Aula 8

Autoencoders - Parte 2

Eduardo Lobo Lustosa Cabral

1. Objetivos

Criar um autoencoder para processar música.

Criar um autoencoder para eliminar ruído de imagens.

Criar um autoencoder para detectar anomalias.

Importação das bilbiotecas necessárias

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import os
print(tf.__version__)
```

```
2.4.1
```

2. Autoencoder para processar música

Antes de definir e treinar autoencoder para processar música, devemos primeiro escolher que tipo de dado vamos utilizar para representar músicas.

Uma música pode ser representada de forma contínua ou discreta.

- A forma contínua mais comum é um sinal de áudio, normalmente armazenado como um arquivo WAV.
- As formas discretas mais comuns incluem arquivos MIDI (Musical Instrument Digital Interface), pianoroll e texto, que representam partituras de músicas.

Vamos utilizar sinal de áudio, por não exigir um software para gerar música a partir de uma partitura.

Importação de bibliotecas específicas

In [2]:
```
import librosa
from IPython import display
from IPython.display import clear_output
import glob
import imageio
import time
import IPython.display as ipd
```

2.1 Modelo do autoencoder para processar música

Vamos utilizar um autoencoder com camadas convolucionais 1D para processar o sinal de áudio das músicas.

Note que o sinal de áudio de uma música é uma série temporal.

Observa-se que camadas convolucionais 1D são muito utlizadas para processar séries temporais.

A vantagem de se utilizar camadas convolucionais 1D para processar séries temporais é que são muito mais rápidas do que as camadas recorrentes, tanto no treinamento como na previsão. No entanto as camadas recorrentes tendem a serem mais eficientes para processar séries temporais.

Camadas convolucionais 1D funcionam exatamente como as camadas convolucionais 2D. A única diferença é que o sinal de entrada tem somente uma única dimensão. Pode-se pensar que uma camada convolucional 1D processa imagens que possuem somente uma única linha.

O tensor de entrada de uma camada convolucional 1D deve ter a seguinte estrutura:

- Eixo 0: eixo dos exemplos;
- Eixo 1: eixo do "comprimento" do dado da série → é o equivalente ao eixo das linhas das camadas convolucionais 2D;
- Eixo 2: eixo das características → é o equivalente ao eixo dos canais das camadas convolucionais 2D.

O eixo das características consiste, por exemplo, em vários tipos de dados dos quais depende a saída.

Para uma série temporal para prever temperatura em um dado local, teríamos como características da série, por exemplo:

1. Hora do dia;
2. Histórico de precipitação;
3. Histórico de pressão atmosférica;
4. Histórico de velocidade do vento;
5. Época do ano;
6. etc.

Nesse caso, a saída desejada da série seria o histórico de temperatura.

2.2 Carregar e processar dados

Vamos usar o conjunto de dados de música GTZAN, que é considerado o MNIST de sons musicais. Esse conjunto de dados pode ser obtido no Kaggle (https://www.kaggle.com/andradaolteanu/gtzan-dataset-music-genre-classification (https://www.kaggle.com/andradaolteanu/gtzan-dataset-music-genre-classification))

O GTZAN consiste de uma coleção de 10 gêneros musicais com 100 arquivos de áudio cada, todos com uma duração de 30 segundos. Esses dados são originalmente usados para classificação de gêneros musicais. Os arquivos de música estão no formato de aúdio WAV.

Para processar os arquivos de aúdio usaremos a biblioteca Python Librosa (https://librosa.org/doc/latest/index.html (https://librosa.org/doc/latest/index.html)).

Na célula abaixo é definida uma função para obter todos os arquivos dentro de uma pasta para depois serem carregados usando a função `librosa.load()` .

Nessa aula vamos usar o gênero de música clássica.

In [4]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [4]:
```
%cd /content/drive/MyDrive/Dados_Trabalhos
!ls
```

```
/content/drive/MyDrive/Dados_Trabalhos
cangurus_pessoas     classical_0011.png  jazz_0000.png  jazz_0013.png
cityscapes_data.zip  classical_0012.png  jazz_0001.png  jazz_0014.png
classical_0000.png   classical_0013.png  jazz_0002.png  jazz_0015.png
classical_0001.png   classical_0014.png  jazz_0003.png  jazz_0016.png
classical_0002.png   classical_0015.png  jazz_0004.png  jazz_0017.png
classical_0003.png   classical_0016.png  jazz_0005.png  jazz_0018.png
classical_0004.png   classical_0017.png  jazz_0006.png  jazz_0019.png
classical_0005.png   classical_0018.png  jazz_0007.png  jazz_0020.png
classical_0006.png   classical_0019.png  jazz_0008.png  npl_data.csv
classical_0007.png   classical_0020.png  jazz_0009.png  TsukubaStereo.zip
classical_0008.png   classical_cvae.gif  jazz_0010.png  Unet.h5
classical_0009.png   Data                jazz_0011.png  Untitled
classical_0010.png   GTZAN_data.zip      jazz_0012.png
```

In [4]:
```
# Função para obter nomes de arquivos em um diretório
def DatasetLoader(file_dir, style):
    music_list = np.array(sorted(os.listdir(file_dir+'/'+style)))
    TrackSet = [(file_dir)+'/'+style+'/%s'%(x) for x in music_list]

    return TrackSet

# Define nome do diretório
file_dir = 'GTZAN_data/Data/genres_original'
#file_dir = 'Data/genres_original'

# Obtém nomes dos aquivos no diretório de músicas clássicas
TrackSet = DatasetLoader(file_dir, 'classical')
print('Número de arquivos de música:', len(TrackSet))

print('\nNome dos primeiros arquivos:\n', TrackSet[:5])
```

```
Número de arquivos de música: 98

Nome dos primeiros arquivos:
 ['GTZAN_data/Data/genres_original/classical/classical.00000.wav', 'GTZAN_data/Data/genres_original/classi
cal/classical.00001.wav', 'GTZAN_data/Data/genres_original/classical/classical.00002.wav', 'GTZAN_data/Dat
a/genres_original/classical/classical.00003.wav', 'GTZAN_data/Data/genres_original/classical/classical.000
04.wav']
```

Carrega arquivos de música no formato WAV.

In [5]:
```
# Define frequência de amostragem da música
freq = 8000

# Define uma música para mostrar exemplo
index = 12

# Carrega música selecionada
sample = TrackSet[index]
sample_, sampling_rate = librosa.load(sample,sr=freq, offset=0.0, duration=30)

# Toca música
ipd.Audio(sample_,rate=freq)
```

Out[5]:

0:00 / 0:00

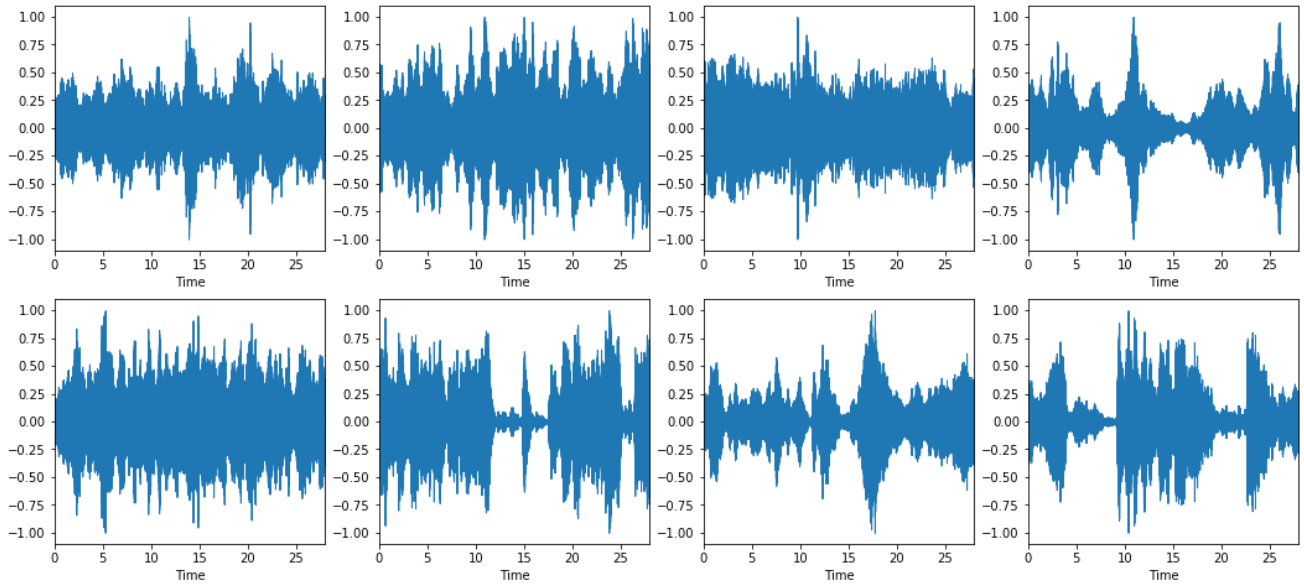
Na célula abaixo é criada uma função para carregar um arquivo de música e selecionar somente 10 segundos da música.

In [6]:
```
def load(file, duration):
    # Carrega arquivo de música
    data, sampling_rate = librosa.load(file,sr=freq, offset=0.0, duration=30)

    # Define número de amostras usadas de cada música
    data = data[:duration*freq]

    # Equalização do som
    vmax = np.max(np.abs(data))
    data = data/vmax

    return data
```

- Observa-se que o som das músicas são equalizados de forma que todas tenham as mesmas amplitudes máxima e mínima. Isso facilita o treinamento do autoencoder.

Na célula abaixo a função `load()` é usada para carregar alguns arquivos de músicas.

In [7]:
```
# Importa função da bilioteca librosa para mostrar onda de áudio
import librosa.display

# Define duração da música a ser usada em segundos
duration = 28

# Carrega música e mostra as amostras em função do tempo
plt.figure(figsize=(18,8))
for i in range(8):
    plt.subplot(2, 4, i + 1)

    # Carrega arquivo de música
    audio = load(TrackSet[i], duration)
    librosa.display.waveplot(audio, sr=freq)
```



Na célula abaixo a função `load()` é usada para carregar um número determinado de músicas para serem usadas para treinar o autoencoder.

In [9]:
```
# Define número de músicas
num_music = 50

# Carrega os arquivos desejados
music_data = []
for file in TrackSet[:num_music]:
    data = load(file, duration)
    music_data.append(data)

# transforma lista de ondas sonaras em tensor Numpy
music_data = np.stack(music_data)
print('Dimensão do tensor de músicas:', music_data.shape)

# Cria variável com número de amostras com músicas
music_length = music_data.shape[1]
print('Número de amostras por música:', music_length)
```

```
Dimensão do tensor de músicas: (50, 224000)
Número de amostras por música: 224000
```

- Observe que cada música consiste de um exemplo com 224.000 amostras de som.

Vamos tocar uma música do tensor para verificação.

In [10]:
```
# Seleciona música
index = 2

# Toca música
ipd.Audio(music_data[index],rate=freq)
```

Out[10]:

0:00 / 0:00

Vamos selecionar somente uma parte das músicas para não demorar muito o treinamento do autoencoder.

In [11]:
```
# Número de amostras desejadas
num_amostras = 131072

# Pega parte das músicas
train_data = music_data[:, :num_amostras]
print('Dimensão do tensor de músicas de treinamento:', train_data.shape)
```

```
Dimensão do tensor de músicas de treinamento: (50, 131072)
```

Como o comprimento das músicas é muito grande vamos dividi-las em partes.

In [12]:
```
# Número de partes em que cada música será dividida
music_parts = 32

# Comprimento de cada parte de música
part_length = int(num_amostras/music_parts)

# Redimensionamento do tensor de músicas para o treinamento
train_data = np.reshape(train_data, (num_music*music_parts, part_length,1))

print('Número de músuicas:', num_music)
print('Número de partes de músicas;', music_parts)
print('Comprimento de cada parte de música:', part_length)
print('Dimensão do tensor de treinamento:', train_data.shape)
```

```
Número de músuicas: 50
Número de partes de músicas; 32
Comprimento de cada parte de música: 4096
Dimensão do tensor de treinamento: (1600, 4096, 1)
```

2.3 Configuração do codificador

O codificador será composto pelas seguintes camadas:

- Quatro camadas convolucionais 1D, com filtros de dimensão 3x1, `strides=2` , função de ativação `LeakyReLU` e `padding=same` ;
- Uma camada `Flatten` para adequar a saída das camadas convolucionais para a camada de densa de saída;
- Camada de saída tipo densa com função de ativação linear.

O número de filtros das camadas convolucionais deve aumentar porgessivamente da primeira até a última.

Observe que a dimensão do espaço latente, definida na variável `latent_dim` é um hiperparâmetro que temos que escolher para gerar o melhor resultado. Dessa forma, é interessante testar alguns valores.

In [13]:

```
# Importa classe camadas e modelo funcional
from tensorflow.keras import layers
from tensorflow.keras.models import Model
import tensorflow.keras.backend as K

# Dimensão do espaço latente
latent_dim = 1024

# Camada de entrada
inputs = layers.Input(shape=(part_length,1))

# Camadas convolucionais com normalização de batelada e função de ativação LeakyRelu
x = layers.Conv1D(32, 3, strides=2, activation=layers.LeakyReLU(), padding='same')(inputs)
x = layers.Conv1D(32, 3, strides=2, activation=layers.LeakyReLU(), padding='same')(x)
x = layers.Conv1D(64, 3, strides=2, activation=layers.LeakyReLU(), padding='same')(x)
x = layers.Conv1D(64, 3, strides=2, activation=layers.LeakyReLU(), padding='same')(x)

# Dimensão da saída das camadas convolucionais
shape_before_flattening = K.int_shape(x)[1:]

# Camada de Flatten
x = layers.Flatten()(x)

# Camadas densas para calcular média e logaritmo da variância
encoder_output = layers.Dense(latent_dim)(x)

# Instancia codificador
encoder = Model(inputs=inputs, outputs=encoder_output)

# Sumário do amostrador
encoder.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 4096, 1)]         0
_________________________________________________________________
conv1d (Conv1D)              (None, 2048, 32)          128
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 1024, 32)          3104
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 512, 64)           6208
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 256, 64)           12352
_________________________________________________________________
flatten (Flatten)            (None, 16384)             0
_________________________________________________________________
dense (Dense)                (None, 1024)              16778240
=================================================================
Total params: 16,800,032
Trainable params: 16,800,032
Non-trainable params: 0
_________________________________________________________________
```

2.4 Configuração do decodificador

A configuração do decodificador que iremos usar é a seguinte:

- Uma camada densa para gerar elementos suficientes para a primeira camada convolucional a partir da representação latente;
- Uma camada de redimensionamanro para adequar a saída da camda densa para a entrada da primeira camada convolucional 1D;
- Quatro camadas convolucionais 1D transpostas, com `stride=12` e função de ativação LeakyReLu;
- Uma camada convolucional 1D, com um único filtro, de dimensão 1x1, com `stride=1` e função de ativação linear.

A função de ativação tangente hiperbólica na camada de saída é usada nesse caso porque as amostras de som estão no intervalo entre 0 e 1.

In [14]:

```
# Camada de entrada
decoder_input = layers.Input(shape = (latent_dim,) , name = 'decoder_input')

# Camada densa para ajustar dimensão do espaço latente para a entrada da primeira camada convolucional
x = layers.Dense(tf.math.reduce_prod(shape_before_flattening))(decoder_input)

# Redimensiona vetor para a dimensão necessária par a comada convolucional 1D
x = layers.Reshape(shape_before_flattening)(x)

# Camadas convolucionais transpostas
x = layers.Conv1DTranspose(64, kernel_size=3, strides=2, padding='same', activation=layers.LeakyReLU()
x = layers.Conv1DTranspose(64, kernel_size=3, strides=2, padding='same', activation=layers.LeakyReLU()
x = layers.Conv1DTranspose(32, kernel_size=3, strides=2, padding='same', activation=layers.LeakyReLU()
x = layers.Conv1DTranspose(32, kernel_size=3, strides=2, padding='same', activation=layers.LeakyReLU()

# Camada convolucional padrão com um único filtro de dimensão 1x1
decoder_output = layers.Conv1D(1, kernel_size=1, strides=1, padding='same', activation='tanh')(x)

# Instancia decodificador
decoder = Model(decoder_input, decoder_output)

# Sumario
decoder.summary()
```

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
decoder_input (InputLayer)   [(None, 1024)]            0
_________________________________________________________________
dense_1 (Dense)              (None, 16384)             16793600
_________________________________________________________________
reshape (Reshape)            (None, 256, 64)           0
_________________________________________________________________
conv1d_transpose (Conv1DTran (None, 512, 64)           12352
_________________________________________________________________
conv1d_transpose_1 (Conv1DTr (None, 1024, 64)          12352
_________________________________________________________________
conv1d_transpose_2 (Conv1DTr (None, 2048, 32)          6176
_________________________________________________________________
conv1d_transpose_3 (Conv1DTr (None, 4096, 32)          3104
_________________________________________________________________
conv1d_4 (Conv1D)            (None, 4096, 1)           33
=================================================================
Total params: 16,827,617
Trainable params: 16,827,617
Non-trainable params: 0
_________________________________________________________________
```

2.5 Autoencoder completo

Para criar o autoencoder completo e treiná-lo temos que juntar o codificador e o decodificador.

In [15]:
```
# Camada de entrada
inputs = layers.Input(shape=(part_length,1))

# Inclui codificador
x = encoder(inputs)

# Incluir decodificador
decoder_output = decoder(x)

# Instância AEV
VAE = Model(inputs, decoder_output)

# Summario do AEV
VAE.summary()
```

```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 4096, 1)]         0
_________________________________________________________________
model (Functional)           (None, 1024)              16800032
_________________________________________________________________
model_1 (Functional)         (None, 4096, 1)           16827617
=================================================================
Total params: 33,627,649
Trainable params: 33,627,649
Non-trainable params: 0
_________________________________________________________________
```

2.6 Compilação e treinamento do autoencoder

Vamos compilar o autoencoder com a seguinte configuração:

- Otimizador: Adam com taxa de aprendizado de 0,001
- Função de custo: "mean square error"
- Métrica: "mean absolute error"

Observa-se que o erro quadrático médio é usado em razão da função de ativação da camada de saída do codificador ser tangente hiperbólica e os valores de saída estarem no intervalo entre 0 e 1.

In [16]:
```
# Define otimizador Adam
rms = tf.keras.optimizers.Adam(lr=0.001, decay=2e-03)

# Compilação do autoencoder
VAE.compile(optimizer=rms, loss='mse', metrics=['mae'])
```

Vamos treinar o autoencoder usando 300 épocas e lotes de 64 elementos

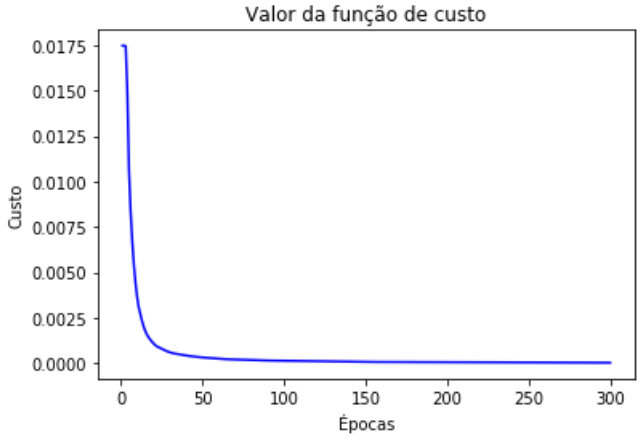
In [17]:
```
results = VAE.fit(train_data, train_data, epochs=300, batch_size=64, verbose=2, shuffle=True)
```

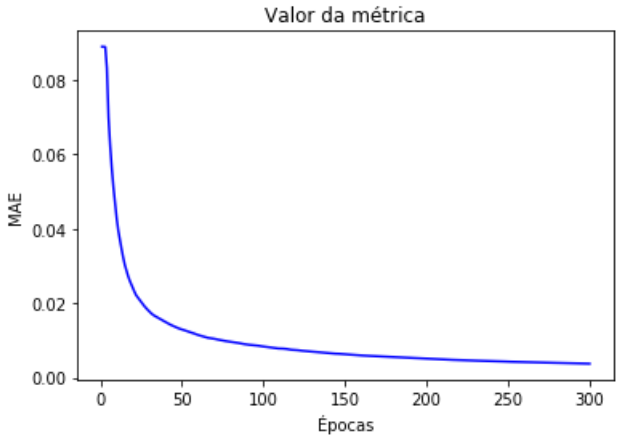
```
Epoch 11/300
25/25 - 20s - loss: 0.0031 - mae: 0.0395

Epoch 12/300
25/25 - 20s - loss: 0.0027 - mae: 0.0369
Epoch 13/300
25/25 - 20s - loss: 0.0023 - mae: 0.0345
Epoch 14/300
25/25 - 20s - loss: 0.0020 - mae: 0.0323
Epoch 15/300
25/25 - 20s - loss: 0.0018 - mae: 0.0303
Epoch 16/300
25/25 - 20s - loss: 0.0016 - mae: 0.0287
Epoch 17/300
25/25 - 22s - loss: 0.0014 - mae: 0.0273
Epoch 18/300
25/25 - 21s - loss: 0.0013 - mae: 0.0260
Epoch 19/300
25/25 - 20s - loss: 0.0012 - mae: 0.0250
Epoch 20/300
25/25 - 20s - loss: 0.0011 - mae: 0.0240
```

Gráficos do processo de treinamento.

In [18]:

```
def plot_train(history):
    history_dict = history.history

    # Salva custos, métricas em vetores
    custo = history_dict['loss']
    acc = history_dict['mae']

    # Cria vetor de épocas
    epocas = range(1, len(custo) + 1)

    # Gráfico dos valores de custo
    plt.plot(epocas, custo, 'b', label='Custo - treinamento')
    plt.title('Valor da função de custo')
    plt.xlabel('Épocas')
    plt.ylabel('Custo')
    plt.show()

    # Gráfico dos valores da métrica
    plt.plot(epocas, acc, 'b', label='mae')
    plt.title('Valor da métrica')
    plt.xlabel('Épocas')
    plt.ylabel('MAE')
    plt.show()

plot_train(results)
```





- Observa-se que o erro absoluto médio é menor do 1% do valor máximo das amostras de som.

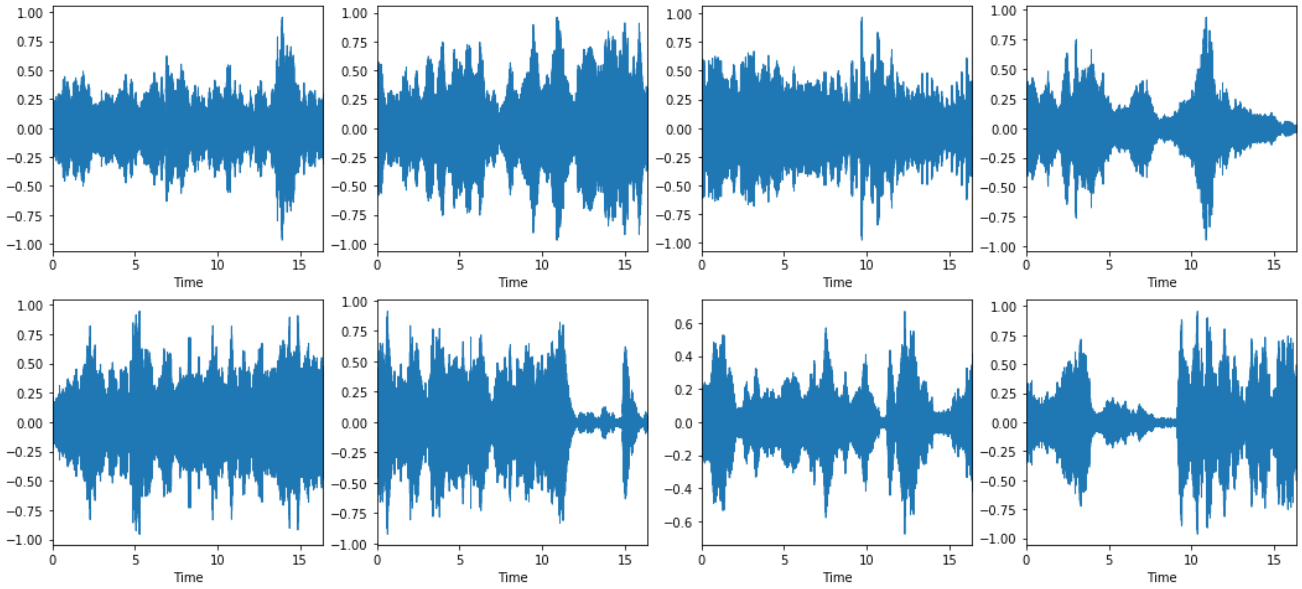
Observa-se que quanto maior a dimensão do espaço latente melhor o desepenho do autoencoder, porém existe um tamanho máximo para o qual problemas de overfitting fazem com que o eempenho autoencoder diminua.

3.8 Comparação das saídas previstas pelo autoencoder com as entradas

Para verificar o desempenho do autoencoder, vamos reconstruir algumas músicas usadas nos dados de treinamento.

In [19]:

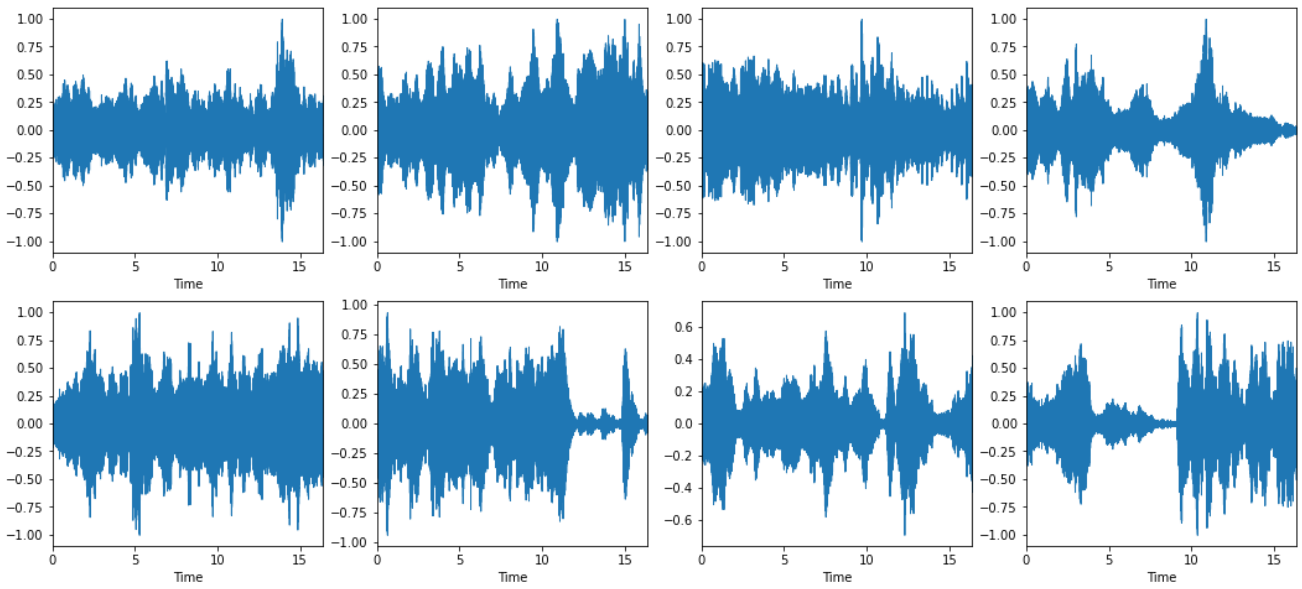
```
# Calcula dados reconstriuídos pelo AE
prev_data = VAE.predict(train_data)

# Redimensiona aídas previstas para agrupar partes das músicas
prev_data = np.reshape(prev_data, (num_music, music_parts*part_length))

# Mostra figuras das 12 primeiras ondas sonoras reconstruídas
plt.figure(figsize=(18,8))
for i in range(8):
    plt.subplot(2, 4, i + 1)
    sample = prev_data[i]
    librosa.display.waveplot(sample, sr=freq)
plt.show()
```



Para poder comparar com as ondas sonoras das músicas originais vamos mostras as figuras das 8 primeiras músicas originais.

In [91]:

```
# Mostra as figuras das primeiras 12 ondas sonoras originais
import librosa.display
plt.figure(figsize=(18,8))
for i in range(8):
    plt.subplot(2, 4, i + 1)
    sample = music_data[i,:num_amostras]
    librosa.display.waveplot(sample, sr=freq)
```



Vamos também ouvir algumas músicas reconstruídas e as suas correspondentes originais para analisar a semelhança.

In [24]:
```
# Seleciona musica
index = 6
sample = prev_data[index]
ipd.Audio(sample,rate=freq)
```

Out[24]:

0:00 / 0:00

In [25]:
```
# Toca música reconstruída correspondente à música original anteriormente tocada
sample = music_data[index, :num_amostras]
ipd.Audio(sample,rate=freq)
```

Out[25]:

0:00 / 0:00

- Observe que as imagens das ondas sonoras reconstruídas são praticamente iguais às originais. Além disso, o som das músicas reconstruídas são iguais ao som das músicas originais.

3.9 Geração de novas músicas

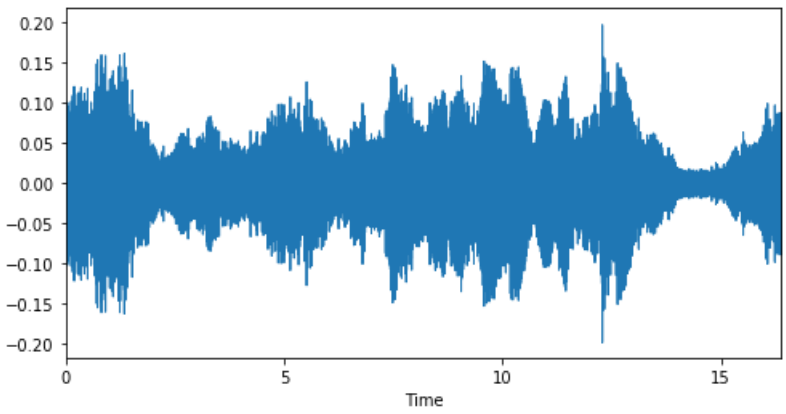
Vamos combinar a representação latente de uma ou mais músicas para gerar novas músicas.

Na célula abaixo é apresentado um exemplo onde se combinam duas representações latentes ponderadas igualmente. Observa-se que é possível usar pesos de ponderaçao difrentes e a soma dos pesos deve ser igual a 1.

In [27]:

```
# Seleciona músicas originais de forma aleatória
index1 = np.random.randint(0, num_music)
index2 = np.random.randint(0, num_music)
print(index1, index2)

# calcula representações latentes
z = encoder(train_data)

# Combina as duas representações latentes igualmente
z_mix = np.zeros((music_parts,latent_dim))
for i in range(music_parts):
    i1 = index1*music_parts + i
    i2 = index*music_parts + i
    z_mix[i] = 0.75*z[i1] + 0.25*z[i2]

# Gera nova música
music = decoder.predict(z_mix)
music = np.reshape(music, (music_parts*part_length))

# Mostra figura com onda somnora
plt.figure(figsize=(8,4))
librosa.display.waveplot(music, sr=freq)
plt.show()

# Toca música
ipd.Audio(music,rate=freq)
```

```
33 15
```



Out[27]:

0:00 / 0:00

- Observa-se que o resultado não é muito bom, sendo que as músicas geradas possuem algum ruído. Porém, isso decorre do fato de que um autoencoder não é feito para gerar novos dados e sim comprimir dados.
- Na próxima aula veremos como usar um autoencoder variacional para gerar novos dados.

3.10 Análise dos resultados

Para concluir, podemos observar que:

1. O autencoder é capaz de representar músicas a partir de uma vetor de dimensão inúmeras vezes menor do que o número de amostras das músicas.
2. O autoencoder é capaz de reconstruir músicas com erro muito pequeno a partir da representação latente.
3. O autoencoder é capaz de gerar novas músicas originais a partir da combinação do espaço latente de algumas músicas.

3. Eliminação de ruídos em imagens

Um autoencoder pode ser treinado para eliminar ruídos em imagens. De fato essa técnica é bastante usada e eliminação de ruídos em imagens é outra aplicação de autoencoders.

Nesse caso o autoencoder é treinado com as imagens de entrada com ruído e as imagens de saída sendo as imagens originais sem ruído. Dessa forma, o autoencoder aprende a subtrair o ruído das imagens e gerar imagens de melhor qualidade.

Observa-se que o autoencoder não recebe a imagem original na sua entrada, mas é esperado que ele regenere a imagem original sem ruído na saída.

Na Figura 6 é apresentado um esquema de um autoencoder treinado para receber imagens com ruídos e gerar a imagem sem rúido.

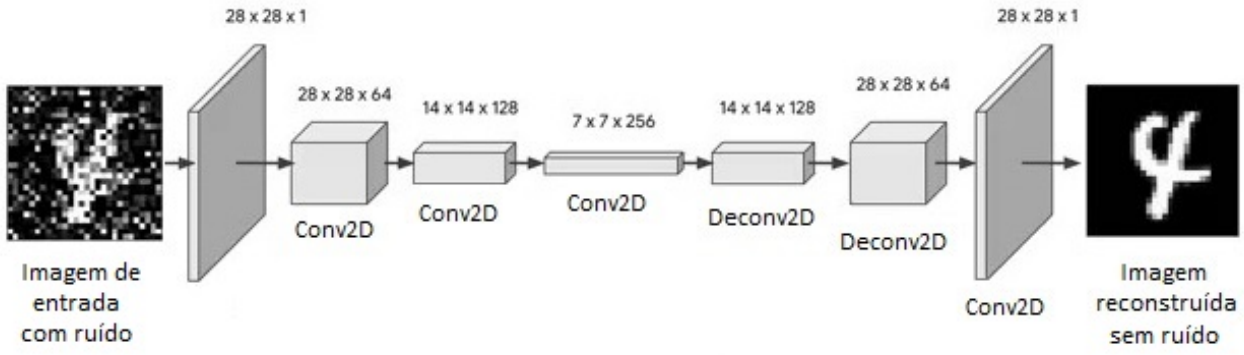


Figura 6 - Esquema de um autoencoder para eliminar ruído de imagens.

Para exemplificar essa aplicação de um autoencoder vamos utilizar o conjunto de dígitos MNIST.

3.1 Carregar e processar dados

A célula abaixo carrega o conjunto de dígitos MNIST da coleção do Keras.

In [ ]:
```
(x_train_orig, y_train), (x_test_orig, y_test) = tf.keras.datasets.mnist.load_data()

print('Dimensão dos dados de entrada de treinamento =', x_train_orig.shape)
print('Dimensão dos dados de entrada de teste =', x_test_orig.shape)
print('Dimensão dos dados de saída de treinamento =', y_train.shape)
print('Dimensão dos dados de saída de teste =', y_test.shape)
```

Vamos redimensionar as imagens para que tenham dimensão de 32x32 pixels. Isso é feito para garantir que as dimensões dos tensores quando as imagens forem reduzidas pelo codificador sejam sempre números inteiros.

In [ ]:
```
from skimage.transform import resize

x_train = resize(x_train_orig, (60000, 32,32), anti_aliasing=True)
x_test = resize(x_test_orig, (10000, 32,32), anti_aliasing=True)

print('Dimensão dos dados de entrada de treinamento =', x_train.shape)
print('Dimensão dos dados de entrada de teste =', x_test.shape)
```

- Observe que a função `resize` da biblioteca SkiImage já realiza a normalização dos pixels das imagens para valores reais entre 0 e 1.

Vamos usar um autoencoder com camadas convolucionais, assim, não precisamos transformar as imagens em vetores.

3.2 Introdução de ruído nas imagens

As imagens de entrada são as imagens originais adicionadas de ruído gaussiando. Vamos testar diferentes desvios padrões para verificar o desempenho do autoencoder em eliminar ruídos.

In [ ]:
```
# Número de exemplos de treinamento e teste
m_train = x_train.shape[0]
m_test = x_test.shape[0]

# Define desvio padrão do ruído
noise_std = 0.5

# Adiciona ruído na simagens de treinamento e teste
x_train_noise = x_train + noise_std*np.random.randn(m_train, 32, 32)
x_test_noise = x_test + noise_std*np.random.randn(m_test, 32, 32)

# Acerta pixles com valores maiores do que 1 e menores do que 0
x_train_noise = np.clip(x_train_noise, 0.0, 1.0)
x_test_noise = np.clip(x_test_noise, 0.0, 1.0)
```

Gráficos de alguns exemplos de imagens originais e com ruído

In [ ]:
```
image_dim = (32, 32)

# Mostra imagens originais e reconstruídas
f, pos = plt.subplots(2, 10, figsize=(24, 5))
for i in range(10):
    pos[0,i].imshow(x_train[i], cmap='gray')
    pos[1,i].imshow(x_train_noise[i], cmap='gray')
plt.show()
```

- Observa-se que o ruído inserido nas imagens é significativo, mas ainda é possivel identificar os dígitos.

3.3 Adicão do eixo de cores nas imagens

As camadas convolucionais do Keras esperam receber imagens coloridas, ou seja, imagens com 3 eixos. Como as imagens de dígitos MNIST são em tons de cinza, devemos adicionar um quarto eixo aos tensores de imagens com dimensão 1.

In [ ]:
```
# Adinção do eixo de cores nas imagens
x_train_4 = np.expand_dims(x_train, axis=3)
x_test_4 = np.expand_dims(x_test, axis=3)
x_train_noise_4 = np.expand_dims(x_train_noise, axis=3)
x_test_noise_4 = np.expand_dims(x_test_noise, axis=3)

# Dimensão dos tensores de imagens
print('Dimensão do tensor de imagens de treinamento:', x_train_4.shape)
print('Dimensão do tensor de imagens de teste:', x_test_4.shape)
print('Dimensão do tensor de imagens com ruído de treinamento:', x_train_noise_4.shape)
print('Dimensão do tensor de imagens com ruído de teste:', x_test_noise_4.shape)
```

3.4 Configuração do autoencoder

Nesse exemplo vamos utilizar um autoencoder com camadas convolucionais e com um grande número de parâmetros em relação ao número total de pixels das imagens.

A arquitetura usada no codificador é a seguinte:

- Camada convolucional com 16 filtros, dimensão dos filtros=3x3, stride=2, função de ativação ReLu, padding=same;
- Camada convolucional com 24 filtros, dimensão dos filtros=3x3, stride=2, função de ativação ReLu, padding=same;
- Camada convolucional com 32 filtros, dimensão dos filtros=3x3, stride=2, função de ativação ReLu, padding=same.

A arquitetura usada no decodificador é a seguinte:

- Camada de deconvolução com 16 filtros, dimensão dos filtros=3x3, stride=2, função de ativação ReLu, padding=same;
- Camada de deconvolucão com 8 filtros, dimensão dos filtros=3x3, stride=2, função de ativação ReLu, padding=same
- Camada de deconvolucão com 4 filtros, dimensão dos filtros=3x3, stride=2, função de ativação ReLu, padding=same
- Camada convolucional com 1 filtro, dimensão dos filtros=1x1, stride=1, função de ativação sigmode, padding=same.

Note que o objetivo desse autoencoder é eliminar ruído de imagens e não obter um espaço latente compacto para representar as imagens. Assim, não é importante que o número de características do código seja muito menor do que o número de pixels das imagens originais.

In [ ]:
```
# Importa camadas necessárias
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Conv2DTranspose, BatchNormalization, LeakyReLU

# Configura codificador
encoder = Sequential()
encoder.add(Conv2D(16, kernel_size=3, strides=2, padding='same', activation=LeakyReLU(), input_shape=(
encoder.add(Conv2D(24, kernel_size=3, strides=2, padding='same', activation=LeakyReLU()))
encoder.add(Conv2D(32, kernel_size=3, strides=2, padding='same', activation=LeakyReLU()))

# Configura decodificador
decoder = Sequential()
decoder.add(Conv2DTranspose(16, kernel_size=3, strides=2, padding='same', activation='relu', input_sha
decoder.add(Conv2DTranspose(8, kernel_size=3, strides=2, padding='same', activation='relu'))
decoder.add(Conv2DTranspose(4, kernel_size=3, strides=2, padding='same', activation='relu'))
decoder.add(Conv2D(1, kernel_size=1, strides=1, padding='same', activation='sigmoid'))

# Configura autoencoder
autoencoder = Sequential([encoder, decoder])

# Sumario do codificador
print('Codificador:')
print(encoder.summary(),'\n\n')

# Sumario do decodificador
print('Decodificador:')
print(decoder.summary(),'\n\n')

# Sumario do autoencoder
print('Autoencoder:')
print(autoencoder.summary())
```

- A saída do codificar é um tensor de dimensão 4x4x32, que possui 512 elementos. Note que as imagens originais com dimensão 28x28 possuem 784 pixels, portanto, o espaço latente consiste de uma pequena redução de cerca de 1,5 vezes em relação às imagens originais.

3.5 Compilação do autoencoder

Vamos compilar o autoencoder com a seguinte configuração:

- Otimizador: Adam com taxa de aprendizado de 0.001;
- Função de custo: "binary cross entropy"
- Métrica: "binary_accuracy"

In [ ]:
```
# Define otimizador Adam
adam = tf.keras.optimizers.Adam(learning_rate=0.001)

# Compilação do autoencoder
autoencoder.compile(optimizer=adam, loss='binary_crossentropy', metrics=['binary_accuracy'])
```

3.6 Treinamento do autoencoder

Vamos treinar o autoencoder usando 30 épocas e lotes de 1024 elementos.

In [ ]:
```
results = autoencoder.fit(x_train_noise_4, x_train_4, epochs=30, batch_size=1024, validation_data=(x_t
```

In [ ]:
```python
def plot_train(history):
    history_dict = history.history

    # Salva custos, métricas e epocas em vetores
    custo = history_dict['loss']
    acc = history_dict['binary_accuracy']
    val_custo = history_dict['val_loss']
    val_acc = history_dict['val_binary_accuracy']

    # Cria vetor de épocas
    epocas = range(1, len(custo) + 1)

    # Gráfico dos valores de custo
    plt.plot(epocas, custo, 'b', label='Custo - treinamento')
    plt.plot(epocas, val_custo, 'r', label='Custo - validação')
    plt.title('Valor da função de custo – treinamento e validação')
    plt.xlabel('Épocas')
    plt.ylabel('Custo')
    plt.legend()
    plt.show()

    # Gráfico dos valores da métrica
    plt.plot(epocas, acc, 'b', label='exatidao- treinamento')
    plt.plot(epocas, val_acc, 'r', label='exatidao- validação')
    plt.title('Valor da métrica – treinamento e validação')
    plt.xlabel('Épocas')
    plt.ylabel('Exatidao')
    plt.legend()
    plt.show()

plot_train(results)
```

3.7 Avaliação do AE

Para avaliar o desempenho do AE vamos calcular o valor da função de custo e da métrica.

In [ ]:
```python
# Calcula função de custo e métrica
autoencoder.evaluate(x_train_noise_4, x_train_4)
autoencoder.evaluate(x_test_noise_4, x_test_4)
```

- Observa-se que a exatidão obtida é da ordem de 74% tanto para os dados de treinamento como de teste. Esse resultado não é muito bom, tendo em vista o fato do espaço latente ter quase o mesmo número de características do que as imagens originais tem de pixels. Porém, nota-se que a amplitude do ruído presente nas imagens de entrada é significativa.

3.8 Espaço latente das imagens

O espaço latente de cada imagem é um tensor de dimensão 4x4x32, assim, ele pode ser visto como sendo composto por 32 imagens de dimensão 4x4.

Vamos visualizar a representação latente de algumas imagens.

In [ ]:
```python
# Calcula códigos das imagens de teste
code = encoder.predict(x_test_noise_4)
print('Dimensão do espaço latente:', code.shape)

# Seleciona imagem do conjunto de teste
index = 0

# Mostra canais do espaço latente
cont = 0
f, pos = plt.subplots(4, 8, figsize=(24, 16))
for i in range(4):
    for j in range(8):
        pos[i,j].imshow(code[index,:,:,cont], cmap='gray')
        cont += 1
plt.show()
```

- Obviamente não é possível analisar esses dados, pois eles somente tem algum significado para o decodificador.
- Observa-se que cada imagem de dimensão 4x4 do espaço latente representa a presença (pixel mais claro) ou não (pixel mais escuro) de uma determinada característica em 16 regiões diferentes das imagens.

3.9 Comparação das saídas previstas pelo autoencoder com as imagens sem ruído

As saídas previstas pelo autoencoder representam como são reconstruídos os dados de entrada a partir da representação latente dos dados de entrada.

Vamos reconstruir as primeras 10 imagens de teste a partir das representações latentes calculadas no item anterior e visualizá-las junto com as imagens originais.

In [ ]:
```
# Reconstrução das imagens usando o espaço latente e o decodificador
x_prev = decoder.predict(code)
#x_prev = x_prev.astype(int)

# Mostra imagens originais e reconstruídas
f, pos = plt.subplots(3, 10, figsize=(24, 10))
for i in range(10):
    img_prev = np.reshape(x_prev[i], (32, 32))
    pos[0,i].imshow(np.squeeze(x_test_4[i]), cmap='gray')
    pos[1,i].imshow(np.squeeze(x_test_noise_4[i]), cmap='gray')
    pos[2,i].imshow(np.squeeze(img_prev), cmap='gray')
plt.show()
```

- Observe que as imagens reconstruídas são muito semelhantes às originais.
- Pode-se concluir que o autoencoder apresenta um desempenho muito bom mesmo com a grand eamplitude do reúid presente nas imagens.
- É interessante analisar a influência da amplitude do ruído na qualidade das imagens reconstruídas.

4. Detecção de anomalias

Neste exemplo, vamos treinar um autoencoder para detectar anomalias em sinais de eletrocardiograma usando o conjunto de dados ECG5000 (http://www.timeseriesclassification.com/description.php?Dataset=ECG5000 (http://www.timeseriesclassification.com/description.php?Dataset=ECG5000)). Este conjunto de dados contém 5.000 eletrocardiogramas, cada um com 140 amostras.

Nesse exemplo usaremos uma versão simplificada do conjunto de dados, onde cada exemplo é rotulado como `0` (correspondendo a um ritmo anormal) ou `1` (correspondendo a um ritmo normal). O objetivo é identificar os ritmos anormais.

Observação: esse conjunto de dados é rotulado, portanto, pode ser usado como um problema de aprendizado supervisionado. Porém, o objetivo deste exemplo é ilustrar o conceito de detecção de anomalias que pode ser aplicado a conjuntos de dados maiores, onde não há rótulos disponíveis (por exemplo, se tiver milhares de ritmos normais e apenas um pequeno número de ritmos anormais).

Um autencoder é treinado para minimizar o erro de reconstrução. Assim, se for desejado que o autoencoder detecte anomalias, então, deve-se treiná-lo apenas com eletrocardiogrmas normais e, utilizá-lo para reconstruir todos os dados. A hipótese é que os ritmos anormais terão maior erro de reconstrução. Dessa forma, é possível classificar um ritmo como uma anomalia se o erro de reconstrução ultrapassar um determinado limiar.

4.1 Carregar e processar dados

Os dados estão disponíveis no TensorFlow e estão em um CSV. Vamos utilizar o Pandas para carregar os dados.

In [ ]:
```
# Importa Pandas
import pandas as pd

# Carrega dados em um DataFrame
dataframe = pd.read_csv('http://storage.googleapis.com/download.tensorflow.org/data/ecg.csv', header=N

# Salva valores do DataFrame em tensir Numpy
raw_data = dataframe.values

# Mostra 5 primeiras linhas do DataFrame
dataframe.head()
```

- Observe que cada linha do DataFrame é um exame de eletrocardiograma com 140 amostras.
- A última coluna do DataFrame é a clasisficação do exame.

Vamos analisar a estatísticas dos dados.

In [ ]:
```
dataframe.describe()
```

Separação das saídas das entradas e divisão dos dados em conjuntos de treinamento e validação.

In [ ]:
```
# Importa função para dividir dados
from sklearn.model_selection import train_test_split

# Separa as saídas
labels = raw_data[:, -1]

# Separa as entrdadas
data = raw_data[:, 0:-1]

# Divide dados em conjuntos de treinamento e validação
x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size=0.2, random_state=21)

print('Dimensão do tensor de dados de entrada de treinamento:', x_train.shape)
print('Dimensão do tensor de dados de entrada de teste:', x_test.shape)
print('Dimensão do tensor de dados de saída de treinamento:', y_train.shape)
print('Dimensão do tensor de dados de saída de treinamento:', y_test.shape)
```

Normalização dos dados no intervalo `[0,1]` .

In [ ]:
```
# Calcula valores mínimos e máximos de cada coluna
min_val = tf.reduce_min(train_data)
max_val = tf.reduce_max(train_data)

# Normalizaçãa das entradas
x_train = (x_train- min_val) / (max_val - min_val)
x_test = (x_test - min_val) / (max_val - min_val)

# Transformação em números reais
x_train = tf.cast(x_train, tf.float32)
x_test = tf.cast(x_test, tf.float32)
```

Vamos treinar o autoencoder usando somente os eletrocardiogramas normais, que são rotulados com saída igual a 1. Dessa forma temos que separa os exames normais dos com problema.

In [ ]:
```
# Transforma saídas em variáveis bolleanas
y_train = y_train.astype(bool)
y_test = y_test.astype(bool)

# Dados de entrada com exames normais
x_train_normal = x_train[y_train]
x_test_normal = x_test[y_test]

# Dados de entrada com exames com problemas
x_train_anormal = x_train[~y_train]
x_test_anormal = x_test[~y_test]
```

Gráfico de ECG normal.

In [ ]:
```
plt.grid()
plt.plot(np.arange(140), x_train_normal[0], 'b')
plt.plot(np.arange(140), x_train_normal[10], 'r')
plt.title("ECG normal")
plt.show()
```

GRáfico de ECG anormal.

In [ ]:
```
plt.grid()
plt.plot(np.arange(140), x_train_anormal[0], 'b')
plt.plot(np.arange(140), x_train_anormal[20], 'r')
plt.title("ECG anormal")
plt.show()
```

4.2 Configuração do autoencoder

Vamos usar camadas densas e modelos sequenciais para configurar o codificador, o decoficador e o autoencoder.

In [ ]:
```python
# Importa classe de modelos e camadas
from tensorflow.keras import layers
from tensorflow.keras import models

# Define dimensões dos dados e do espaço latente
ecg_dim = x_train.shape[1]
latent_dim = 8

# Configura encoder
encoder = models.Sequential()
encoder.add(layers.Dense(32, activation="relu", input_shape=(ecg_dim,)))
encoder.add(layers.Dense(16, activation="relu"))
encoder.add(layers.Dense(8, activation="relu"))
print('Codificador')
encoder.summary()

# Configura decoder
print('\nDecodificar')
decoder = models.Sequential()
decoder.add(layers.Dense(16, activation="relu", input_shape=(latent_dim,)))
decoder.add(layers.Dense(32, activation="relu"))
decoder.add(layers.Dense(140, activation="sigmoid"))
decoder.summary()

# Configura autoencoder
print('\nAutoencoder')
autoencoder = models.Sequential([encoder, decoder])
autoencoder.summary()
```

4.3 Compilação e treinamento do autoencoder

Para compilar o autoencoder, vamos usar o método de otimização Adam com taxa de aprendizagem 0,001, a função de custo erro quadrático médio (`mae`) e a métrica erro absoluto médio (`mae`).

In [ ]:
```python
autoencoder.compile(optimizer='adam', loss='mse', metrics=['mae'])
```

Para treinar o autoencoder vamos usar 200 épocas e um lote de 256 exemplos.

Note que o autoencoder é treinado usando somente os exames normais, mas é avaliado usando todos os dados.

Notice that the autoencoder is trained using only the normal ECGs, but is evaluated using the full test set.

In [ ]:
```python
results = autoencoder.fit(x_train_normal, x_train_normal,
          epochs=200,
          batch_size=512,
          validation_data=(x_test_normal, x_test_normal))
```

Gráfico do processo de treinamento.

In [ ]:
```python
history = results.history

plt.plot(history["loss"], 'b', label="Custo treinamento")
plt.plot(history["val_loss"], 'r', label="Custo valdação")
plt.legend()
plt.show()

plt.plot(history["mae"], 'b', label="Métrica treinamento")
plt.plot(history["val_mae"], 'r', label="Métrica valdação")
plt.legend()
plt.show()
```

- Observe que não aparece problema de overfitting no treinamento.
- O erro de absoluto médio é de cerca de 0.013, o que é um bom resultado. Porém, como não está ocorrende overfitting poderia testar dimensões maiores para o espaço latente para reduzir esse erro.

4.4 Avaliação e teste

Para avaliar o autoencoder vamos fazer o gráfico de exames normais/anormais originais, das suas reconstruções pelo autoencoder e do erro de reconstrução

Gráficos de exames normais.

In [ ]:
```
# Reconstrução de exames normais
code = encoder(x_test_normal).numpy()
exames_rec = decoder(code).numpy()

# Mostra exames normais originais, reconstruídos e erro
print('Exames normais')
f, pos = plt.subplots(4, 1, figsize=(10, 16))
for i in range(4):
    pos[i].plot(x_test_normal[i], 'b')
    pos[i].plot(exames_rec[i], 'r')
    pos[i].fill_between(np.arange(140), exames_rec[i], x_test_normal[i], color='lightcoral' )
    pos[i].legend(labels=["Exame normal", "Reconstrução", "Erro"])
plt.show()
```

Gráficos de exames com problemas.

In [ ]:
```
code = encoder(x_test_anormal).numpy()
exames_rec = decoder(code).numpy()

# Mostra exames normais originais, reconstruídos e erro
print('Exames com problemas')
f, pos = plt.subplots(4, 1, figsize=(10, 16))
for i in range(4):
    pos[i].plot(x_test_anormal[i], 'b')
    pos[i].plot(exames_rec[i], 'r')
    pos[i].fill_between(np.arange(140), exames_rec[i], x_test_anormal[i], color='lightcoral' )
    pos[i].legend(labels=["Exame normal", "Reconstrução", "Erro"])
plt.show()
```

- Observe que o erro de reconstrução é bem maior para os casos dos exames com problemas.

4.5 Deteção de anomalias

Anomalias nos exames são detectadas se o erro de reconstrução é maior que um determinado limiar.

Vamos usar o erro absoluto médio para calcular o erro de reconstrução e vamos considerar exames como sendo anormais se o erro de reconstrução for maior do que um desvio padrão do conjunto de treinamento.

Cálculo do erro de reconstrução dos exames normais

In [ ]:
```
# Reconstrução dos exames normais
exames_normais_rec = autoencoder.predict(x_train_normal)

# Cálculo do erro de reconstrução dos exames normais
erro_rec = tf.keras.losses.mae(exames_normais_rec, x_train_normal)

# Gráfico do hstograma do erro de reconstrução
plt.title('Erro de reconstrução - exames normais')
plt.hist(erro_rec, bins=50)
plt.xlabel("Erro de reconstrução")
plt.ylabel("Númeor de exemplos")
plt.show()
```

Cálculo do limiar de normalidade

Vamos calcular o desvio padrão do erro de reconstrução dos exames normais para usar como limiar para identificar anomalias.

In [ ]:
```
# Calculo do desvio padrão
std = np.std(erro_rec)

# Limiar de normalidade
limiar = np.mean(erro_rec) + std
print("Limiar: ", limiar)
```

Oberve que existem outras estratégias que podem ser usadas para selecionar um valor limite acima do qual os exeames devem ser classificados como anormais → a abordagem adequada depende do conjunto de dados.

Se forem examinados os erros de reconstrução para os exemplos anormais do conjunto de teste, é possível notar que a maioria tem um erro de reconstrução maior do que o limiar adotado.

Observa-se que Variando o limiar é possível ajustar a "precisão" e a "revocação" do detector de anomalias.

Cálculo do erro de reconstrução dos exames anormais

In [ ]:
```
# Reconstrução dos exames normais
exames_anormais_rec = autoencoder.predict(x_train_anormal)

# Cálculo do erro de reconstrução dos exames anormais
erro_rec = tf.keras.losses.mae(exames_anormais_rec, x_train_anormal)

# Gráfico do hstograma do erro de reconstrução
plt.hist(erro_rec, bins=50)
plt.title('Erro de reconstrução - exames anormais')
plt.xlabel("Erro de reconstrução")
plt.ylabel("Número de exemplos")
plt.show()
```

- Observe que o erro de reconstrução de particamente todo os exames anormais é maior do que o limiar calculado de 0.01956.

Identificação de anomalias

Para identificar anomalias vamos criar uma função (`diagnostico()`) que calcula o erro de reconstrução e o compara com o limiar e retorna o diagnóstico.

In [ ]:
```
# Importa funções da biblioteca SkLearn
from sklearn.metrics import accuracy_score, precision_score, recall_score

# Função para diagnosticar anomalia em exame ECG
def diagnostico(model, data, limiar):
    reconstrucao = model(data)
    erro = tf.keras.losses.mae(reconstrucao, data)
    return tf.math.less(erro, limiar)

# Função para mostrar resultados
def print_stats(prevs, labels):
    print("Exatidão = {}".format(accuracy_score(labels, prevs)))
    print("Precisão = {}".format(precision_score(labels, prevs)))
    print("Revocação = {}".format(recall_score(labels, prevs)))
```

In [ ]:
```
prevs = diagnostico(autoencoder, x_test, limiar)
print_stats(prevs, y_test)
```

- A função `tf.math.less(x, y)` , retorna `False` se `y >= x` e `True` se `y < x` .
- As funções `accuracy_score()` , `precision_score()` e `recall_score()` da bilbioteca Sklearn calculam a exatidão, a precisão e a revocação entre dois conjuntos de dados.

5. Conclusão

Como vimos, os autoencoders são muito eficientes para realizar alguns tipos de tarefas, tais como: compressão de dados, eliminação de ruídos, restauração de dados, detecção de anomalias e até geração de novos dados de forma limitada.

Um problema dos autoencoders que faz com eles sejam ruins para gerar novos exemplos é que o espaço latente criado por eles contém lacunas e não é possível saber como são os dados gerados nessas lacunas. Isso equivale à falta de dados em um problema de aprendizado supervisionado, pois o modelo não é treinado para essas regiões do espaço latente.

Outro problema dos autoencoders é a separabilidade dentre as regiões do espaço latente que representam cada classe, várias classes ficam bem separadas, mas também existem regiões onde as classes se misturam aleatoriamente, tornando difícil separá-las.

Esses problemas com os autoencoders são resolvidos nos autoencoders variacionais, que aprendem a distribuição de probabilidade dos dados usados no treinamento.

In [ ]:

```
1
```