Exercicio_5_redes_neurais-2

November 17, 2021

0.1 SCC5809 - REDES NEURAIS ARTIFICIAIS E APRENDIZADO PROFUNDO

0.2 SCC - ICMC - USP

Profa. Roseli Ap. Francelin Romero 2o. sem. de 2021

Exercício de Rede GAN usando CNNs nos módulos:

modelo Discriminador modelo Generativo para geração de imagens (conjunto MNIST)

0.3 Exercício Prático 5 - GAN

0.3.1 Equipe 4:

- Bruno F. Bessa 5881890
- Leonardo Almeida 5834097
- Khennedy Bacule 12619430

0.4 Setup

[1]:
```
import tensorflow as tf
```

[2]:
```
!pip install imageio
!pip install git+https://github.com/tensorflow/docs
```

```
Requirement already satisfied: imageio in /usr/local/lib/python3.7/dist-packages
(2.4.1)
Requirement already satisfied: pillow in /usr/local/lib/python3.7/dist-packages
(from imageio) (7.1.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages
(from imageio) (1.19.5)
Collecting git+https://github.com/tensorflow/docs
  Cloning https://github.com/tensorflow/docs to /tmp/pip-req-build-haqqzg82
  Running command git clone -q https://github.com/tensorflow/docs /tmp/pip-req-
build-haqqzg82
Requirement already satisfied: astor in /usr/local/lib/python3.7/dist-packages
```

```
(from tensorflow-docs==0.0.0.dev0) (0.8.1)
Requirement already satisfied: absl-py in /usr/local/lib/python3.7/dist-packages
(from tensorflow-docs==0.0.0.dev0) (0.12.0)
Requirement already satisfied: protobuf>=3.14 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-docs==0.0.0.dev0) (3.17.3)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.7/dist-packages
(from tensorflow-docs==0.0.0.dev0) (3.13)
Requirement already satisfied: six>=1.9 in /usr/local/lib/python3.7/dist-
packages (from protobuf>=3.14->tensorflow-docs==0.0.0.dev0) (1.15.0)
```

[3]:
```
import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
from tensorflow.keras import layers
import time

from IPython import display
```

0.4.1 Carregando e pré-processando o dataset

Será utilizado o dataset MNIST para que seja treinado o modelo gerador e o discriminador.

[4]:
```
# Carregando o dataset
(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
```

Em passos futuros, vamos utilizar de ruídos aleatórios dados por uma distribuição normal entre [-1, 1]. Portanto, temos que nosso dataset também deve ser normalizado dessa maneira. Como temos imagens de 8 bits (valores de 0 a 255), vamos utilzar metade do valor total, isto é, 127.5, para normalizarmos os dados. Temos, portanto, a seguinte normalização:

$$\frac{x - 127.5}{127.5}$$

Percebemos que se substituirmos os valores de máximo e mínimo (0 e 255), teremos como resultado, respectivamente, -1 e 1.

[5]:
```
# Normalizando o dataset
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).
↪astype('float32')
train_images = (train_images - 127.5) / 127.5
```

[6]:
```
# Variáveis para embaralharmos o dataset e separarmos em batches
BUFFER_SIZE = 60000
BATCH_SIZE = 256
```

[7]:
```
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).
↪shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

0.5 Criação dos modelos

Ambos modelos gerador e discriminador são definidos utilizando um modelo sequencial do framework - Keras Sequential API.

0.5.1 Modelo Gerador

O modelo gerador utiliza de uma seed aleatória para fazer a imagem. `tf.keras.layers.Conv2DTranspose` faz essa construção. Ao final, devemos obter uma imagem 28x28x1. Faça um modelo gerador utilizando `tf.keras.Sequential` com os seguintes critérios: - Para cada camada, utilize o parâmetro `use_bias=False`. - Utilize Batch Normalization após cada camada (não necessária na de saída) - Utilize a função Leaky ReLU como ativação. - Utilize tanh como função de ativação da última camada. Siga a seguinte arquitetura: - Camada densa com $7 \cdot 7 \cdot 256$ com entrada (100,). A entrada será um seed (ruído aleatório) para servir de base para a construção da imagem. Escolhemos essa saída para conseguirmos gerar ao final facilmente uma imagem (28, 28, 1). - Camada de `Conv2DTranspose` para construir uma imagem. Ela deve possuir 128 filtros (5, 5) aplicando zero-padding. - Camada de `Conv2DTranspose` para construir uma imagem. Ela deve possuir 64 filtros (5, 5) aplicando zero-padding com stride de (2, 2). - Camada de saída `Conv2DTranspose` para construir a imagem final. Ela deve possui 1 saída com filtro (5, 5) aplicando zero-padding com stride de (2, 2).

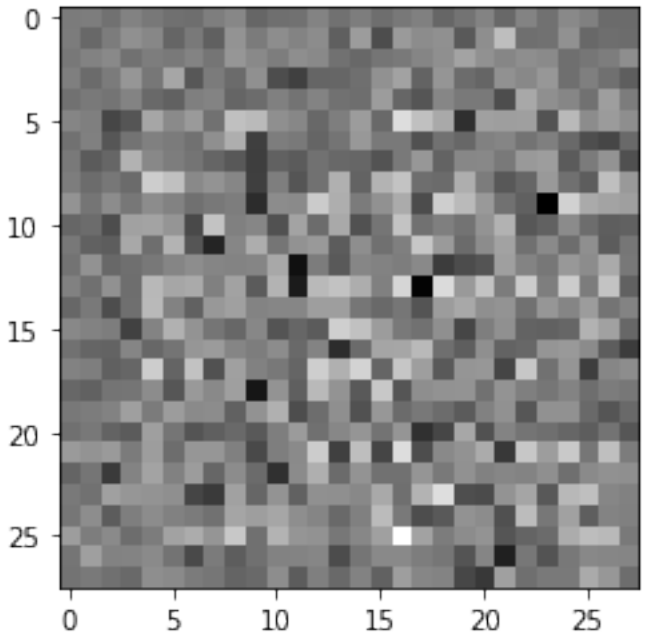
[8]:
```
def modelo_gerador():

    model = tf.keras.Sequential()
    # TODO: seu código da camada densa
    model.add(layers.Dense(7*7*256, input_shape=(100,), use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Reshape((7,7,256)))

    # TODO: seu código das camadas Conv2DTranspose
    model.add(layers.Conv2DTranspose(filters=128, kernel_size=5,␣
↪padding="same", use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(filters=64, kernel_size=5, padding="same",␣
↪strides=2, use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
```

```
    model.add(layers.Conv2DTranspose(filters=1, kernel_size=5, padding="same",␣
↪strides=2, use_bias=False, activation='tanh'))
    model.add(layers.BatchNormalization())

    return model
```

Usando o modelo gerador sem treinamento

[9]:
```
gerador = modelo_gerador()

seed = tf.random.normal([1, 100])
imagem_gerada = gerador(seed, training=False)
plt.imshow(imagem_gerada[0, :, :, 0], cmap='gray')
```

[9]:
```
<matplotlib.image.AxesImage at 0x7f57f0063dd0>
```



0.5.2 Modelo Discriminador

O modelo discriminador consiste em um classificador de imagem baseado em uma rede neural convolucional.

Na função abaixo, declare um modelo sequencial do keras (`tf.keras.Sequential`) com as seguintes camadas: - Camada convolucional de input $28 \times 28 \times 1$ (dimensão de cada imagem), com 64 filtros de tamanho 5×5, stride de 2 e zero-padding (`padding = 'same'`) - Função de ativação não linear leaky ReLU - Dropout com $p = 0.3$ - Camada convolucional com 128 filtros de tamanho 5×5,

stride de 2 e zero-padding - Função de ativação não linear leaky ReLU - Dropout com $p = 0.3$ - Flatten dos mapas de características obtidos - Camada densa de apenas um neurónio

[10]:
```
def modelo_discriminador():
    model = tf.keras.Sequential()

    # TODO: seu código aqui
    # 28x28x1: BW images
    model.add(layers.Conv2D(filters=64, kernel_size=5, strides=2,␣
 ↪activation="relu", padding="same", input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(.3))
    model.add(layers.Conv2D(filters=128, kernel_size=5, strides=2,␣
 ↪activation="relu", padding="same"))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(.3))
    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model
```

Uma vez com os modelos gerador e discriminador em mãos (e ainda não treinados), podemos classificar as imagens geradas como reais ou falsas.

Iremos treinar o modelo de forma que o resultado positivo indique imagens reais, enquanto valores negativos indiquem imagens falsas.

[11]:
```
discriminador = modelo_discriminador()
decision = discriminador(imagem_gerada)
print(decision)
```

```
tf.Tensor([[-0.00105526]], shape=(1, 1), dtype=float32)
```

0.6 Definindo a função de custo e otimizadores

Agora, devemos definir as funções de custo e quais serão os otimizadores para ambos os modelos para que possa ocorrer o treinamento.

[12]:
```
# Função de entropia cruzada para cálculo da loss
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

0.6.1 Discriminator loss

Com a função abaixo devemos quantificar quão bem o discriminador classifica as imagens reais e/ou imagens falsas.

Para isso, devemos comparar as predições do discriminador para as imagens reais ao array de uns e as predições do discriminador em imagens falsas, criadas pelo nosso modelo gerador, com um array de valores zeros.

Na função, atribua à `fake_loss` a entropia cruzada do output em cima de imagens criadas pelo modelo gerador em relação ao array de valores zeros.

[13]:
```
def discriminador_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    # TODO: defina a fake_loss
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)

    total_loss = real_loss + fake_loss
    return total_loss
```

0.6.2 Generator loss

A função de custo do modelo gerador é quantificada a partir do quão confuso foi a predição do modelo discriminador.

Assim, intuitivamente o que temos é que se o modelo gerador é capaz de gerar imagens boas, o discriminador irá classificar imagens falsas (0) como verdadeiras/reais (1), configurando uma baixa loss.

Para realizar a computação desta função de custo, devemos realizar a entropia cruzada entre o array de uns e o array da predição do nosso discriminador para as imagens falsas.

Considerando um array da predição dentro do conjunto de imagens geradas como {0 1 0 1 0 1 0 0 1 0 0 1}, temos

- Número de imagens falsas classificadas como falsas: 7
- Número de imagens falsas classificadas como reais: 5

Ao realizarmos a entropia cruzada com um vetor {1 1 1 1 1 1 1 1 1 1 1 1}, iremos **penalizar para cada um dos zeros que aparecem**, uma vez que indicam uma classificação certa do modelo discriminador.

[14]:
```
def gerador_loss(fake_output):
  # TODO: defina a loss do gerador
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

Vamos definir os otimizadores para cada modelo

[15]:
```
gerador_opt = tf.keras.optimizers.Adam(1e-4)
discriminador_opt = tf.keras.optimizers.Adam(1e-4)
```

0.7 Loop de treinamento

[16]:
```
epocas = 70
seed_dim = 100
n_exemplos = 16
seed = tf.random.normal([n_exemplos, seed_dim])
```

Temos que o treinamento começa com o **gerador** recebendo uma seed como input e produzindo uma imagem. Após isso, o discriminador é usado para classificar as imagens do dataset original e as produzidas pelo gerador. Calculamos a **função perda (loss)** de cada modelo e seus gradientes.

[17]:
```
@tf.function
def step_treino(images):
    seed = tf.random.normal([BATCH_SIZE, seed_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
      imagens_geradas = gerador(seed, training=True) # Construindo imagens␣
↪falsas pelo modelo gerador

      real_output = discriminador(images, training=True) # Previsão do␣
↪discriminador com imagens reais
      fake_output = discriminador(imagens_geradas, training=True) # Previsão do␣
↪discriminador com as imagens geradas

      # TODO
      # Calcule a loss do gerador
      gen_loss = gerador_loss(fake_output)
      # Calcule a loss do discriminador
      disc_loss = discriminador_loss(real_output, fake_output)

    gradientes_gerador = gen_tape.gradient(gen_loss, gerador.
↪trainable_variables)
    gradientes_discriminador = disc_tape.gradient(disc_loss, discriminador.
↪trainable_variables)

    gerador_opt.apply_gradients(zip(gradientes_gerador, gerador.
↪trainable_variables))
    discriminador_opt.apply_gradients(zip(gradientes_discriminador,␣
↪discriminador.trainable_variables))
```

[18]:
```
def treino(dataset, epocas):
  for epoca in range(epocas):
    start = time.time()

    for image_batch in dataset:
      step_treino(image_batch)
```

```
    # Produzindo imagens para formarmos um GIF ao final
    display.clear_output(wait=True)
    gerar_imagens(gerador, epoca + 1, seed)

    print ('Tempo de execução para a época {} é {} segundos'.format(epoca + 1,␣
↪time.time()-start))

  # Salvando imagens para criação do GIF
  display.clear_output(wait=True)
  gerar_imagens(gerador, epocas, seed)
```

Função para gerarmos e salvarmos as imagens

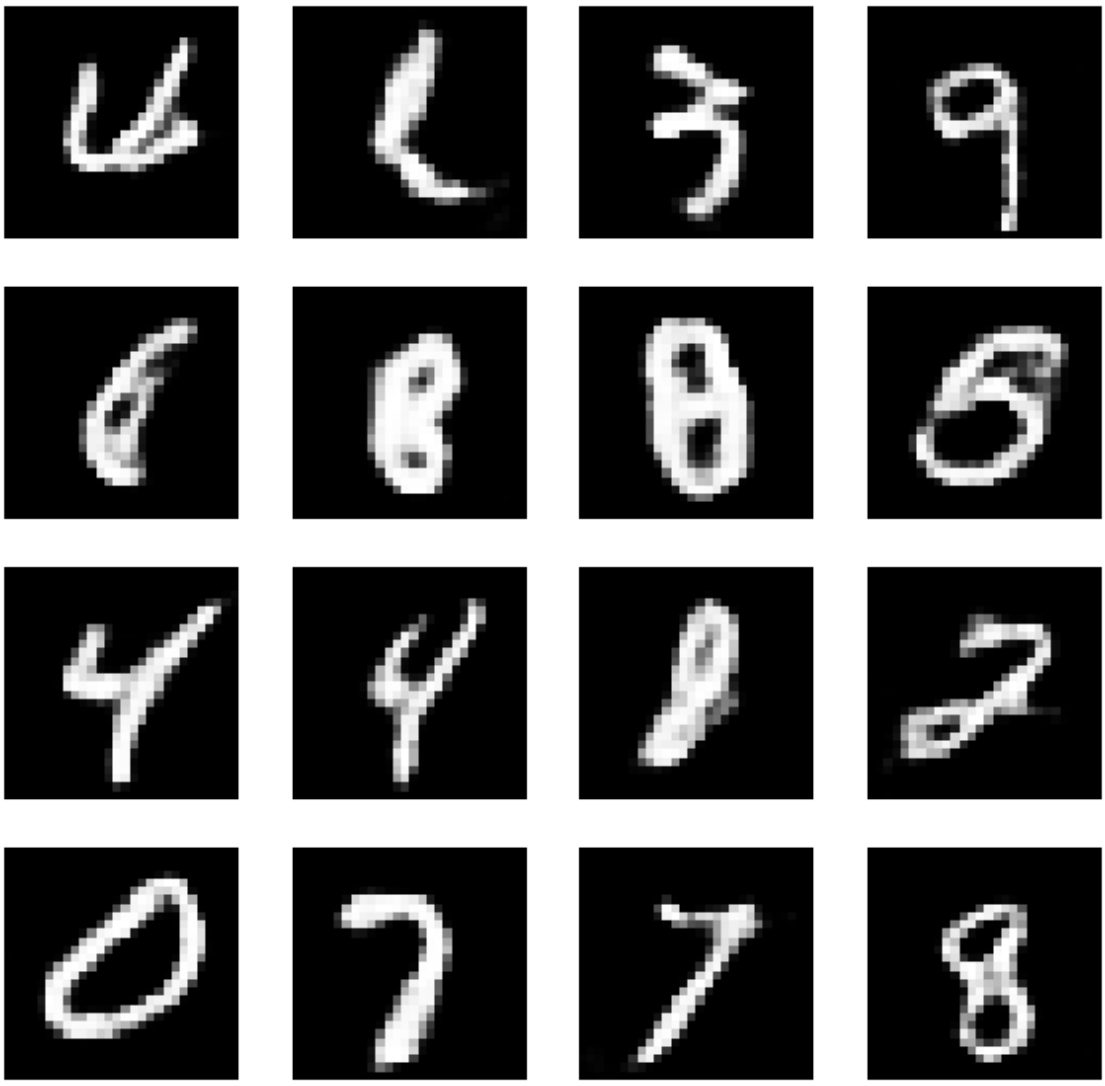
[19]:
```
def gerar_imagens(model, epoca, test_input):
  preds = model(test_input, training=False)

  fig = plt.figure(figsize=(10, 10))

  for i in range(preds.shape[0]):
      plt.subplot(4, 4, i+1)
      plt.imshow(preds[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
      plt.axis('off')

  plt.savefig('imagem_epoca_{:04d}.png'.format(epoca))
  plt.show()
```

0.8 Treinamento

Chamamos agora a função de treino definida acima para treinar simultaneamente os modelos.

Vamos analisar as imagens geradas cada época.

[20]:
```
treino(train_dataset, epocas)
```

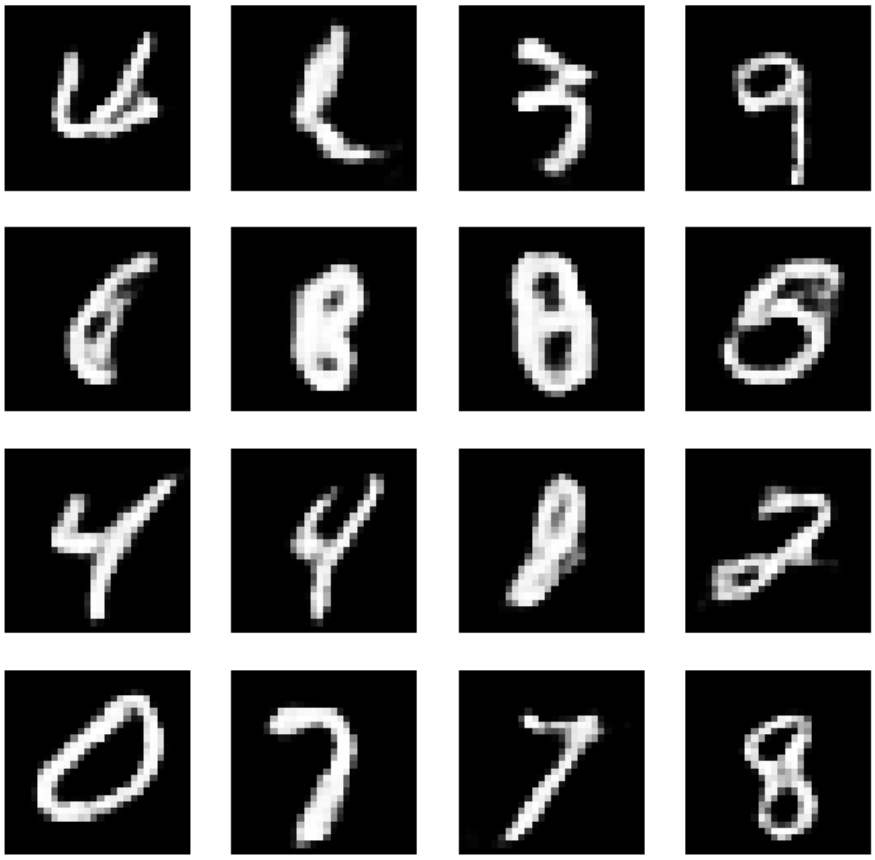


0.9 Criando um GIF

[21]:
```
def display_image(epoch_no):
  return PIL.Image.open('imagem_epoca_{:04d}.png'.format(epoch_no))
```

[22]:
```
display_image(epocas)
```

[22]:



Vamos usar `imageio` para criar um GIF animado a partir das imagens salvas

[23]:
```
anim_file = 'dcgan.gif'

with imageio.get_writer(anim_file, mode='I') as writer:
  filenames = glob.glob('image*.png')
  filenames = sorted(filenames)
  for filename in filenames:
    image = imageio.imread(filename)
    writer.append_data(image)
  image = imageio.imread(filename)
  writer.append_data(image)
```

[24]:
```
import tensorflow_docs.vis.embed as embed
embed.embed_file(anim_file)
```

[24]:
```
<IPython.core.display.HTML object>
```

[ ]:
```

```