Aula 7

Autoencoders - Parte 1 ¶

Eduardo Lobo Lustosa Cabral

1. Objetivos

Definir conceito de espaço latente.

Apresentar o que é um autoencoder.

Apresentar como criar autoencoders com o TensorFlow.

Criar um autoencoder com camadas densas.

Criar um autoencoder com camadas convolucionais.

Importação das bilbiotecas necessárias

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
print(tf.__version__)
```

```
2.4.1
```

2. Introdução

Antes de iniciar a formulação dos modelos generativos e mais precisamente nessa aula os Autoencoders, vamos relembrar o que são modelos generativos.

RNAs deep learning são capazes de produzir imagens, textos e músicas artificiais. Essa capacidade das RNAs têm muitas aplicações promissoras em negócios e arte.

Existem dois tipos de modelos generativos baseados em RNAs:

1. Autoencoders variacionais (AEV)
2. Redes adversárias generativas (GANs)

Existem atualmente muitas aplicações de modelos generativos em pesquisa, negócios e artes.

A longo prazo os modelos generativos consitem de uma grande promessa da IA para se tornarem sistemas automáticos que aprenderem sem supervisão características de um conjunto de dados, ou seja, eles tem o potencial de serem sistemas que aprendem por conta própria.

Observa-se que um autoencoder comum, como o que veremos nessa aula, não é de fato considerado um modelo generativo, mas eles são capazes de gerar novos dados, porém de uma forma limitada.

Os modelos generativos usam uma representação dos dados chamada de espaço latente (ou representação latente) para produzir novos dados. Assim, a primeira etapa para um modelo produzir novos dados é aprender o espaço latente do que é desejado gerar.

3. Espaço latente

A palavra “latente” significa “escondido”. Espaço latente é um termo muito usado em IA. Um espaço latente consiste de uma transfromação dos dados originais, onde exemplos de dados semelhantes ficam "próximos" entre sí e exemplos de dados diferentes ficam "distantes".

Espaço latente se refere a um espaço multidimensional abstrato que contém valores de características de dados que não podemos interpretar diretamente, mas que codifica uma representação interna de eventos observados externamente.

Assim como as pessoas possuem compreensão de uma ampla gama de tópicos e dos eventos pertencentes a esses tópicos, o espaço latente visa fornecer uma compreensão semelhante para um computador por meio de uma representação quantitativa.

Uma RNA para realizar uma dada tarefa (classificação, regressão, reconstrução de imagem) primeiramente extrai caracteríticas dos dados usando suas diversas camadas. Dizemos que a função que mapeia a entrada para as camadas projeta os dados no espaço latente. Em outras palavras, o espaço latente é o espaço onde estão as características dos dados.

Uma das motivações para aprender um espaço latente de um certo tipo de dado é que grandes diferenças no espaço dos dados podem ser ocasionadas por pequenas variações no espaço latente. Portanto, aprender um espaço latente ajuda um modelo a entender melhor os dados do que os próprios dados observados, que é um espaço muito grande para aprender.

Alguns exemplos de espaço latente são:

1) Embedding de palavras ("word embedding"): consiste em vetores que representam palavras, onde palavras semelhantes em significado são representadas por vetores que ficam próximos uns dos outros no espaço (conforme medido por exemplo pela distância euclidiana) e palavras que não estão relacionadas ficam distantes.

2) Espaço de caracteríticas de imagens: as camadas finais das redes convolucionais codificam características de alto nível que permitem detectar efetivamente o que uma imagem mostra, por exemplo, a presença de um gato em uma imagem em diferentes condições de iluminação e ângulos de observação, o que é uma tarefa difícil no espaço dos pixels da imagem.

3) Métodos de modelagem de tópicos (LDA, PLSA) capazes de extrair significados de textos, usam abordagens estatísticas para obter um conjunto latente de tópicos contidos em um conjunto de documentos. Na extração de significado de textos, o modelo produz um espaço latente de forma que documentos pertencentes a tópicos semelhantes fiquem mais próximos no espaço latente de tópicos.

Por exemplo, considere as 4 imagens da Figura 1.

Figura 1 - Exemplos de imagens de cadeiras e mesas.

No espaço de pixels que as pessoas enxergam, não há semelhança imediata entre as duas imagens de cadeiras e de mesas. No entanto, se essas imagens fossem mapeadas para um espaço latente, as imagens das cadeiras vão estar mais próximas entre elas do que entre as imagens de cadeiras e de mesas e o mesmo acontece com as imagens das mesas. O espaço latente captura a essência da estrutura dos dados para realizar uma dada tarefa.

Um espaço latente pode ter uma pequena ou uma grande dimensão em relação à dimensão dos dados originais. Os termos "grande dimensão" e "pequena dimensão" de um espaço latente ajuda a definir quão específicos ou gerais são os tipos de características que desejadas que o espaço latente aprenda e represente. Um espaço latente de grande dimensão é sensível a características mais específicas dos dados de entrada e às vezes pode levar ao sobreajuste quando não há dados de treinamento suficientes. Um espaço latente de pequena dimensão visa capturar as características mais importantes necessárias para aprender e representar os dados de entrada.

4. Autoencoders

Um autoencoder é uma RNA capaz de aprender representar dados sem supervisão, ou seja, realiza uma tarefa de aprendizado não supervisionado onde os dados de treinamento não são rotulados.

O conceito de um autoencoder é muito similar à ideia de compactação de informação. No caso de uma imagem, o autoencoder gera uma matriz (ou um vetor) capaz de representar a imagem original, sem muita perda de informação, com uma dimensão menor do que a imagem original.

Os autoencoders podem ser usados para redução de dimensão de dados de forma a facilitar visualização. Por exemplo, dados de 4 dimensões podem ser reduzidos para 2 ou 3 dimensões, permitindo a sua vizualização e obviamente a sua compactação.

Os autoencoders podem ser usados para criar novos dados similares aos dados originais, permitindo gerar novos exemplos para serem usados, por exemplo, em treinamento supervisionado de RNAs.

Os autoencoders consistem de uma forma eficiente de representar dados. Um autoencoder reproduz os dados de entrada na sua saída e ao fazer isso aprende como representar os dados de entrada.

O objetivo de um autoencoder é criar uma representação latente dos dados. Usando essa representação latente é possível reconstruir os dados originais e/ou produzir novos exemplos de dados.

4.1 Aplicações de autoencoders

A remoção de ruído de dados e a redução de dimensionalidade para visualização de dados são as duas principais aplicações dos autoencoders. Nota-se que os autoencoders podem aprender projeções de dados melhor do que um PCA. [11]

Alguns exemplos de aplicação de autoencoders são descritos a seguir.

Compactação de dados

Por exemplo, uma imagem de dimensão 256x256 pixels pode ser representada por 28x28 pixels. O Google está usando esse tipo codificação para permitir uma utilização mais otimizada dos sistemas de transmissão de dados. Nesse caso, somente a representação latente da imagem é transmitida e, em seguida, no telefone/computador do usuário o decodificador reconstrói a imagem.

Processamento de linguagem antural

No processamento natural de linguagem usando deep learning os autoencoders são usados com diversas finalidades. [9].

- Codificação "embedding" de palavras, frases, ou contexto de uma palavra em um documento.
- Agrupamento (classificação) de documentos por categorias (assuntos). Hinton [10] usou um autoencoder para reduzir os vetores "embedding" de forma a representar as probabilidades de palavras em notícias.
- Autoencoders são usados em sistemas de codificação de voz para criar espectrogramas se sinais de voz [12].
- Autoencoders são usados para eliminação de ruído em sinais de voz aumentando o desempenho de sistemas de reconhecimento automático de fala.
- É comum ter duas frases que parecem ser diferentes, mas ambas têm o mesmo significado. Autoencoders são usados para detectar essas frases com precisão.

Processamento de imagens

Autoencoders tem muitas aplicações em processamento de imagens. Por exemplo, são usados na reconstrução de imagens com partes ausentes, para eliminar ruído de imagens, para alterar a coloração de imagens e gerar imagens de alta resolução.

Autoencoder para detecção de anomalias em vídeo

A detecção de eventos anômalos em cenas de vídeo é um problema desafiador devido à complexidade da “anomalia”, bem como a presença de fundos complexos, objetos e movimentos desordenados nas cenas. Zhao, Deng e Shen (2018), propuseram um modelo chamado Spatio-Temporal AutoEncoder, que utiliza redes neurais para aprender a representação de vídeo automaticamente e extrair caracteríticas espaciais e temporais usando convoluções tridimensionais.

4.2 Comparação entre autoencoders e PCA

Os autoencoders, assim como o PCA, são usados para obter uma representação compacta dos dados. Porém os autoencoders apresentam diversas vantagens em relação ao PCA nesse tipo de tarefa, tais como:

- Treinar um autoencoder composto por um codificador com uma camada densa e um decodificador com uma camada densa e função de ativação linear é essencialmente equivalente a executar PCA. Porém, um autoencoder pode aprender transformações não lineares, ao contrário do PCA, se for usada uma função de ativação não linear e múltiplas camadas.
- Um autoencoder pode usar camadas convolucionais para aprender de forma mais eficiente dados de vídeos, imagens e séries temporais.
- As diversas camadas de um autoencoder podem fornecer representações latentes de dimensões diferentes.

- Um autoencoder permite usar modelos pré-treinados, para transferência de aprendizado do codificador e decodificador, facilitanto o seu treinamento.

4.3 Arquitetura dos autoencoders

Autoencoders consistem de um tipo específico de redes neurais onde a entrada é igual à saída. Eles compactam a entrada em um código com dimensão inferior à da entrada e, em seguida, reconstroem a saída usando essa representação.

O código gerado pelo Autoencoder é uma versão “compactada” da entrada, também chamada de representação do espaço latente.

Um autoencoder é composto por três componentes: 1) codificador; 2) código; e 3) decodificador. O codificador compacta a entrada e produz o código, o decodificador reconstrói a entrada usando apenas este código. Na Figura 2 é apresentado um esquema de um autoencoder.

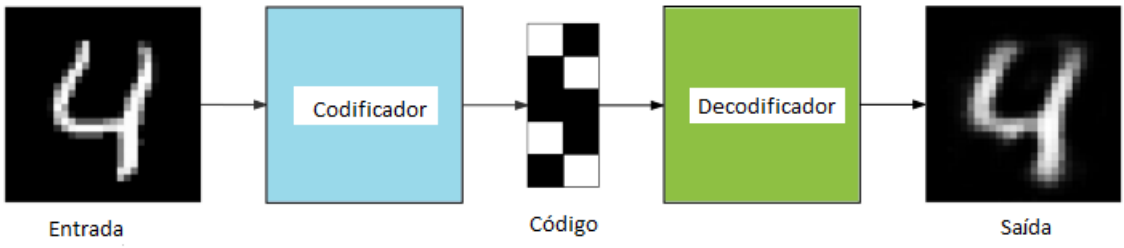


Figura 2 - Esquema simplificado de um autoencoder.

Para construir um autoencoder, precisamos de três elementos: 1) um método de codificação; 2) um método de decodificação; e 3) uma função de custo para comparar a saída prevista pelo modelo com a entrada e, assim, permitir o aprendizado por meio da minimização dessa função de custo.

Autoencoders consistem principalmente de um algoritmo de redução (ou compressão) de dimensionalidade com algumas propriedades importantes:

- Especificidade aos dados: os autoencoders só são capazes de compactar dados de maneira significativa quando as entradas forem semelhantes aos dados usados no treinamento. Como eles aprendem caracteríticas específicas dos dados de treinamento fornecidos, eles são diferentes de algoritmos de compactação de dados padrão, tal como, "jpeg" para imagens. Portanto, não podemos esperar que um autoencoder treinado em dígitos manuscritos seja capaz de compactar fotos de paisagens.

- Apresentam perdas: a saída do autoencoder não é exatamente igual à entrada, mas sim uma representação próxima degradada. Se for desejado compactação sem perdas, os autoencoders não podem ser usados.

- Treinamento não supervisionado: para treinar um autoencoder não é necessário nada muito complexo, basta apenas usar os dados de entrada como entrada e como saída. Os autoencoders são treinados usando aprendizagem não supervisionada, pois não precisam de rótulos explícitos para treinar. Para ser mais preciso, eles são auto supervisionados porque geram seus próprios rótulos a partir dos dados de treinamento.

Um autoencoder pode possuir qualquer tipo de camada, dependendo do tipo de dados e do problema que se deseja resolver. Em um autoencoder com camadas densas, tanto o codificador quanto o decodificador são redes neurais feedforward totalmente conectadas. O código (saída do codificador) consiste na saída de uma única camada da RNA com o número de neurônios de nossa escolha. O número de neurônios no código é um hiperparâmetro que é definido antes de treinar o autoencoder.

Na Figura 3 é mostrada uma arquitetura mais geral de um autoencoder com camadas densas. A entrada é processada pelo codificador, para produzir o código. O código é produzido no "gargalo" do autoencoder. O gargalo é a camada do autoencoder que possui o menor número de ativações e consiste na última camada do codificador. O decodificador, que tem uma estrutura semelhante ao codificador, gera a saída usando o código. O objetivo é obter uma saída idêntica à entrada. Observe que a arquitetura do decodificador é a imagem espelhada do codificador. Este não é um requisito, mas normalmente é o caso. O único requisito é a dimensionalidade da entrada e da saída que precisam ser a mesma.

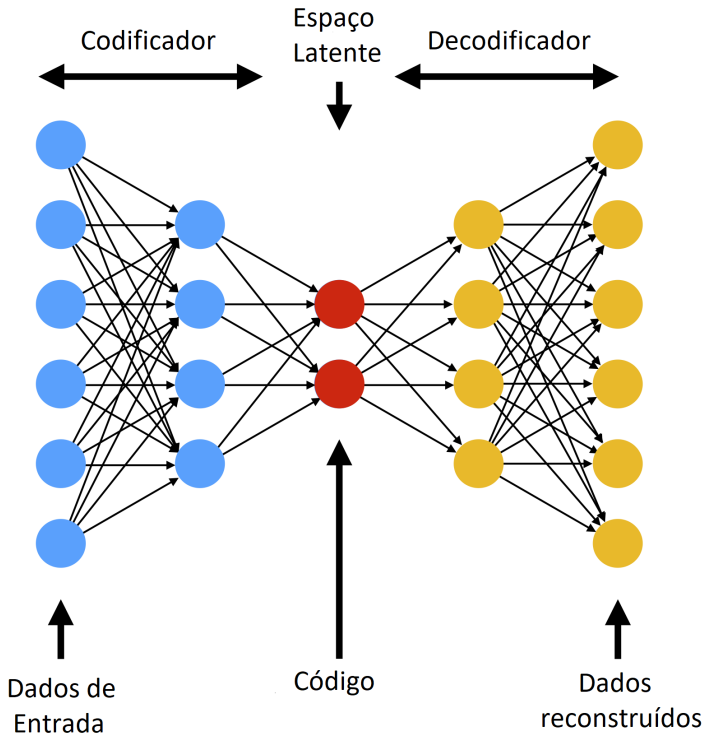


Figura 3 - Esquema greal de um autoencoder.

Existem quatro hiperparâmetros que precisam ser definidos antes de treinar um autoencoder:

- Dimensão do código: número de elementos da representação latente, quanto menor a dimensão, maior a compressão.
- Número de camadas: o autoencoder pode ter quantas camadas for desejado. Na Figura 3 acima, temos duas camadas no codificador e duas no decodificador.
- Número de neurônios por camada: normalmente, os autoencoders parecem ser um “sandwitch”. O número de neurônios por camada diminui com cada camada subsequente do codificador e aumenta de volta no decodificador. Além disso, o decodificador é simétrico ao codificador em termos de estrutura de camadas. Conforme já observado, isso não é necessário e temos controle total sobre esses parâmetros.
- Função de custo: usamos o erro quadrático médio (mse) ou a entropia cruzada binária. Se os valores de entrada estiverem no intervalo [0, 1], normalmente usamos a entropia binária cruzada, caso contrário, usamos o erro quadrático médio.

4.4 Matemática dos autoencoders

A matemática por trás dos autoencoders é bastante simples e fácil de entender.

Como visto, dividimos o autoencoder em duas partes, o codificador e o decodificador.

- Codificador: $\phi : X \rightarrow F$
- Decodificaor: $\psi : F \rightarrow X$

A função do codificador, denotado por ϕ, é mapear os dados originais X, para um espaço latente F, que é gerado no "gargalo" do autoencoder. A função do decodificador, denotado por ψ, é mapear o espaço latente F para a saída. A saída, neste caso, é igual à entrada. Assim, basicamente estamos tentando recriar os dados originais após ser realizada uma compressão não linear nos mesmos. Portanto, o objetivo de treinar um autoencoder é minimizar o erro entre a entrada e a saída reconstruída pelo decodificador. Esse objetivo pode ser expresso matematicamente por:

$$\phi, \psi = \min_{\phi,\psi} Erro[X - (\phi o \psi)(X)]$$

onde $(\phi o \psi)$ representa a função implemenetada pelo autoencoder, que é composição do codificador e decodificador, e $Erro$ é uma determinada função de custo.

Tendo em vista que o treino de um autoencoder tem a mesma função do treino de qualquer RNA, então, os autoencoders são treinados da mesma maneira que as RANs via retropropagação.

5. Autoencoder simples

Na Figura 4 é mostrado um autoencoder simples. Nesse exemplo deseja-se construir um autoencoder para representar pontos no espaço 3D em um espaço 2D.

A entrada do autoencoder consiste de 3 elementos que consistem nas coordenadas, x, y e z de um ponto no espaço. O código, ou representação latente dos pontos é um vetor 2D, que pode ser visualizado de forma simples em um plano. A Saída do autoencoder desejada é a reconstrução dos pontos no espaço 3D.

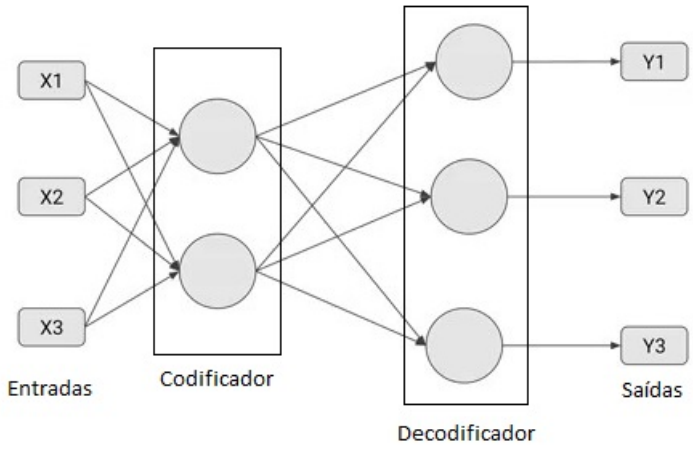


Figura 4 - Esquema de um autoencoder para codificar pontos no espaço 3D em 2D.

Na codificação dos pontos 3D em 2D perde-se uma dimensão, assim, a reconstrução dos pontos pelo codificador nunca será exata.

A saída do codificador (saída da camada com 2 neurônios) representa os dados codificados em 2D.

A saída do decodificador (saída da RNA) representa os dados reconstruídos em 3D a partir dos códigos (espaço latente) gerados pelo codificador.

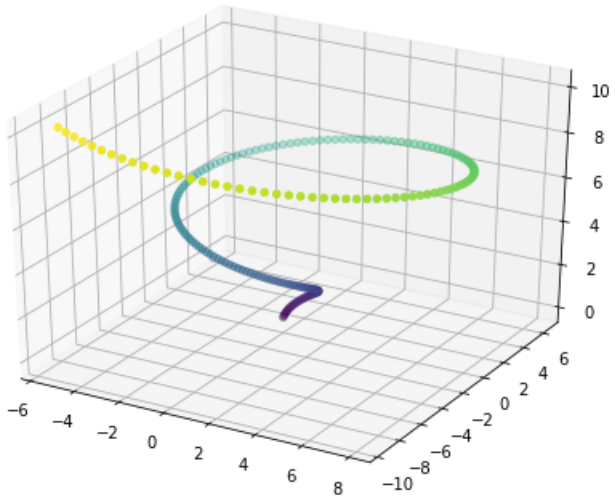
Observa-se que nesse autoencoder não se pode ter no decodificador o mesmo número de neurônios do codificador, pois teria um passe direto dos dados de entrada para a saída e nada seria aprendido.

5.1 Dados de entrada

Na célula abaixo é criado um conjunto de pontos no formato de um círculo em 3D, onde a coordenada z aumenta com a coordenada y.

In [2]:
```
# Importa funções para fazer gráficos 3D
from mpl_toolkits import mplot3d

# Número de exemplos de treinamento
n = 200

# Geração dos dados de treinamento
zdata = np.linspace(0, 10, 200)# 10 * np.random.random(n)
xdata = zdata*np.sin(zdata) + 0.0 * np.random.randn(n)
ydata = zdata*np.cos(zdata) + 0.0 * np.random.randn(n)

# Visualização dos pointos em 3Dr
fig = plt.figure(figsize=(8,6))
ax = plt.axes(projection='3d')
ax.scatter3D(xdata, ydata, zdata, c=zdata)
```

Out[2]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7f9276039350>
```



Os exemplos de treinamento são obtidos combinando as coordenadas x, y e z dos pontos.

In [3]:
```
x = np.reshape(xdata, (n,1))
y = np.reshape(ydata, (n,1))
z = np.reshape(zdata, (n,1))

x_train = np.concatenate([x, y, z], axis=1)
print('Dimensão dos dados de treinamento:', x_train.shape)
```

```
Dimensão dos dados de treinamento: (200, 3)
```

- Observa-se que temos 500 exemplos de treinamento.

5.2 Codificação do autoencoder

Nesse exemplo, tanto o codificador como o decoficador são modelos sequenciais, cada um com uma única camada densa.

O autoencoder é criado pela combinação do codificador com o decodificador.

In [4]:
```
# Importa classes do Keras
from tensorflow.keras import models
from tensorflow.keras import layers

# Configura codificador
encoder = models.Sequential([layers.Dense(2, activation='linear', input_shape=(3,))])

# Configura decodificador
decoder = models.Sequential([layers.Dense(3, activation='linear', input_shape=(2,))])

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

# Sumário do AE
autoencoder.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential (Sequential)      (None, 2)                 8
_________________________________________________________________
sequential_1 (Sequential)    (None, 3)                 9
=================================================================
Total params: 17
Trainable params: 17
Non-trainable params: 0
_________________________________________________________________
```

- Observe que o coficador e o decodificador devem ser criados separadamente se quisermos gerar as representações latentes dos dados de entrada.
- Nesse exemplo foram usadas funções de ativação linear em ambas as camadas. Podemos usar outras funções de ativação, mas isso poderia exigir alguma normalização dos dados e entrada.

5.3 Compilação do autoencoder

Vamos compilar o autoencoder usando a seguinte configuração:

- Otimizador: RMSprop
- Função de custo: MSE
- Métrica: MAE

In [5]:
```
# Importa classe do otimizadores
from tensorflow.keras import optimizers

# Instância otimizador
rms = optimizers.RMSprop()

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

5.4 Treinamento do autoencoder

O treinamento do autoencoder é realizado como qualquer RNA usando o método `fit` .

Basta treinar o autoencoder completo que tanto o codificador como o decodificador isolados são também treinados, pois o autoencoder é formado por ambos.

Observe que a saída desejada é igual aos dados de entrada.

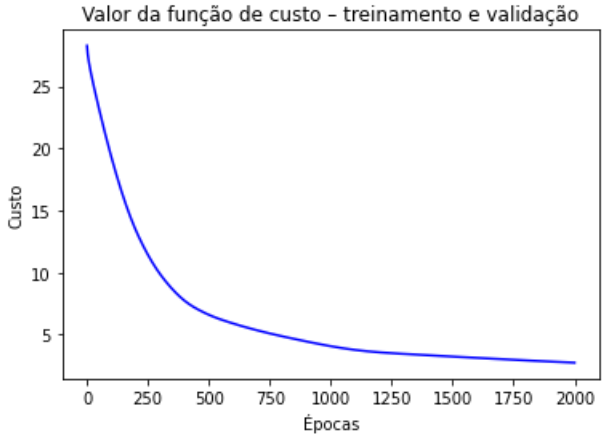
O treinamento do autoencoder é realizado usando 2000 épocas com um único lote de treinamento.

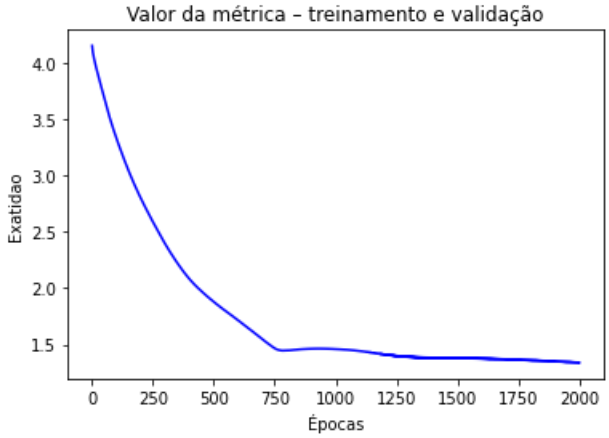
In [6]:
```
results = autoencoder.fit(x_train, x_train, batch_size=n, epochs=2000, verbose=0)
```

Resultados do processo de treinamento.

In [7]:

```python
# Define função para fazer gráficos
def plot_train(history):
    history_dict = history.history

    # Salva custos, métricas em vetores
    custo = history_dict['loss']
    mae = 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 – treinamento e validação')
    plt.xlabel('Épocas')
    plt.ylabel('Custo')
    plt.show()

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

plot_train(results)
```





5.5 Avaliação do autoencoder

Para avaliar o desempenho do autoencoder vamos calcular a função de custo e a métrica.

In [8]:
```
# Calcula função de custo e métrica
autoencoder.evaluate(x_train, x_train)
```

```
7/7 [==============================] - 0s 4ms/step - loss: 2.7355 - mae: 1.3398
```

Out[8]:
```
[2.735527992248535, 1.3397712707519531]
```

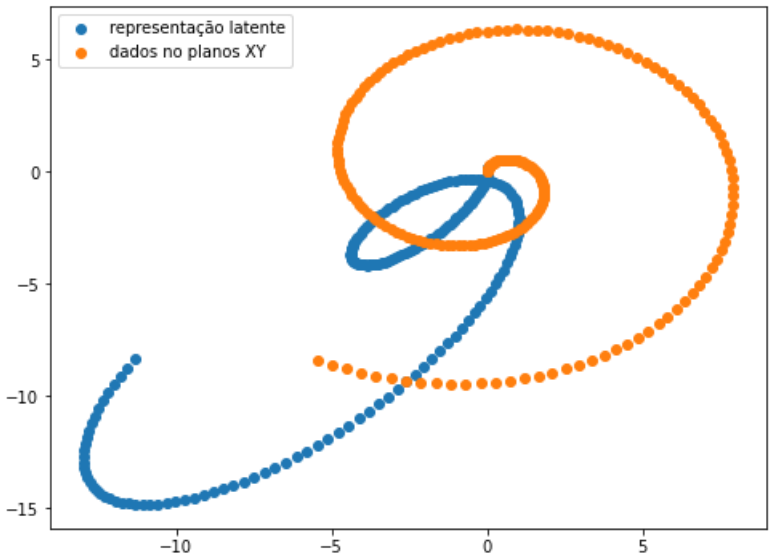
- Observa-se que tanto o erro quadrático médio quanto o erro absoluto médio são relativamente grandes, o que parece não ser um bom resultado.

5.6 Representação do espaço latente

Para reconstruir os dados a partir do espaço latente devemos obter os códigos que representam os dados no espaço latente. Os códigos são calculados pelo codificador tendo como entrada pontos no espaço 3D.

In [9]:
```
# Calcula espaço latente (saída do codificador)
code = encoder.predict(x_train)
print('Dimensão do espaço latente:', code.shape)

# Gráfico dos resultados
plt.figure(figsize=(8,6))
plt.scatter(code[:,0], code[:,1], label='representação latente')
plt.scatter(xdata, ydata, label='dados no planos XY')
plt.legend()
plt.show()
```

```
Dimensão do espaço latente: (200, 2)
```



- Observa-se que é difícil analisar esses dados, pois eles somente tem algum significado para o decodificador.

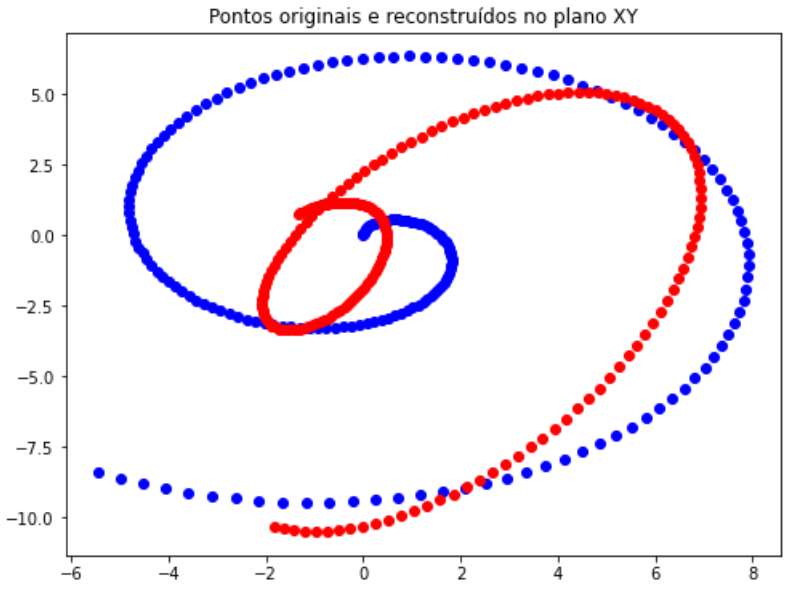
5.7 Comparação das saídas previstas pelo autoencoder com as entradas

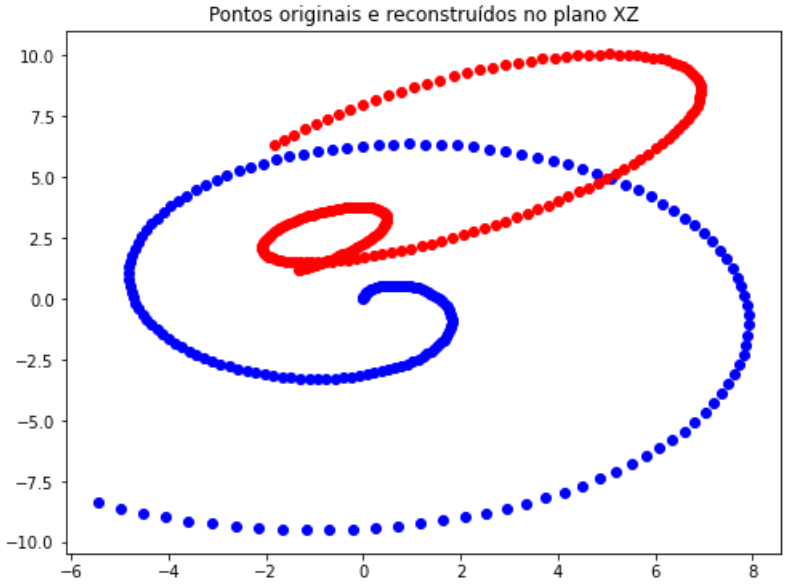
As saídas previstas pelo autoencoder representam com são reconstruídos os dados de entrada a partir da representação latente dos dados de entrada.

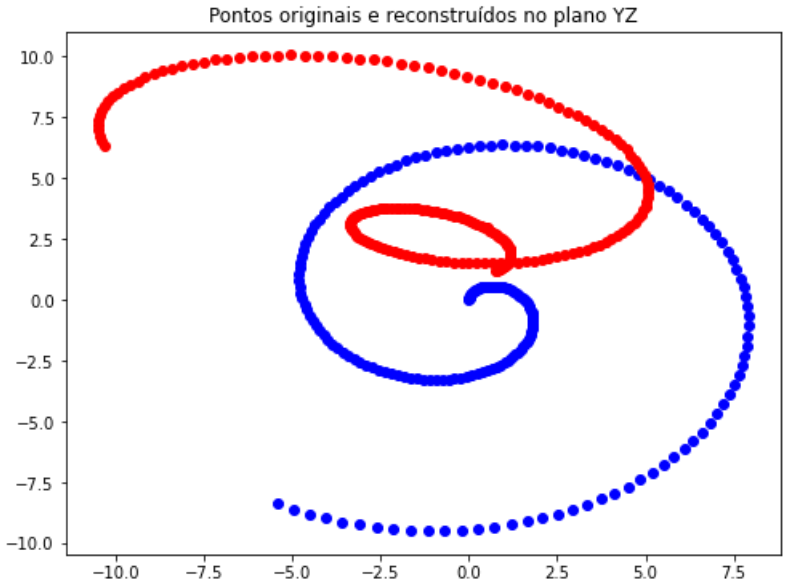
Para poder comparar os dados reconstruídos com os originais, vamos visualizar as projeções dos dados de entrada e reconstruídos nos planos XY, XZ e YZ.

In [10]:

```
# Calcula dados reconstriuídos pelo AE
x_prev = decoder.predict(code)

# Visualização dos pointos nos planos XY
fig = plt.figure(figsize=(8,6))
c = ['b', 'r']
plt.scatter(xdata, ydata, c=c[0])#, label='Dados reais')
plt.scatter(x_prev[:,0], x_prev[:,1], c=c[1])
plt.title('Pontos originais e reconstruídos no plano XY')
plt.show()

# Visualização dos pointos nos planos XZ
fig = plt.figure(figsize=(8,6))
c = ['b', 'r']
plt.scatter(xdata, ydata, c=c[0])#, label='Dados reais')
plt.scatter(x_prev[:,0], x_prev[:,2], c=c[1])
plt.title('Pontos originais e reconstruídos no plano XZ')
plt.show()

# Visualização dos pointos nos planos YZ
fig = plt.figure(figsize=(8,6))
c = ['b', 'r']
plt.scatter(xdata, ydata, c=c[0])
plt.scatter(x_prev[:,1], x_prev[:,2], c=c[1])
plt.title('Pontos originais e reconstruídos no plano YZ')
plt.show()
```







Os resultados da reconstrução dos dados é insatisfatória. A razão disso é que a coordenada z não depende das coordenadas x e y, além do que usamos um codificador muito simples com somente uma camada e sem não lineridades (funções de ativação lineares).

Observa-se que no plano XY a reconstrução dos dados é um pouco melhor porque tanto x como y dependem da coordenada z, e o AE de alguma forma consegue extrair essa informação.

6. Exemplo de autoencoder com camadas densas

Vamos implementar um autoencoder com camadas densas para reconstruir o conjunto de dígitos MNIST.

Como as imagens do conjunto de dígitos MNIST são pequenas e em tons de cinza, vamos processar as imagens de dígitos usando RNAs somente com camadas densas.

6.1 Carregar e processar dados

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

In [11]:
```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

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

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz (https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz)
11493376/11490434 [==============================] - 0s 0us/step
Dimensão dos dados de entrada de treinamento = (60000, 28, 28)
Dimensão dos dados de entrada de teste = (10000, 28, 28)
Dimensão dos dados de saída de treinamento = (60000,)
Dimensão dos dados de saída de teste = (10000,)
```

- Observa-se que as saídas não são usadas pelo autoencoder.

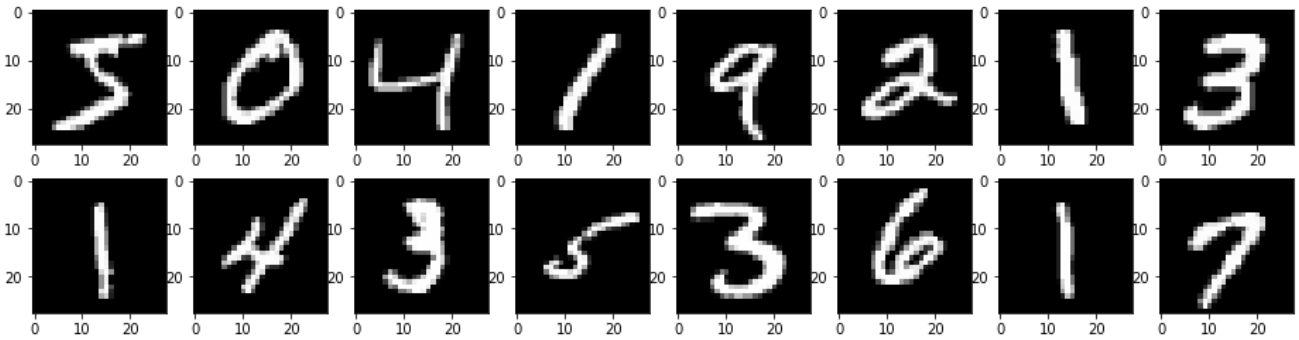
Vamos redimensionar as imagens para transformá-las em vetores.

In [12]:
```
# Número total de pixels nas imagens
nx = x_train.shape[1]*x_train.shape[2]

# Redimensionamento e normalização das imagens
x_train_flat = np.reshape(x_train, (x_train.shape[0], nx))/255.
x_test_flat = np.reshape(x_test, (x_test.shape[0], nx))/255.

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

```
Dimensão dos dados de entrada de treinamento = (60000, 784)
Dimensão dos dados de entrada de teste = (10000, 784)
```

Gráficos de alguns exemplos

In [14]:
```
fig, axs = plt.subplots(2, 8, figsize=(16, 4))
index = 0
for i in range(2):
    for j in range(8):
        axs[i,j].imshow(x_train[index], cmap='gray', vmin=0, vmax=255)
        index += 1
plt.show()
```



6.2 Configuração do autoencoder

Para esse autoencoder vamos usar duas camadas densas no codificador e também no decodificador.

A configuração do autoencoder é realizada usando a classe sequencial do Keras.

Da mesma forma feita para o autoencoder simples, vamos criar separadamente o codificador e o decodificador para depois criar o autoencoder com a composição dos dois. Isso é feito para podermos utilizar o codificador e o decodificador separadamente após o treinamento para gerar o espaço latente e novos dados.

In [18]:

```python
# Importa classes e funções
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization

# Define dimensões das camadas do autoencoder
input_size = 784 # (28*28)
hidden1_size = 128
code_size = 32

# Configura codificador
encoder = Sequential()
encoder.add(Dense(units=hidden1_size, activation='relu', input_shape=(input_size,)))
encoder.add(Dense(units=code_size, activation='relu'))
encoder.add(BatchNormalization())

# Configura decodificador
decoder = Sequential()
decoder.add(Dense(units=hidden1_size, activation='relu', input_shape=(code_size,)))
decoder.add(Dense(units=input_size, 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())
```

```
Codificador:
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 128)               100480
_________________________________________________________________
dense_7 (Dense)              (None, 32)                4128
_________________________________________________________________
batch_normalization_1 (Batch (None, 32)                128
=================================================================
Total params: 104,736
Trainable params: 104,672
Non-trainable params: 64
_________________________________________________________________
None


Decodificador:
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 128)               4224
_________________________________________________________________
dense_9 (Dense)              (None, 784)               101136
=================================================================
Total params: 105,360
Trainable params: 105,360
Non-trainable params: 0
_________________________________________________________________
None


Autoencoder:
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential_6 (Sequential)    (None, 32)                104736
_________________________________________________________________
```

```
sequential_7 (Sequential)    (None, 784)               105360
=================================================================
Total params: 210,096
Trainable params: 210,032
Non-trainable params: 64
_________________________________________________________________
None
```

- Observe que todas as camadas densas usam a função de ativação `relu` a menos da camada de saída do autoencoder que usa a ativação sigmóide porque precisamos que as saídas estejam entre [0, 1]. A entrada também está nesse intervalo.
- Foi incluída uma camada de normalização de batelada na saída do codificador na tentativa de normalizar a sua saída, que consiste na representação latente dos dados, para valores em torno de zero.
- Na primeira camada do decodificador tem que definir a dimensão das entradas para podermos usá-lo separadamente do autoencoder.

6.3 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"

Observa-se que como as saídas do autoencoder são os pixels das imagens normalizados entre 0 e 1, então, esse valores podem ser bem representados por uma probabilidade que possui valores entre 0 e 1. Em razão disso o uso da função de custo entropia cruzada binária é justificada e funciona melhor nesse caso do que o erro quadrático médio.

In [19]:
```
# 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'])
```

6.4 Treinamento do autoencoder

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

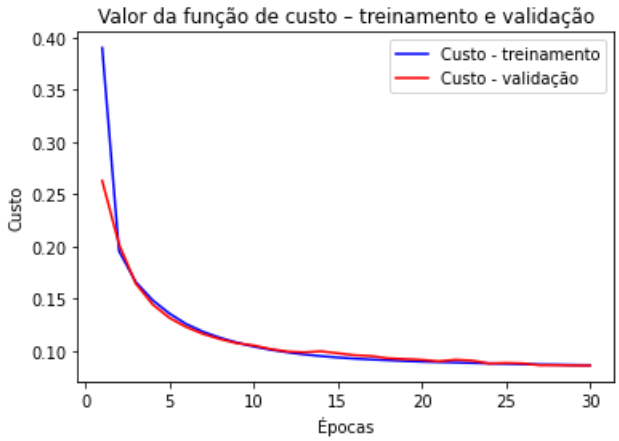
In [20]:
```
1 results = autoencoder.fit(x_train_flat, x_train_flat, epochs=30, batch_size=1024, validation_data=(x_t
```

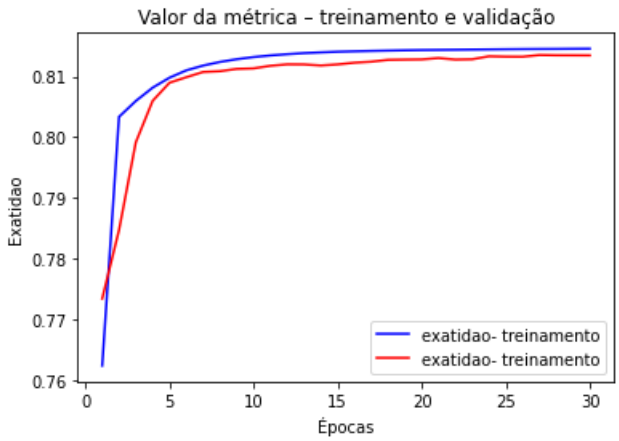
```
Epoch 1/30
59/59 [==============================] - 2s 14ms/step - loss: 0.5236 - binary_accuracy: 0.6870 - val_loss:
0.2630 - val_binary_accuracy: 0.7734
Epoch 2/30
59/59 [==============================] - 1s 11ms/step - loss: 0.2065 - binary_accuracy: 0.8026 - val_loss:
0.2020 - val_binary_accuracy: 0.7847
Epoch 3/30
59/59 [==============================] - 1s 11ms/step - loss: 0.1712 - binary_accuracy: 0.8052 - val_loss:
0.1643 - val_binary_accuracy: 0.7992
Epoch 4/30
59/59 [==============================] - 1s 11ms/step - loss: 0.1519 - binary_accuracy: 0.8078 - val_loss:
0.1443 - val_binary_accuracy: 0.8060
Epoch 5/30
59/59 [==============================] - 1s 11ms/step - loss: 0.1384 - binary_accuracy: 0.8098 - val_loss:
0.1314 - val_binary_accuracy: 0.8090
Epoch 6/30
59/59 [==============================] - 1s 11ms/step - loss: 0.1274 - binary_accuracy: 0.8108 - val_loss:
0.1226 - val_binary_accuracy: 0.8099
Epoch 7/30
59/59 [==============================] - 1s 11ms/step - loss: 0.1198 - binary_accuracy: 0.8114 - val_loss:
0.1160 - val_binary_accuracy: 0.8108
Epoch 8/30
59/59 [==============================] - 1s 11ms/step - loss: 0.1140 - binary_accuracy: 0.8117 - val_loss:
0.1112 - val_binary_accuracy: 0.8109
Epoch 9/30
59/59 [==============================] - 1s 11ms/step - loss: 0.1088 - binary_accuracy: 0.8126 - val_loss:
0.1072 - val_binary_accuracy: 0.8113
Epoch 10/30
59/59 [==============================] - 1s 11ms/step - loss: 0.1046 - binary_accuracy: 0.8139 - val_loss:
0.1050 - val_binary_accuracy: 0.8113
Epoch 11/30
59/59 [==============================] - 1s 11ms/step - loss: 0.1013 - binary_accuracy: 0.8139 - val_loss:
0.1016 - val_binary_accuracy: 0.8118
Epoch 12/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0990 - binary_accuracy: 0.8134 - val_loss:
0.0992 - val_binary_accuracy: 0.8120
Epoch 13/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0967 - binary_accuracy: 0.8135 - val_loss:
0.0984 - val_binary_accuracy: 0.8120
Epoch 14/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0950 - binary_accuracy: 0.8138 - val_loss:
0.0994 - val_binary_accuracy: 0.8118
Epoch 15/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0936 - binary_accuracy: 0.8141 - val_loss:
0.0975 - val_binary_accuracy: 0.8120
Epoch 16/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0928 - binary_accuracy: 0.8142 - val_loss:
0.0957 - val_binary_accuracy: 0.8123
Epoch 17/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0919 - binary_accuracy: 0.8141 - val_loss:
0.0948 - val_binary_accuracy: 0.8124
Epoch 18/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0910 - binary_accuracy: 0.8142 - val_loss:
0.0927 - val_binary_accuracy: 0.8128
Epoch 19/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0903 - binary_accuracy: 0.8143 - val_loss:
0.0920 - val_binary_accuracy: 0.8128
Epoch 20/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0896 - binary_accuracy: 0.8145 - val_loss:
0.0912 - val_binary_accuracy: 0.8128
Epoch 21/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0892 - binary_accuracy: 0.8146 - val_loss:
0.0898 - val_binary_accuracy: 0.8130
Epoch 22/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0888 - binary_accuracy: 0.8143 - val_loss:
0.0913 - val_binary_accuracy: 0.8128
Epoch 23/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0882 - binary_accuracy: 0.8145 - val_loss:
0.0905 - val_binary_accuracy: 0.8128
Epoch 24/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0879 - binary_accuracy: 0.8145 - val_loss:
0.0877 - val_binary_accuracy: 0.8133
Epoch 25/30
```

```
59/59 [==============================] - 1s 11ms/step - loss: 0.0875 - binary_accuracy: 0.8147 - val_loss:
0.0882 - val_binary_accuracy: 0.8133
Epoch 26/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0871 - binary_accuracy: 0.8145 - val_loss:
0.0878 - val_binary_accuracy: 0.8133
Epoch 27/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0869 - binary_accuracy: 0.8144 - val_loss:
0.0861 - val_binary_accuracy: 0.8135
Epoch 28/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0867 - binary_accuracy: 0.8145 - val_loss:
0.0861 - val_binary_accuracy: 0.8135
Epoch 29/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0863 - binary_accuracy: 0.8146 - val_loss:
0.0859 - val_binary_accuracy: 0.8135
Epoch 30/30
59/59 [==============================] - 1s 11ms/step - loss: 0.0862 - binary_accuracy: 0.8146 - val_loss:
0.0858 - val_binary_accuracy: 0.8135
```

In [22]:

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

    # Salva custos, métricas 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- treinamento')
    plt.title('Valor da métrica – treinamento e validação')
    plt.xlabel('Épocas')
    plt.ylabel('Exatidao')
    plt.legend()
    plt.show()

plot_train(results)
```





- Observa-se que não ocorre overfitting no treinamento.

6.5 Avaliação do autoencoder

Para avaliar o desempenho do autoencoder vamos calcular a função de custo e a métrica para os dados de treinamento e teste.

In [23]:
```
# Calcula função de custo e métrica
autoencoder.evaluate(x_train_flat, x_train_flat)
autoencoder.evaluate(x_test_flat, x_test_flat)
```

```
1875/1875 [==============================] - 5s 2ms/step - loss: 0.0863 - binary_accuracy: 0.8144
313/313 [==============================] - 1s 2ms/step - loss: 0.0858 - binary_accuracy: 0.8135
```

Out[23]:
```
[0.08577760308980942, 0.8134880661964417]
```

- Observa-se que a exatidão é da ordem de 81% tanto para os dados de treinamento como de teste. Esse resultado não é muito bom, mas tendo em vista a grande diversidade de imagens também não é muito ruim.

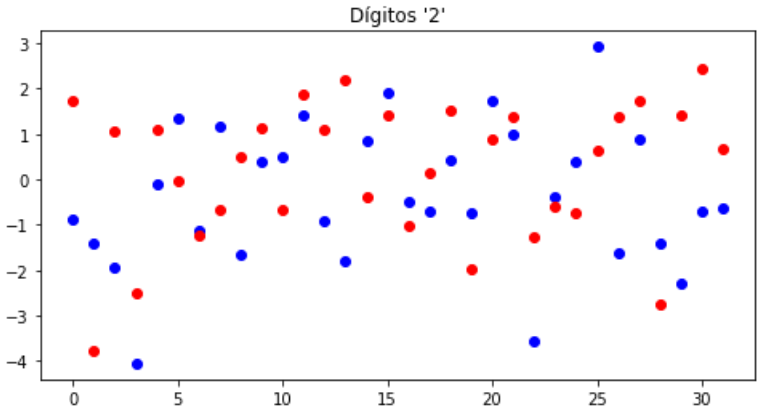
6.6 Espaço latente dos dígitos

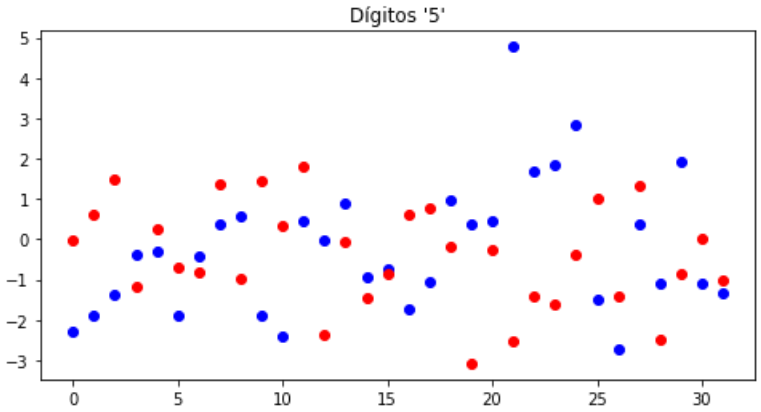
Para poder reconstruir os dados a partir do espaço latente devemos obter os códigos que representam os dados no espaço latente de alguns dígitos 2 e 5.

In [24]:

```
# Calcula espaço latente (saída do codificador)
code = encoder.predict(x_test_flat)
print('Dimensão do espaço latente:', code.shape)

# Seleciona dígitos iguais a 2
ind2 = np.where((y_test == 2))

# Seleciona digitos iguais a 5
ind5 = np.where((y_test == 5))

# Gráfico dos códigos dos dígitos 2
plt.figure(figsize=(8,4))
plt.plot(code[ind2[0][0]], 'bo')
plt.plot(code[ind2[0][1]], 'ro')
plt.title("Dígitos '2'")
plt.show()

# Gráfico dos códigos dos dígitos 5
plt.figure(figsize=(8,4))
plt.plot(code[ind5[0][0]], 'bo')
plt.plot(code[ind5[0][1]], 'ro')
plt.title("Dígitos '5'")
plt.show()
```

```
Dimensão do espaço latente: (10000, 32)
```





- Observa-se que não é possível analisar esses dados, pois eles somente tem algum significado para o decodificador.

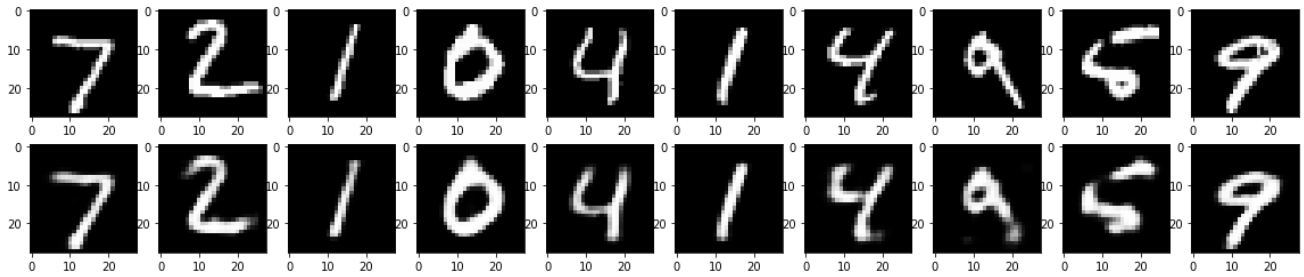
6.7 Comparação das saídas previstas pelo autoencoder com as entradas

Para verificar visualmente o desempenho do autoencoder, vamos reconstruir algumas imagens de teste.

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

In [25]:

```
# Calcula dados reconstriuídos pelo AE
x_prev = 255*decoder.predict(code)
x_prev = x_prev.astype(int)

# Mostra imagens originais e reconstruídas
f, pos = plt.subplots(2, 10, figsize=(20, 4))
for i in range(10):
    img_prev = np.reshape(x_prev[i], [28,28])
    pos[0,i].imshow(x_test[i], cmap='gray')
    pos[1,i].imshow(img_prev, cmap='gray')
plt.show()
```



- A primeira linha são as imagens originais e a segunda linha são as imagens reconstruídas.
- Observe que as imagens reconstruídas são muito semelhantes às originais, mas não são exatamente iguais.
- Como essa era uma tarefa simples, o autoencoder apresenta um desempenho muito bom.

6.8 Criação de novas imagens de dígitos

As representações latentes podem ser modificadas para criar novas imagens. As operações que podem ser realizadas no espaço latente com um autoencoder são as seguintes:

- Adição de ruído;
- Combinação linear de representações latentes de vários exemplos;
- Alteração da intensidade luminosa.

Adição de ruído

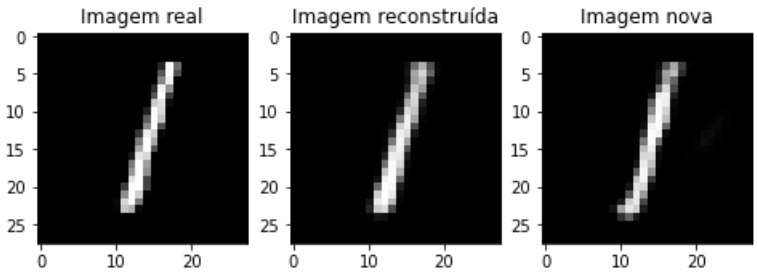
Vamos adicionar um ruído à representação latente de um dígito e verificar como é alterada a imagem reconstruída com essa nova representação latente.

O ruído adicionado possui distribuição normal com média zero. Vamos testar diferentes desvios padrão para verificar a sua influência na nova imagem.

In [27]:

```
"""Adição de ruído"""

# Trocar o valor de index para usar outras imagens
index = 2

# Seleciona código de um dígito 2
codei = code[index]
codei = np.reshape(codei, [1,32,])

# Média e desvio padrão do código
print('Média:', np.mean(codei))
print('Desvio padrão:', np.std(codei))

# Reconstroi imagem original
img_prev = 255*decoder.predict(codei)
img_prev = img_prev.astype(int)
img_prev = np.reshape(img_prev, [28,28])

# Define desvio padrão do ruído
dvp = 0.5

# Cria nova imagem adicionando ruído gaussiano ao código
code_noise = codei + dvp*np.random.randn(1,32)
img_noise = 255*decoder.predict(code_noise)
img_noise = img_noise.astype(int)
img_noise = np.reshape(img_noise, [28,28])

# Mostra imagens de dígitos 2
f, pos = plt.subplots(1, 3, figsize=(8, 8))
pos[0].imshow(x_test[index], cmap='gray')
pos[0].set_title('Imagem real')
pos[1].imshow(img_prev, cmap='gray')
pos[1].set_title('Imagem reconstruída')
pos[2].imshow(img_noise, cmap='gray')
pos[2].set_title('Imagem nova')
plt.show()
```

```
Média: -0.5522667
Desvio padrão: 1.1421635
```



- Obviamente, na medida em que o desvio padrão do ruído adicionado à representação latente aumenta, a nova imagem se torna mais diferente da original.

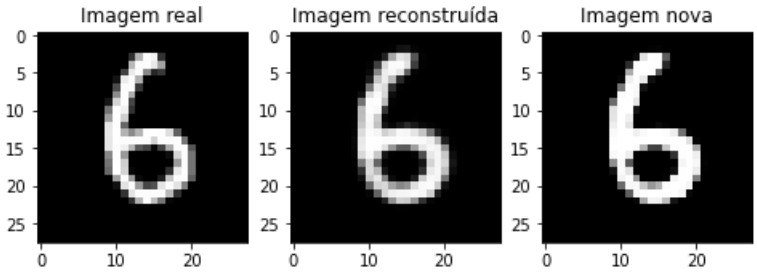
Alteração da intensidade luminosa

Vamos alterar a amplitude da representação latente das imagens, multiplicando o código das imagens por uma constante e verificar o efeito que tem na nova imagem criada.

Observa-se que essa constante deve ter valor positivo.

In [30]:

```
"""Alteração da intensidade luminosa"""

# Trocar o valor de index para usar outras imagens
index = 21

# Seleciona código de um dígito 2
codei = code[index]
codei = np.reshape(codei, [1,32,])

# Reconstroi imagem original
img_prev = 255*decoder.predict(codei)
img_prev = img_prev.astype(int)
img_prev = np.reshape(img_prev, [28,28])

# Define fator para alteração da intensidade luminosa (f_lum = 1, não altera imagem original)
f_lum = 2

# Cria nova imagem adicionando ruído gaussiano ao código
code_amp = f_lum*codei
img_amp = 255*decoder.predict(code_amp)
img_amp = img_amp.astype(int)
img_amp = np.reshape(img_amp, [28,28])

# Mostra imagens de dígitos 2
f, pos = plt.subplots(1, 3, figsize=(8, 8))
pos[0].imshow(x_test[index], cmap='gray')
pos[0].set_title('Imagem real')
pos[1].imshow(img_prev, cmap='gray')
pos[1].set_title('Imagem reconstruída')
pos[2].imshow(img_amp, cmap='gray')
pos[2].set_title('Imagem nova')
plt.show()
```



- Observe que a alteração da amplitude dos valores da representação latente para essas imagens de dígitos tem o efeito de alterar o contraste da imagem.

Combinação linear de duas imagens

Duas imagens podem ser combinadas para gerar um nova imagem.

Essa combinação é realizada pela combinação linear das representações latentes de duas ou mais imagens.

A nova imagem gerada pela combinação de duas representações latentes de dígitos diferentes é uma combinação dos dois dígitos e depende do peso de ponderação dado para cada código.

Vamos realizar uma operação que consiste na transição de um dígito para outro. Essa transição é realizada alterando linearmente os pesos dos códigos de cada imagem. Usaremos a seguinte equação para obter a representação latente da combinação dos dois dígitos:

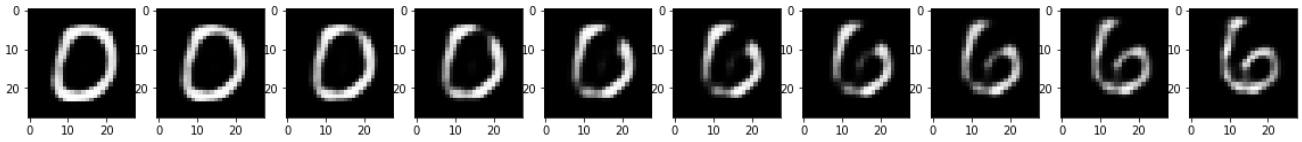
$$code_{comb} = code_1 + (code_2 - code_1) * \alpha$$

onde $code_1$ e $code_2$ são as representações latentes de duas imagens, $code_{com}$ é a representação latente da combinação das duas imagens originais e α é um valor que varia linearmente entre 0 e 1.

In [31]:

```
"""Conbinação linear de dois dígitos diferentes"""

# Trocar os valores dos indexes para usar outras imagens
index1 = 10
index2 = 100

# Obtém os codigos das imagens selecionadas
code1 = code[index1]
code2 = code[index2]

# Classes de dígitos da imagens selecionadas
print('Digito da imagem 1:', y_test[index1])
print('Digito da imagem 2:', y_test[index2])

# Inclui eixo dos exemplos nos códigos
code1 = np.reshape(code1, [1,32,])
code2 = np.reshape(code2, [1,32,])

# Cria sequencia de 10 imagens transitando da imagem index1 para a imagem index2
f, pos = plt.subplots(1, 10, figsize=(20, 4))
for i in range(10):
    # Gera novo código pela combinação linear dos dois códigos
    code_comb = code1 + (code2 - code1)*i/9

    # Reconstroi imagem
    img_comb = 255*decoder.predict(code_comb)
    img_comb = img_comb.astype(int)
    img_comb = np.reshape(img_comb, [28,28])
    pos[i].imshow(img_comb, cmap='gray')
plt.show()
```

```
Digito da imagem 1: 0
Digito da imagem 2: 6
```



6.9 Análise dos resultados

Os resultados obtidos por esse autoencoder podem ser considerados muito bons, sendo que ele é capaz de reproduzir as imagens de entrada usando as suas representações latentes. Além disso, o autoencoder é capaz de criar nova imagens diferentes das originais usando transfromações no espaço latente.

Se for desejado, podemos melhorar o autoencoder aumentando o número de camadas e o número de neurônios, ou talvez usar camadas convolucionais.

Aumentar o número de parâmetros permite que o autoencoder aprenda codificações mais complexas. Porém, deve-se ter cuidado, pois um grande número de parâmetros pode causar sobreajuste dos dados de treinamento e, portanto, o autoencoder perderá a capacidade de generalização.

7. Autoencoder com camadas convolucionais

Como já vimos as camadas convolucionais são muito eficientes para extrair características de imagens, portanto, são também muito boas para criar imagens a partir de representações latente.

A arquitetura de um autoencoder com camadas convolucionais segue a mesma filosofia de um autoencoder com camadas densas, ou seja:

- O codificador do autoencoder deve reduzir a resolução da imagem enquanto aumenta o número de filtros.
- O decodificador deve aumentar a resolução da imagem enquanto diminui o número de filtros.
- A camada de saída do decodificador deve ter as mesmas dimensões da imagem original.

Na Figura 5 é apresentado um exemplo de um esquema de arquitetura de um autoencoder com camadas convolucionais. Esse esquema mostra um autoencoder configurado para processar as imagens do conjunto de dígitos MNIST.

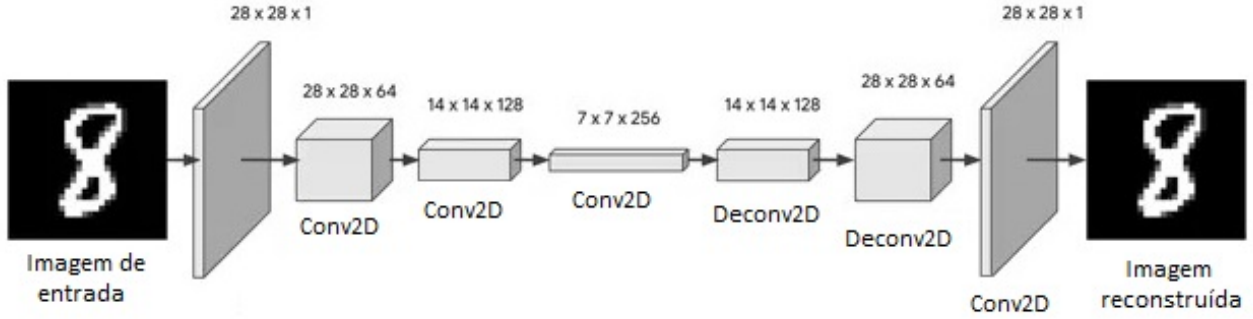


Figura 5 - Esquema de um autoencoder convolucional aplicado às imagens do conjunto de dígitos MNIST.

Para exemplificar o uso de autoencoders com camadas convolucionais vamos usar o conjunto de dados CIFAR10, que contém 10 classes de imagens coloridas.

7.1 Carregar e processar dados

O conjunto de dados CIFAR1-10 pode ser carregado diretamente do Keras. Os comandos para carregar esse conjunto de dados pode ser visto no link https://www.tensorflow.org/api_docs/python/tf/keras/datasets/cifar10/load_data (https://www.tensorflow.org/api_docs/python/tf/keras/datasets/cifar10/load_data).

Características dos dados:

- As imagens são coloridas e estão no padrão RGB;
- Cada imagem tem dimensão de 32x32x3;
- O valor da intensidade luminosa de cada plano de cor é um número inteiro entre 0 e 255;
- As saídas representam o rótulo do objeto mostrado na imagem, sendo um número inteiro de 0 a 9.

In [3]:
```
# Leitura do arquivo de dados
(x_train_orig, y_train_orig), (x_test_orig, y_test_orig) = tf.keras.datasets.cifar10.load_data()

print("Dimensão x_train_orig:", x_train_orig.shape, "Dimensão y_train_orig:", y_train_orig.shape)
print("Dimensão x_test_orig:", x_test_orig.shape, "Dimensão y_test:", y_test_orig.shape)
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz (https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz)
170500096/170498071 [==============================] - 2s 0us/step
Dimensão x_train_orig: (50000, 32, 32, 3) Dimensão y_train_orig: (50000, 1)
Dimensão x_test_orig: (10000, 32, 32, 3) Dimensão y_test: (10000, 1)
```

Pela dimensão dos tensores com os dados de treinamento e teste temos:

- 50.000 imagens de treinamento com dimensão de 32x32x3 pixels;
- 10.000 imagens de teste com dimensão de 32x32x3 pixels.

Na célula abaixo é realizada a normalização dos pixels das imagens.

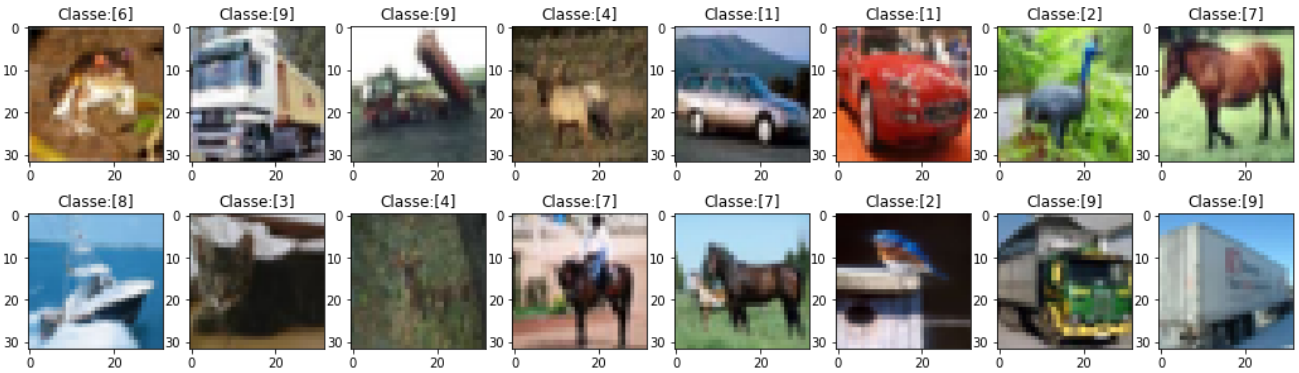
In [20]:
```
# Guarda dimensão das imagens
image_dim = x_train_orig.shape[1:4]
print("Dimensão das imagens de entrada=", image_dim)

# Transformação dos dados em números reais entre 0 e 1
x_train = x_train_orig.astype('float32')/255.
x_test = x_test_orig.astype('float32')/255.
```

```
Dimensão das imagens de entrada= (32, 32, 3)
```

Vamos visualizar algumas imagens juntamente com as suas classes.

In [21]:

```
fig, axs = plt.subplots(2, 8, figsize=(18, 5))
index = 0
for i in range(2):
    for j in range(8):
        axs[i,j].imshow(x_train[index], cmap='gray')
        axs[i,j].set_title('Classe:' + str(y_train_orig[index]))
        index += 1
plt.show()
```



7.2 Configuração do autoencoder

A arquitetura do codificador é a seguinte:

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

A arquitetura do decodificador é a seguinte:

- Camada de deconvolução com 32 filtros, dimensão dos filtros=3x3, stride=2, função de ativação ReLu, padding=same;
- Camada de deconvolucão 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 convolucional com 3 filtros, dimensão dos filtros=1x1, stride=1, função de ativação sigmode, padding=same.

Observa-se que para os autoencoders convolucionais foi verificado que:

- É melhor utilizar stride=2 nas camadas convolucionais do que camadas de pooling. O efeito de redução das dimensões das imagens é o mesmo mas o desempenho final do autoencoder é melhor.
- Nas camadas do codificador é melhor usar função de ativação LeakyReLu e nas camadas do decodificar função de ativação ReLu.

In [37]:

```
# 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(32, kernel_size=3, strides=2, padding='same', activation=LeakyReLU(), input_shape=ir
encoder.add(Conv2D(32, kernel_size=3, strides=2, padding='same', activation=LeakyReLU()))
encoder.add(Conv2D(32, kernel_size=3, strides=2, padding='same', activation=LeakyReLU()))
encoder.add(BatchNormalization())

# Configura decodificador
decoder = Sequential()
decoder.add(Conv2DTranspose(32, kernel_size=3, strides=2, padding='same', activation='relu', input_sha
decoder.add(Conv2DTranspose(16, kernel_size=3, strides=2, padding='same', activation='relu'))
decoder.add(Conv2DTranspose(8, kernel_size=3, strides=2, padding='same', activation='relu'))
decoder.add(Conv2D(3, 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())
```

```
Codificador:
Model: "sequential_12"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_15 (Conv2D)           (None, 16, 16, 32)        896
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 8, 8, 32)          9248
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 4, 4, 32)          9248
_________________________________________________________________
batch_normalization_4 (Batch (None, 4, 4, 32)          128
=================================================================
Total params: 19,520
Trainable params: 19,456
Non-trainable params: 64
_________________________________________________________________
None


Decodificador:
Model: "sequential_13"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_transpose_9 (Conv2DTr (None, 8, 8, 32)          9248
_________________________________________________________________
conv2d_transpose_10 (Conv2DT (None, 16, 16, 16)        4624
_________________________________________________________________
conv2d_transpose_11 (Conv2DT (None, 32, 32, 8)         1160
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 32, 32, 3)         27
=================================================================
Total params: 15,059
Trainable params: 15,059
Non-trainable params: 0
_________________________________________________________________
None


Autoencoder:
Model: "sequential_14"
_________________________________________________________________
```

```
Layer (type)                 Output Shape              Param #
=================================================================
sequential_12 (Sequential)   (None, 4, 4, 32)          19520
_________________________________________________________________
sequential_13 (Sequential)   (None, 32, 32, 3)         15059
=================================================================
Total params: 34,579
Trainable params: 34,515
Non-trainable params: 64
_________________________________________________________________
None
```

- Observe que a função de ativação LeakyReLu é uma camada, assim, deve ser importada e usada como um objeto.
- A saída do codificar é um tensor de dimensão 4x4x32, que possui 512 elementos. Note que as imagens originais com dimensão 32x32x3 possuem 3072 pixels, portanto, o espaço latente consiste de uma redução de 6 vezes em relação às imagens originais.
- Note que as possibilidades de configuração de um autoencoder são muitas e essa que foi escolhida é somente um exemplo.

7.3 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: "mae"

Observa-se que como as saídas do autoencoder são os pixels das imagens normalizados entre 0 e 1, então, esse valores podem ser bem representados por uma probabilidade que possui valores entre 0 e 1.

A métrica erro absoluto médio é mais adequada para esse problema porque representa um erro médio entre os valores de pixels previstos e os reais das imagens.

In [40]:
```
# Define otimizador Adam
adam = tf.keras.optimizers.Adam(learning_rate=0.001)

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

7.4 Treinamento do autoencoder

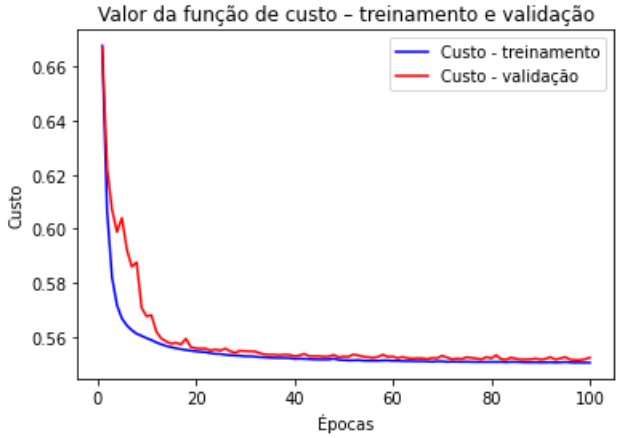
Vamos treinar o autoencoder usando 100 épocas e lotes de 1024 elementos.

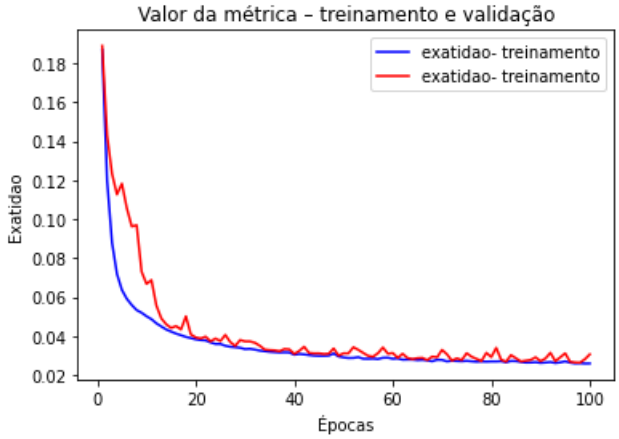
In [41]:
```
results = autoencoder.fit(x_train, x_train, epochs=100, batch_size=1024, validation_data=(x_test, x_te
```

```
Epoch 1/100
49/49 [==============================] - 3s 49ms/step - loss: 0.6832 - mae: 0.2022 - val_loss: 0.6672 -
val_mae: 0.1890
Epoch 2/100
49/49 [==============================] - 2s 42ms/step - loss: 0.6162 - mae: 0.1316 - val_loss: 0.6222 -
val_mae: 0.1430
Epoch 3/100
49/49 [==============================] - 2s 42ms/step - loss: 0.5853 - mae: 0.0929 - val_loss: 0.6072 -
val_mae: 0.1233
Epoch 4/100
49/49 [==============================] - 2s 41ms/step - loss: 0.5738 - mae: 0.0748 - val_loss: 0.5989 -
val_mae: 0.1127
Epoch 5/100
49/49 [==============================] - 2s 41ms/step - loss: 0.5677 - mae: 0.0649 - val_loss: 0.6040 -
val_mae: 0.1182
Epoch 6/100
49/49 [==============================] - 2s 41ms/step - loss: 0.5641 - mae: 0.0601 - val_loss: 0.5925 -
val_mae: 0.1057
Epoch 7/100
```

In [44]:

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

    # Salva custos, métricas e epocas em vetores
    custo = history_dict['loss']
    acc = history_dict['mae']
    val_custo = history_dict['val_loss']
    val_acc = history_dict['val_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.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- treinamento')
    plt.title('Valor da métrica – treinamento e validação')
    plt.xlabel('Épocas')
    plt.ylabel('Exatidao')
    plt.legend()
    plt.show()

plot_train(results)
```





7.5 Avaliação do autoencoder

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

In [45]:
```
# Calcula função de custo e métrica
autoencoder.evaluate(x_train, x_train)
autoencoder.evaluate(x_test, x_test)
```

```
1563/1563 [==============================] - 5s 3ms/step - loss: 0.5514 - mae: 0.0305
313/313 [==============================] - 1s 3ms/step - loss: 0.5523 - mae: 0.0306
```

Out[45]:
```
[0.5523313283920288, 0.03062686137855053]
```

- Observa-se que o erro absoluto médio é da ordem de 0,03 tanto para os dados de treinamento como de teste. Esse resultado é bom, tendo em vista a complexidade das imagens e o fato do espaço latente ser 6 vezes menor do que as imagens originais.
- Note que se for desejado, provavelmente é possível obter resultados melhores aumentando o número de camadas e o número de filtros do autoencoder.

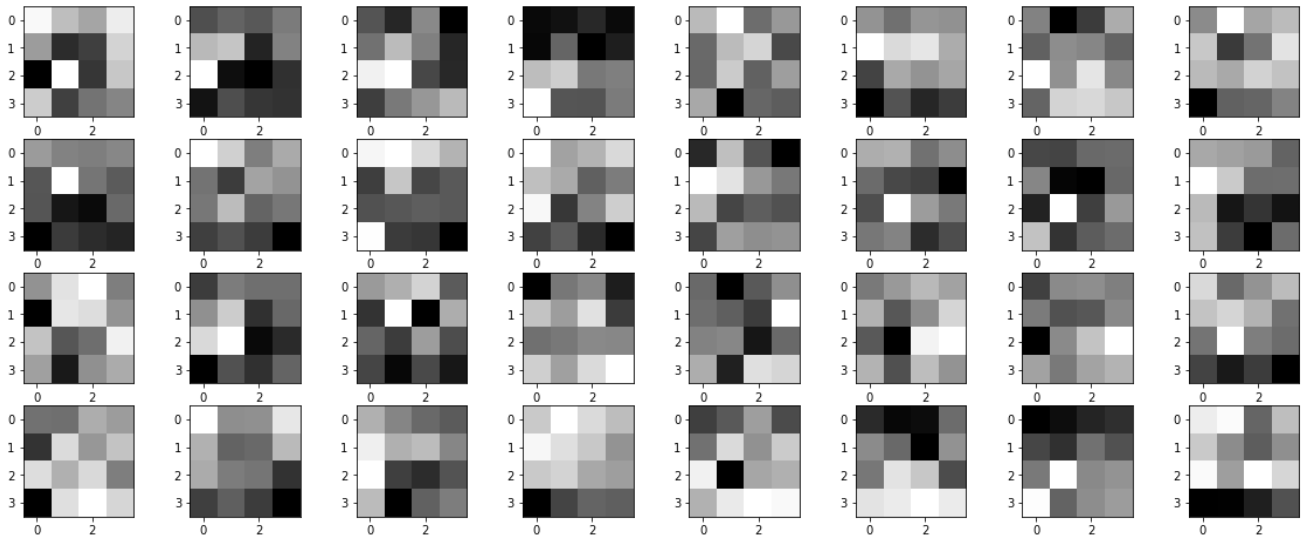
7.6 Espaço latente das imagens

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 [48]:
```
# Calcula códigos das imagens de teste
code = encoder.predict(x_test)
print('Dimensão do espaço latente:', code.shape)

# Seleciona imagem do conjunto de teste
index = 0

# Mostra as 32 imagens
cont = 0
f, pos = plt.subplots(4, 8, figsize=(20, 8))
for i in range(4):
    for j in range(8):
        pos[i,j].imshow(code[index,:,:,cont], cmap='gray')
        cont += 1
plt.show()
```

```
Dimensão do espaço latente: (10000, 4, 4, 32)
```



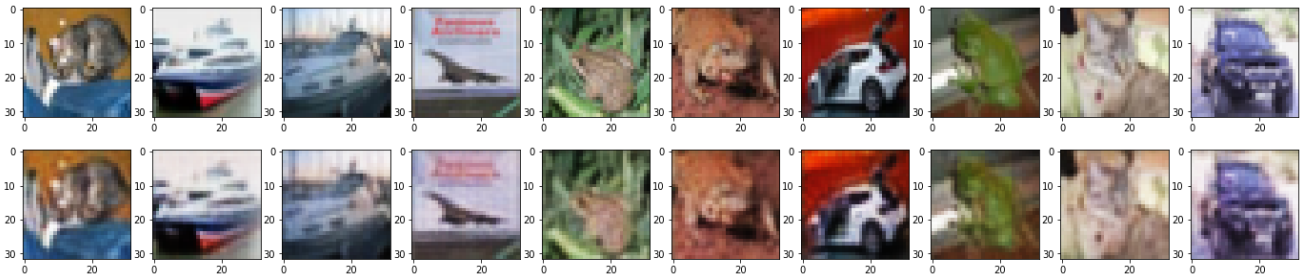
- Obviamente não é possível analisar esses dados, pois eles somente tem algum significado para o decodificador.
- Observa-se que cada imagem de dimensão 4x4 do espaço latente representa a presença (pixel mais claro) ou não (pixel mais escuro) de uma determinada característica em 16 regiões diferentes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

7.7 Comparação das saídas previstas pelo autoencoder com as entradas

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 [49]:
```
# Reconstrução das imagens usando o espaço latente e o decodificador
x_prev = 255*decoder.predict(code)
x_prev = x_prev.astype(int)

# Mostra imagens originais e reconstruídas
f, pos = plt.subplots(2, 10, figsize=(24, 5))
for i in range(10):
    img_prev = np.reshape(x_prev[i], image_dim)
    pos[0,i].imshow(x_test[i], cmap='gray')
    pos[1,i].imshow(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 grande compactação existente entre a imagem original e a sua representação latente.

O erro médio porcentual entre as imagens reconstruídas e as originais é calculado na célula abaixo para termos uma ideía da sua grandeza

In [53]:
```
# Erro médio absoluto
erro = np.mean(np.abs(x_test-x_prev/255.))
erro_per = 100*erro/np.mean(x_test)
print('Erro médio porcentual:', erro_per,'%')
```

```
Erro médio porcentual: 6.527634484746409 %
```

7.8 Criação de novas imagens

Com visto, as representações latentes podem ser modificadas para criar novas imagens.

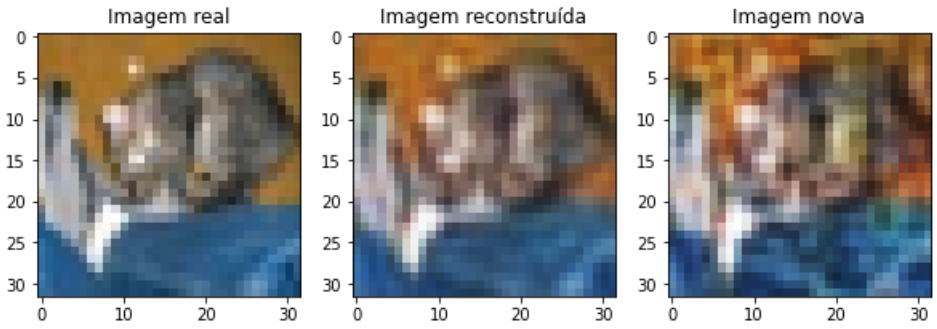
Vamos criar novas imagens usando as mesmas transformações aplicadas nas representações latentes dos dígitos MNIST realizada anteriormente, ou seja:

1. Adição de ruído gaussiano com média zero na representação latente;
2. Alteração da amplitude da representação latente;
3. Combinação da representação latente de duas imagens.

In [58]:

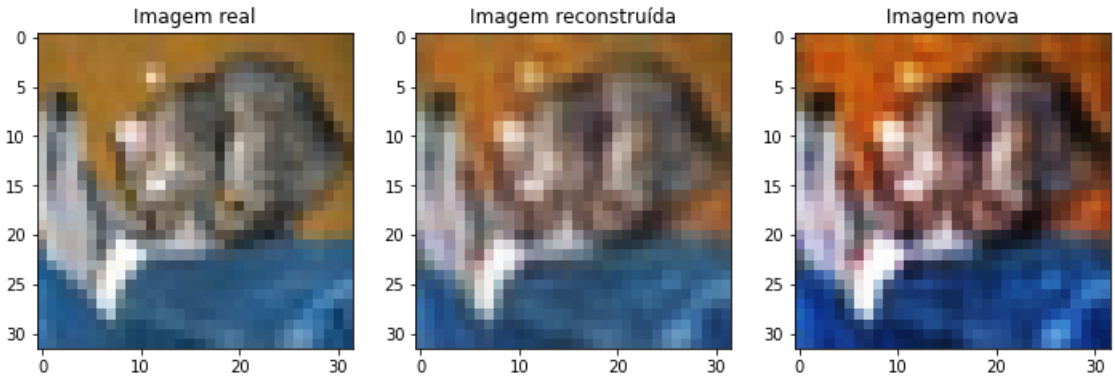
```
"""Adição de ruído"""

# Trocar o valor de index para mostrar outras imagens
index = 0

# Seleciona código de um dígito 2
codeI = code[index]

# Insere eixo dos exemplos
codeI = np.expand_dims(codeI, axis=0)

# Reconstroi imagem original
img_prev = 255*decoder.predict(codeI)
img_prev = img_prev.astype(int)

# Cálculo da média e desvio padrão dos códigos
print('Média:', np.mean(codeI))
print('Desvio padrão:', np.std(codeI))

# Desvio padrão do ruído
dvp = 0.4

# Cria nova imagem adicionando ruído gaussiano ao código
code_noise = codeI + dvp*np.random.randn(1,4,4,32)
img_noise = 255*decoder.predict(code_noise)
img_noise = img_noise.astype(int)

# Mostra imagens de dígitos 2
f, pos = plt.subplots(1, 3, figsize=(10, 8))
pos[0].imshow(x_test[index])
pos[0].set_title('Imagem real')
pos[1].imshow(img_prev[0])
pos[1].set_title('Imagem reconstruída')
pos[2].imshow(img_noise[0])
pos[2].set_title('Imagem nova')
plt.show()
```

```
Média: -0.03584589
Desvio padrão: 0.9440253
```



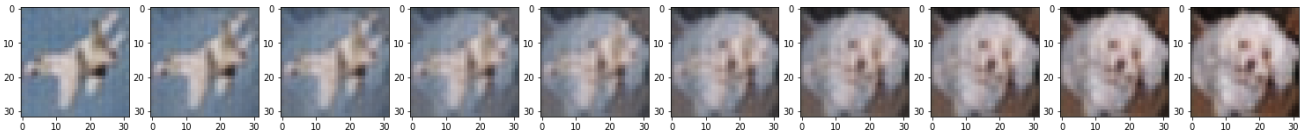
In [57]:

```
"""Alteração de intensidade"""

# Trocar o valor de index para mostrar outras imagens
index = 0

# Seleciona código de um dígito 2
codeI = code[index]

# Insere eixo dos exemplos
codeI = np.expand_dims(codeI, axis=0)

# Reconstroi imagem original
img_prev = 255*decoder.predict(codeI)
img_prev = img_prev.astype(int)

# Fator de alteração
f_lum = 1.5

# Cria nova representação latente alterando amplitude
code_amp = f_lum*codeI
img_amp = 255*decoder.predict(code_amp)
img_amp = img_amp.astype(int)

# Mostra imagens de dígitos 2
f, pos = plt.subplots(1, 3, figsize=(10, 8))
pos[0].imshow(x_test[index])
pos[0].set_title('Imagem real')
pos[1].imshow(img_prev[0])
pos[1].set_title('Imagem reconstruída')
pos[2].imshow(img_amp[0])
pos[2].set_title('Imagem nova')
plt.show()
```



In [59]:

```
"""Conbinação linear de duas images diferentes"""

# Trocar o valor de index para usar outras imagens
index1 = 10
index2 = 1000

# Obter os códigos das imagens selecionadas
code1 = code[index1]
code2 = code[index2]

# Gera imagens de transição da imagem 1 para a imagem 2
f, pos = plt.subplots(1, 10, figsize=(26, 6))
for i in range(10):
    # Gera novo código
    codei = code1 + (code2 - code1)*i/9

    # Inclui eixo dos exemplos
    codei = np.expand_dims(codei, axis=0)

    # Reconstroi imagem
    imgi = 255*decoder.predict(codei)
    imgi = imgi.astype(int)
    pos[i].imshow(imgi[0])
plt.show()
```



7.9 Análise dos resultados

Podemos conluir que um autoencoder é uma ferramenta poderosa para comprimir imagens.

Porém a geração de novas imagens por um autoencoder é limitada, consistindo basicamente de misturar imagens utlizadas no treinamento.