As amostras falsas são produzidas pelo gerador. Após o treinamento, o discriminador é descartado pois estarmos interessados somente no gerador para gerar as amostras artificiais.

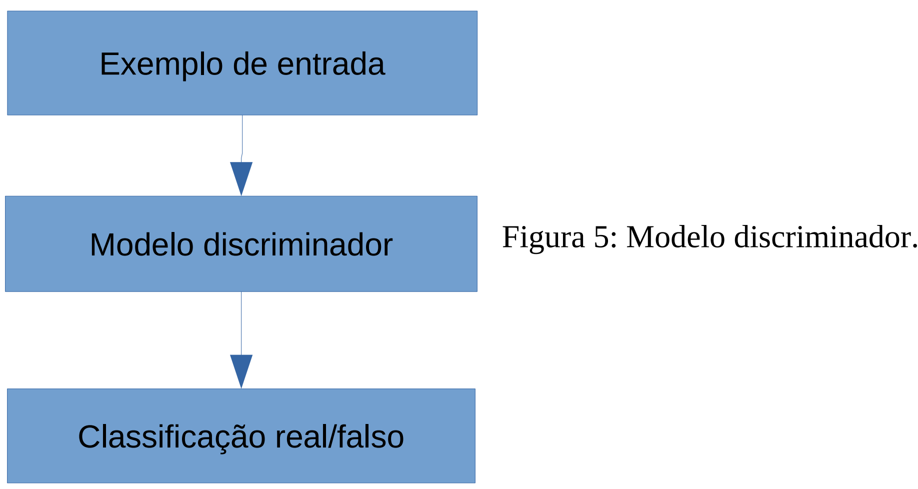


Figura 5: Modelo discriminador.

GAN como um jogo com dois jogadores

Os dois modelos, o gerador e o discriminador, são treinados juntos. Acompanhe a explicação pela figura 6.

Passo 1) O gerador gera um lote de amostras falsas e estas, junto com um lote de amostras reais, são fornecidas ao discriminador que as classifica como reais ou falsas. A atualização dos parâmetros (back-propagation) é feita nos passos (2) e (3) seguintes.

Passo 2) O discriminador é atualizado para melhorar a classificação de amostras reais/falsas, da mesma forma que faria no treino de uma CNN para classificar imagens "normal".

Passo 3) Por sua vez, o gerador é atualizado para gerar amostras que se pareçam cada vez mais reais, de forma que o discriminador pense que as amostras geradas são reais. O gerador está preocupado somente com o desempenho do discriminador nas amostras falsas, pois o gerador não tem controle sobre as amostras reais. Assim, somente as amostras falsas são usadas para atualizar o gerador, marcando-as como reais e apresentando-as ao discriminador para que calcule a função custo. O backpropagation vai atualizar os parâmetros do gerador para minimizar a função custo. Com isso, o gerador aprende a gerar amostras falsas que se parecem um pouco mais com as verdadeiras. Todos os parâmetros do discriminador são congelados antes de atualizar o gerador, para não "estragar" o discriminador. Se não fizesse isso, estaríamos treinando o discriminador para pensar que as amostras falsas são verdadeiras, que não é o que gostaríamos de fazer.

Os três passos acima são executados repetidamente. Podemos pensar no gerador como um falsificador que tenta criar dinheiro falso, e no discriminador como um policial que tenta distinguir cédula verdadeira da falsificada. O que queremos obter são cédulas falsas que se pareçam verdadeiras (por exemplo, um número "7" artificial que se pareça que foi escrito por um ser humano). Para isso, o falsificador é treinado para que consiga criar cédulas falsas indistinguíveis da genuína, com a ajuda do policial aprende continuamente a distinguir as cédulas verdadeiras daquelas geradas pelo falsificador. Dessa forma, os dois modelos estão competindo entre si, são adversários no sentido da teoria dos jogos e estão jogando um jogo de soma zero.

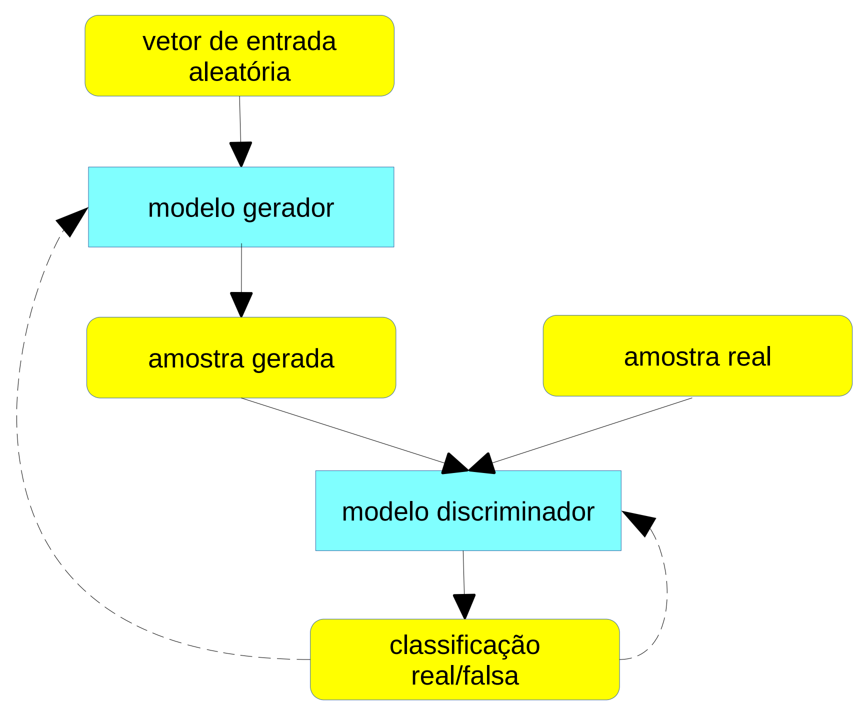


Figura 6: Exemplo de arquitetura GAN.

GAN para gerar dígitos "7"

Brownlee [Brownlee2019b] apresenta um programa que gera os dígitos de MNIST artificiais usando GAN. Vamos adaptar esse programa para entender melhor o seu funcionamento. Vamos gerar apenas o dígito "7", em vez de gerar todos os 10 dígitos do MNIST, como no programa original, o que diminui o tempo de treino. Os trechos alterados mais importantes estão marcados em amarelo no programa 1. Vou tentar explicar como funciona o programa seguindo os passos da execução.

O programa principal começa na linha 121, definindo a dimensão do espaço latente como 10. A dimensão foi reduzida de 100 do programa original [Brownlee2019b] para 10, pois o programa original gera os 10 dígitos enquanto que o nosso programa irá gerar somente o dígito "7".

Na linha 122, o programa define o modelo discriminador, chamando a função *define_discriminator* (linha 16). Esta rede precisa distinguir dígito real do falso. Para isso, podemos usar alguma CNN que já usamos outras vezes para classificação. Alterei o discriminador original [Brownlee2019b] pela rede "tipo LeNet" que já utilizamos para classificar MNIST (apostila *convkeras-ead*). O discriminador só tem 1 neurônio na saída, pois precisa classificar a imagem em dígito "7" verdadeiro ou imitação. Além disso, a ativação final é sigmoide, pois precisamos obter saída entre 0 e 1 para poder calcular a função custo *binary_crossentropy*. Na linha 26, o otimizador Adam está com parâmetros que [Brownlee2019b] afirma serem os mais adequados para GAN.

Na linha 123, o programa define o modelo gerador chamando a função *define_generator* (linha 30). Esta rede recebe um vetor de 10 números aleatórios com distribuição normal, média zero e desvio um (gerado pela função *generate_latent_points* na linha 69) e o converte em um dígito "7" artificial. Para isso, primeiro converte o vetor aleatório em 64 mapas de atributos 7×7 usando camada densa (linhas 32 e 33). Depois, aumenta a resolução desses mapas para 14×14 usando *UpSampling2D*, seguido por convolução 2×2 (linhas 34 e 35). Isto é repetido mais uma vez para obter 8 mapas 28×28 (linhas 36 e 37). Finalmente, uma convolução 1×1 converte os 8 mapas de atributos 28×28 na imagem 28×28 final (linha 38). A ativação final é sigmoide para gerar saídas entre 0 e 1. Aqui também alterei o gerador original de [Brownlee2019b] que usava *Conv2DTranspose* para usar *UpSampling2D*. Sem fazer treino, o gerador irá gerar imagens com pixels aleatórios entre 0 e 1.

Na linha 124, o programa chama a função *define_gan* (linha 41) para criar um modelo artificial composto pelo gerador e discriminador, para poder treinar o gerador. Talvez o nome da função não seja o mais adequado, pois o modelo composto (chamado pelo programa de "gan") na verdade é um modelo lógico criado somente para treinar o gerador. Como vimos, para treinar o gerador, os pesos do discriminador devem ser congelados. Isto é feito na linha 43, fazendo *d_model.trainable = False*. Isto não afeta o treino do discriminador, pois a propriedade *trainable* estava *True* quando compilamos o discriminador. Depois, o programa cria a rede lógica composta por gerador e discriminador nas linhas 44-46. Esta rede é compilada usando otimizador Adam com parâmetros adequados para GAN.

Na linha 125, o programa chama a função *load_real_samples* (linha 54) para carregar MNIST. O programa original carrega todos os 10 dígitos do MNIST. Aqui, estamos carregando somente o dígito "7" (linhas 55-57). Isto acelera o processamento, pois precisamos processar somente 10% dos dados, permitindo fazer testes sem perder muito tempo.

Na linha 126, o programa imprime as amostras reais com nome "generated_plot_e000.png" (usa número 000 para indicar amostras reais).

Na linha 127, o programa chama a função *train* (linha 96) passando como parâmetros o gerador (*g_model*), o discriminador (*d_model*), a rede lógica criada para poder treinar o discriminador

(*gan_model*), o tensor somente com imagens de dígitos “7” (*dataset*) e a dimensão do espaço latente (*latent_dim*=10).

Nas linhas 100-102, dentro da função *train*, há dois laços encaixados. O primeiro (for i ..., linha 100) enumera as épocas. O segundo (for j..., linha 102) enumera os lotes (batches). Em cada lote, são pegos 100 amostras reais, e são gerados 100 amostras falsas usando *g_model* (linhas 103-104). Essas duas amostras são empilhadas (linha 105) para formar um lote de treino.

Na linha 106 é treinado o discriminador.

Na linha 107, o gerador gera um lote de vetores com números aleatórios. Esses vetores serão convertidos em imagens falsas pelo gerador (*g_model*).

A linha 109 rotula as imagens falsas geradas pelo gerador como se fossem reais.

A linha 110 treina a rede artificial *gan_model* usando esse lote. Isto é, irá atualizar os parâmetros do gerador (*g_model*).

As linhas 111-118 imprimem informações sobre o progresso do treino. Também imprimem periodicamente amostras de imagens geradas.

Figura 7 mostra algumas imagens geradas e algumas imagens reais.

```python
# gan_completo5.py
# import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
# os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'

# example of training a gan on mnist
from numpy import expand_dims, zeros, ones, vstack
from numpy.random import randn, randint
from tensorflow.keras.datasets.mnist import load_data
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import Sequential
from keras.layers import Dense, Reshape, Flatten, Conv2D, Conv2DTranspose, \
  LeakyReLU, Dropout, MaxPooling2D, UpSampling2D
from matplotlib import pyplot
import sys

def define_discriminator(in_shape=(28,28,1)):
  model = Sequential() # 28x28
  model.add(Conv2D(20, kernel_size=(5,5), activation='relu',
    input_shape=in_shape )) #20x24x24
  model.add(MaxPooling2D(pool_size=(2,2))) #20x12x12
  model.add(Conv2D(40, kernel_size=(5,5), activation='relu')) #40x8x8
  model.add(MaxPooling2D(pool_size=(2,2))) #40x4x4
  model.add(Flatten()) #640
  model.add(Dense(100, activation='relu')) #100
  model.add(Dense(1, activation='sigmoid')) #1
  opt = Adam(learning_rate=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

def define_generator(latent_dim):
  model = Sequential()
  model.add(Dense(64 * 7 * 7, input_dim=latent_dim,activation="relu"))
  model.add(Reshape((7, 7, 64)))
  model.add(UpSampling2D(size = (2,2))) #14x14
  model.add(Conv2D(64, 2, activation = 'relu', padding = 'same'))
  model.add(UpSampling2D(size = (2,2))) #28x28
  model.add(Conv2D( 8, 2, activation = 'relu', padding = 'same'))
  model.add(Conv2D( 1, 1, activation = 'sigmoid', padding = 'same'))
  return model

def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  model = Sequential()
  model.add(g_model)
  model.add(d_model)
  # Muda os nomes dos layers
  model.layers[0]._name="gerador"
  model.layers[1]._name="discriminador"
  opt = Adam(learning_rate=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model

def load_real_samples():
  (trainX, trainy), (_, _) = load_data()
  trainX = trainX[(trainy==7)]
  del trainy
  X = expand_dims(trainX, axis=-1)
  X = X.astype('float32')
  X = X / 255.0
  return X

def generate_real_samples(dataset, n_samples):
  ix = randint(0, dataset.shape[0], n_samples)
  X = dataset[ix]
  y = ones((n_samples, 1))
  return X, y

def generate_latent_points(latent_dim, n_samples):
  x_input = randn(n_samples, latent_dim)
  return x_input

def generate_fake_samples(g_model, latent_dim, n_samples):
  x_input = generate_latent_points(latent_dim, n_samples)
  X = g_model.predict(x_input)
  y = zeros((n_samples, 1))
  return X, y

def save_plot(examples, epoch, n=5):
  for i in range(n * n):
    pyplot.subplot(n, n, 1 + i)
    pyplot.axis('off')
    pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
  filename = 'generated_plot_e%03d.png' % (epoch+1)
  pyplot.savefig(filename)
  pyplot.close()

def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=100):
  X_real, y_real = generate_real_samples(dataset, n_samples)
  _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  print('Acuracia do discriminador em amostras reais: %.0f%%, falsas: %.0f%%' % (acc_real*100, acc_fake*100))
  save_plot(x_fake, epoch)

def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=100):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs): #i==epoch
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      X_real, y_real = generate_real_samples(dataset, half_batch)
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
```

```
105         X, y = vstack((X_real, X_fake)), vstack((y_real, y_fake))
106         d_loss, d_accuracy = d_model.train_on_batch(X, y)
107         X_gan = generate_latent_points(latent_dim, n_batch)
108         # create inverted labels for the fake samples
109         y_gan = ones((n_batch, 1))
110         g_loss = gan_model.train_on_batch(X_gan, y_gan)
111         if (j+1==bat_per_epo):
112           print('epoch=%d, batch=%d/%d, d_acc=%.3f, g_loss=%.3f (bin_crossentr)'
113             % (i+1, j+1, bat_per_epo, d_accuracy, g_loss))
114       if (i+1) % 10 == 0:
115         summarize_performance(i, g_model, d_model, dataset, latent_dim)
116       if i+1==n_epochs:
117         filename = 'generator_model_%03d.h5' % (i + 1)
118         g_model.save(filename)
119
120 # size of the latent space
121 latent_dim = 10
122 d_model = define_discriminator()
123 g_model = define_generator(latent_dim)
124 gan_model = define_gan(g_model, d_model)
125 dataset = load_real_samples()
126 save_plot(dataset,-1)
127 train(g_model, d_model, gan_model, dataset, latent_dim)
```

Programa 1: GAN para gerar dígitos “7”.

[https://colab.research.google.com/drive/1sfVlHlQXw8N0Ug-88mu3--M9Z5C2mNaQ?usp=sharing]

Parte da saída gerada pelo programa 1 (gan_completo5.py) está abaixo. Um problema em implementar GAN é que não há como medir objetivamente o quanto as imagens geradas são boas ou não. Se o discriminador errar a classificação de uma grande parte das imagens falsas pode ser que o gerador seja bom, mas também pode ser que o discriminador seja ruim.

Assim, o programa 1 não informa quão boa está a saída gerada. Porém, é possível usá-la para verificar quando o treino não está convergindo (neste caso, alguma saída será 0% ou 100%).

```
epoch=10, batch=62/62, d_acc=0.840, g_loss=1.110 (bin_crossentr)
Acuracia do discriminador em amostras reais: 80%, falsas: 78%
epoch=20, batch=62/62, d_acc=0.560, g_loss=0.955 (bin_crossentr)
Acuracia do discriminador em amostras reais: 41%, falsas: 89%
epoch=30, batch=62/62, d_acc=0.660, g_loss=0.654 (bin_crossentr)
Acuracia do discriminador em amostras reais: 84%, falsas: 49%
epoch=40, batch=62/62, d_acc=0.510, g_loss=0.835 (bin_crossentr)
Acuracia do discriminador em amostras reais: 38%, falsas: 88%
epoch=50, batch=62/62, d_acc=0.610, g_loss=1.001 (bin_crossentr)
Acuracia do discriminador em amostras reais: 26%, falsas: 95%
epoch=60, batch=62/62, d_acc=0.610, g_loss=0.838 (bin_crossentr)
Acuracia do discriminador em amostras reais: 39%, falsas: 84%
epoch=70, batch=62/62, d_acc=0.550, g_loss=0.729 (bin_crossentr)
Acuracia do discriminador em amostras reais: 74%, falsas: 44%
epoch=80, batch=62/62, d_acc=0.670, g_loss=0.710 (bin_crossentr)
Acuracia do discriminador em amostras reais: 68%, falsas: 46%
epoch=90, batch=62/62, d_acc=0.620, g_loss=0.595 (bin_crossentr)
Acuracia do discriminador em amostras reais: 95%, falsas: 23%
epoch=100, batch=62/62, d_acc=0.550, g_loss=0.747 (bin_crossentr)
Acuracia do discriminador em amostras reais: 57%, falsas: 73%
```

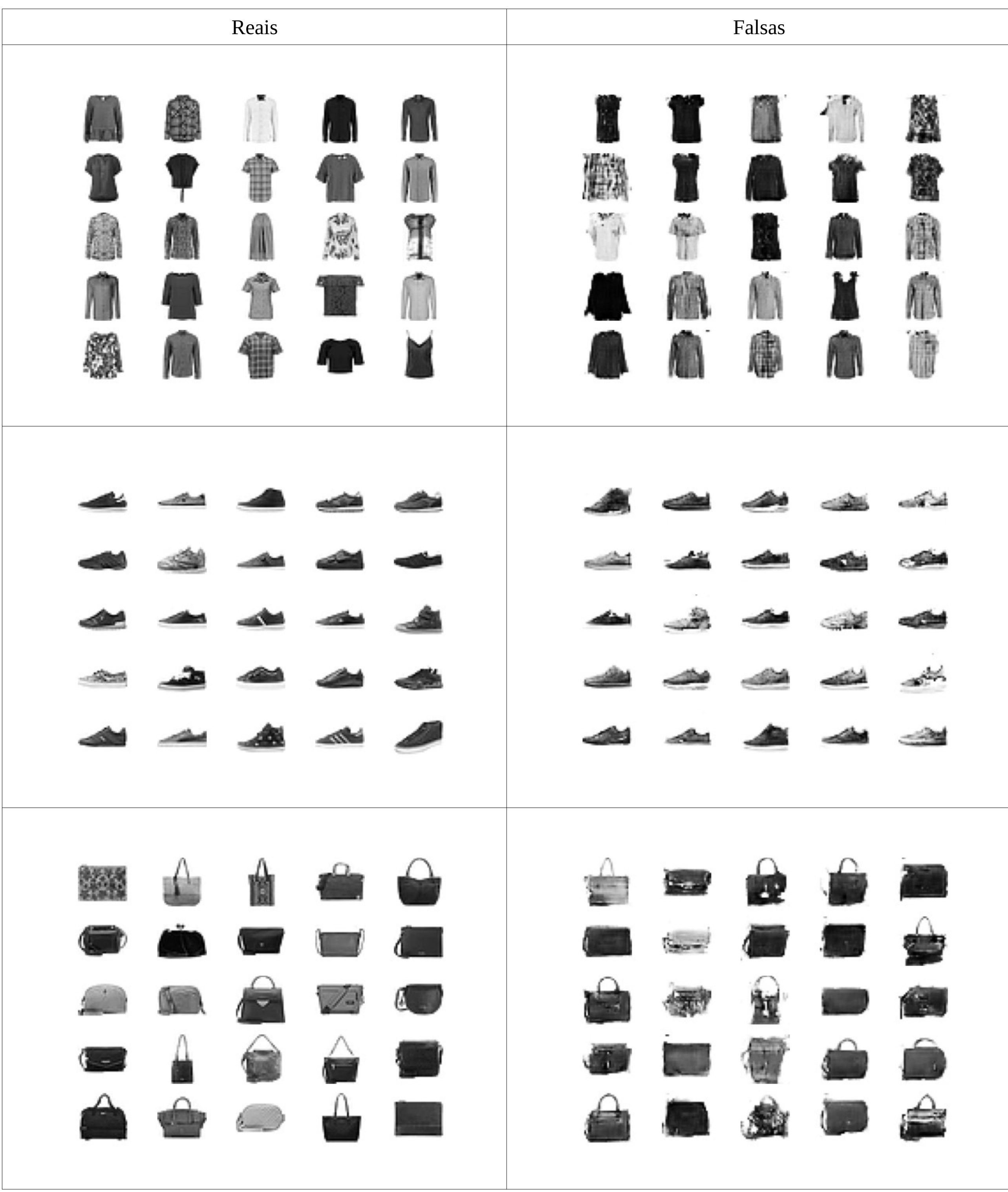
época 10	época 20
época 40	época 80
época 100	Dígitos reais (o programa chama de 000)

Figura 7: As imagens geradas pelo programa 1 (gan_completo5.py).

Depois, rodei o programa 1 acima usando fashion_mnist. Neste caso, usando learning rate lr=0.0002 (linhas 26 e 50) o treino não convergia. Assim, aumentei lr=0.0005. As imagens obtidas estão a seguir.

Reais	Falsas

Reais	Falsas


Reais
Falsas

O programa está em [https://colab.research.google.com/drive/1wBBUMIsJ6MI06JF5Iy_jhEHdw-CORzG-L?usp=sharing]

[PSI3472-2023. Aulas 11/12. Lição de casa #2. Vale 5,0.] Modifique o programa 1 acima para que gere automóveis do CIFAR10 (categoria índice 1). Imprima 25 automóveis originais e 25 automóveis gerados pela GAN. A figura abaixo mostra os sapos gerados desta forma. Provavelmente, os automóveis gerados terão qualidade visual ruim.

Figura: (Esquerda) Sapos originais do CIFAR10. (Direita) Imagens que obtive modificando o programa 1.

[PSI3472-2023. Aulas 11/12. Lição de casa extra. Vale +2,0.] Provavelmente, os automóveis gerados no exercício anterior terão qualidade visual ruim. Pesquise quais são as técnicas que permitem melhorar a qualidade das imagens geradas pelo GAN. Implemente uma delas e execute, mostrando que essa técnica consegue melhorar a qualidade das imagens.

Referências:

Explicação de GAN em português: https://www.deeplearningbook.com.br/introducao-as-redes-adversarias-generativas-gans-generative-adversarial-networks/

[Brownlee2019a] https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/

[Brownlee2019b] https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-an-mnist-handwritten-digits-from-scratch-in-keras/

[Goodfellow2014] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, Generative Adversarial networks, https://arxiv.org/abs/1406.2661

[PSI3472-2023. Aula 11 parte 2. Fim]