Rede Neural Convolucional (CNN) - Introdução

Exemplo com base de dados rotulada de imagens (MNIST)

Importar as bibliotecas

In [7]:

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
```

Dividir a base de dados em conjuntos de treinamento e teste e normalizar.

- 1 - Carregar a base de dados Fashion MNIST e divida-a em conjuntos de treinamento e teste.
- 2 - Pré-processamento dos dados:
- 3 - Normalizar os pixels da imagem para que estejam entre 0 e 1.
- 4 Transforar os rótulos em vetores de categoria.

In [2]:

```
# Download da base de dados Fashion MNIST
(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()

# Normalizando as imagens
x_train = x_train / 255.0
x_test = x_test / 255.0

y_train = keras.utils.to_categorical(y_train, num_classes=10)
y_test = keras.utils.to_categorical(y_test, num_classes=10)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-d
atasets/train-labels-idx1-ubyte.gz (https://storage.googleapis.com/tensorf
low/tf-keras-datasets/train-labels-idx1-ubyte.gz)
29515/29515 [==============================] - 0s 1us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-d
atasets/train-images-idx3-ubyte.gz (https://storage.googleapis.com/tensorf
low/tf-keras-datasets/train-images-idx3-ubyte.gz)
26421880/26421880 [==============================] - 2s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-d
atasets/t10k-labels-idx1-ubyte.gz (https://storage.googleapis.com/tensorfl
ow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz)
5148/5148 [==============================] - 0s 0s/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-d
atasets/t10k-images-idx3-ubyte.gz (https://storage.googleapis.com/tensorfl
ow/tf-keras-datasets/t10k-images-idx3-ubyte.gz)
4422102/4422102 [==============================] - 0s 0us/step
```

Arquitetura da Rede Neural Convolucional

In [3]:

```
# Definindo a arquitetura da CNN
model = keras.Sequential()
model.add(keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(28, 28, 1)))
model.add(keras.layers.MaxPooling2D((2,2)))
model.add(keras.layers.Conv2D(64, (3,3), activation='relu'))
model.add(keras.layers.MaxPooling2D((2,2)))
model.add(keras.layers.Conv2D(64, (3,3), activation='relu')) ##
model.add(keras.layers.MaxPooling2D((2, 2))) ##
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(128, activation='relu'))
model.add(keras.layers.Dense(10, activation='softmax'))
model.summary()


# Compilando o modelo
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])


# Treinando o modelo
model.fit(x_train, y_train, epochs=50)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 3, 3, 64)          36928

 max_pooling2d_2 (MaxPooling  (None, 1, 1, 64)         0
 2D)
```

Avaliação do modelo

Uma métrica comum para avaliar o desempenho de um modelo de classificação é a acurácia, que mede a porcentagem de previsões corretas do modelo. Pode-se calcular a acurácia do seu modelo no conjunto de teste da seguinte maneira:

In [4]:

```
# Avaliando o modelo no conjunto de teste
test_loss, test_acc = model.evaluate(x_test, y_test)
print('Acuracia do teste:', test_acc)
```

```
313/313 [==============================] - 2s 7ms/step - loss: 0.8573 - ac
curacy: 0.8876
Acuracia do teste: 0.8876000046730042
```

A acurácia é uma métrica comum para avaliar o desempenho de um modelo de classificação e é definida como a porcentagem de previsões corretas do modelo. Portanto, uma acurácia de 89% significa que o modelo conseguiu prever corretamente 89% das classes das imagens do conjunto de teste.

É importante lembrar que a acurácia por si só pode não ser uma medida suficientemente boa do desempenho do modelo, pois ela não leva em consideração o desbalanceamento de classes. Por exemplo, se o conjunto de dados tiver 90% de imagens de uma classe e 10% de imagens de outra classe, um modelo que sempre preveja a classe mais prevalente terá uma acurácia de 90%, mas isso não significa que o modelo esteja realmente aprendendo a distinguir as duas classes. Nesses casos, é importante avaliar o modelo usando outras métricas, como a matriz de confusão abaixo, que mostra que na diagonal principal os volumes de acertos realmente apontam para um bom desempenho do modelo.

In [8]:

```
from sklearn.metrics import confusion_matrix
y_pred = model.predict(x_test)
y_pred = np.argmax(y_pred, axis=1)

y_test = np.argmax(y_test, axis=-1)

print('Matriz de Confusao')

confusion_matrix(y_test, y_pred)
```

```
313/313 [==============================] - 2s 7ms/step
Matriz de Confusao
```

Out[8]:

```
array([[861,   0,  24,  19,   7,   2,  79,   0,   8,   0],
       [  3, 974,   4,  13,   4,   0,   2,   0,   0,   0],
       [ 19,   0, 820,  17,  82,   0,  62,   0,   0,   0],
       [ 31,  12,  18, 875,  37,   0,  27,   0,   0,   0],
       [  2,   1,  43,  23, 876,   0,  54,   0,   1,   0],
       [  2,   0,   0,   0,   0, 976,   0,  14,   1,   7],
       [136,   1,  72,  33, 127,   0, 625,   0,   6,   0],
       [  0,   0,   0,   0,   0,  10,   0, 974,   0,  16],
       [  7,   0,   5,   3,  11,   5,   9,   1, 958,   1],
       [  1,   0,   0,   0,   1,  10,   0,  51,   0, 937]], dtype=int64)
```

Redução da dimensionalidade por Encoder + MLP

Um encoder é uma parte de uma rede neural que é usada para codificar os dados de entrada em uma representação de menor dimensionalidade, geralmente com o objetivo de reduzir a complexidade dos dados ou para facilitar a generalização do modelo.

1 - Definir a arquitetura do encoder. Isso inclui escolher o número de camadas, o número de filtros em cada camada, o tamanho dos filtros e a função de ativação a ser usada.

2 - Compilar o encoder. Escolher uma função de perda, um otimizador e uma métrica para avaliar o desempenho do encoder.

3 - Treinar o encoder. Fornecer os conjuntos de treinamento e validação ao encoder e especificar o número de épocas e o tamanho do lote.

4 - Usar o encoder para codificar as imagens de entrada. Isso pode ser feito usando o método predict do encoder e passando as imagens como entrada, resultando em uma representação codificada de menor

In [9]:

```
# Refazer para nao travar a memoria da GPU
import tensorflow as tf
from tensorflow import keras

(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()

x_train = x_train / 255.0
x_test = x_test / 255.0

y_train = keras.utils.to_categorical(y_train, num_classes=10)
y_test = keras.utils.to_categorical(y_test, num_classes=10)

x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)


# Arquitetura do Encoder
encoder_input = keras.Input(shape=(28, 28, 1))
x = keras.layers.Conv2D(32, (3, 3), activation='relu')(encoder_input)
x = keras.layers.MaxPooling2D((2, 2))(x)
x = keras.layers.Conv2D(64, (3, 3), activation='relu')(x)
x = keras.layers.MaxPooling2D((2, 2))(x)
encoder_output = keras.layers.Flatten()(x)

encoder = keras.Model(encoder_input, encoder_output)

# Arquitetura do MLP
mlp_input = keras.Input(shape=(encoder.output_shape[1],))
x = keras.layers.Dense(64, activation='relu')(mlp_input)
x = keras.layers.Dense(32, activation='relu')(x)
mlp_output = keras.layers.Dense(10, activation='softmax')(x)

mlp = keras.Model(mlp_input, mlp_output)

# Juntando o Encoder e o MLP
model = keras.Model(encoder_input, mlp(encoder_output))

# Compilando
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Treinando
model.fit(x_train, y_train, epochs=50)
```

```
Epoch 1/50
1875/1875 [==============================] - 31s 16ms/step - loss: 0.4976
- accuracy: 0.8187
Epoch 2/50
1875/1875 [==============================] - 30s 16ms/step - loss: 0.3258
- accuracy: 0.8818
Epoch 3/50
1875/1875 [==============================] - 29s 15ms/step - loss: 0.2759
- accuracy: 0.8999
Epoch 4/50
1875/1875 [==============================] - 29s 16ms/step - loss: 0.2427
- accuracy: 0.9099
Epoch 5/50
1875/1875 [==============================] - 29s 16ms/step - loss: 0.2186
- accuracy: 0.9191
Epoch 6/50
1875/1875 [==============================] - 30s 16ms/step - loss: 0.1970
- accuracy: 0.9263
Epoch 7/50
1875/1875 [==============================] - 29s 16ms/step - loss: 0.1774
- accuracy: 0.9346
Epoch 8/50
1875/1875 [==============================] - 30s 16ms/step - loss: 0.1604
- accuracy: 0.9411
Epoch 9/50
1875/1875 [==============================] - 30s 16ms/step - loss: 0.1481
- accuracy: 0.9443
Epoch 10/50
1875/1875 [==============================] - 29s 16ms/step - loss: 0.1364
- accuracy: 0.9499
Epoch 11/50
1875/1875 [==============================] - 28s 15ms/step - loss: 0.1236
- accuracy: 0.9543
Epoch 12/50
1875/1875 [==============================] - 29s 16ms/step - loss: 0.1144
- accuracy: 0.9575
Epoch 13/50
1875/1875 [==============================] - 26s 14ms/step - loss: 0.1048
- accuracy: 0.9614
Epoch 14/50
1875/1875 [==============================] - 30s 16ms/step - loss: 0.0985
- accuracy: 0.9629
Epoch 15/50
1875/1875 [==============================] - 30s 16ms/step - loss: 0.0900
- accuracy: 0.9665
Epoch 16/50
1875/1875 [==============================] - 30s 16ms/step - loss: 0.0816
- accuracy: 0.9699
Epoch 17/50
1875/1875 [==============================] - 29s 15ms/step - loss: 0.0799
- accuracy: 0.9698
Epoch 18/50
1875/1875 [==============================] - 28s 15ms/step - loss: 0.0741
- accuracy: 0.9729
Epoch 19/50
1875/1875 [==============================] - 29s 15ms/step - loss: 0.0714
- accuracy: 0.9733
Epoch 20/50
1875/1875 [==============================] - 29s 15ms/step - loss: 0.0638
- accuracy: 0.9757
Epoch 21/50
```

```
1875/1875 [==============================] - 27s 14ms/step - loss: 0.0622
- accuracy: 0.9765
Epoch 22/50
1875/1875 [==============================] - 30s 16ms/step - loss: 0.0577
- accuracy: 0.9787
Epoch 23/50
1875/1875 [==============================] - 30s 16ms/step - loss: 0.0540
- accuracy: 0.9795
Epoch 24/50
1875/1875 [==============================] - 30s 16ms/step - loss: 0.0500
- accuracy: 0.9814
Epoch 25/50
1875/1875 [==============================] - 29s 16ms/step - loss: 0.0510
- accuracy: 0.9809
Epoch 26/50
1875/1875 [==============================] - 28s 15ms/step - loss: 0.0494
- accuracy: 0.9819
Epoch 27/50
1875/1875 [==============================] - 28s 15ms/step - loss: 0.0433
- accuracy: 0.9844
Epoch 28/50
1875/1875 [==============================] - 28s 15ms/step - loss: 0.0439
- accuracy: 0.9843
Epoch 29/50
1875/1875 [==============================] - 29s 15ms/step - loss: 0.0437
- accuracy: 0.9840
Epoch 30/50
1875/1875 [==============================] - 29s 16ms/step - loss: 0.0397
- accuracy: 0.9855
Epoch 31/50
1875/1875 [==============================] - 28s 15ms/step - loss: 0.0413
- accuracy: 0.9850
Epoch 32/50
1875/1875 [==============================] - 29s 15ms/step - loss: 0.0366
- accuracy: 0.9868
Epoch 33/50
1875/1875 [==============================] - 30s 16ms/step - loss: 0.0386
- accuracy: 0.9863
Epoch 34/50
1875/1875 [==============================] - 29s 16ms/step - loss: 0.0344
- accuracy: 0.9874
Epoch 35/50
1875/1875 [==============================] - 29s 15ms/step - loss: 0.0374
- accuracy: 0.9869
Epoch 36/50
1875/1875 [==============================] - 27s 15ms/step - loss: 0.0335
- accuracy: 0.9882
Epoch 37/50
1875/1875 [==============================] - 28s 15ms/step - loss: 0.0375
- accuracy: 0.9867
Epoch 38/50
1875/1875 [==============================] - 28s 15ms/step - loss: 0.0312
- accuracy: 0.9894
Epoch 39/50
1875/1875 [==============================] - 28s 15ms/step - loss: 0.0334
- accuracy: 0.9883
Epoch 40/50
1875/1875 [==============================] - 29s 15ms/step - loss: 0.0311
- accuracy: 0.9891
Epoch 41/50
1875/1875 [==============================] - 28s 15ms/step - loss: 0.0295
```

```
- accuracy: 0.9895
Epoch 42/50
1875/1875 [==============================] - 28s 15ms/step - loss: 0.0297
- accuracy: 0.9895
Epoch 43/50
1875/1875 [==============================] - 29s 16ms/step - loss: 0.0298
- accuracy: 0.9898
Epoch 44/50
1875/1875 [==============================] - 28s 15ms/step - loss: 0.0339
- accuracy: 0.9885
Epoch 45/50
1875/1875 [==============================] - 29s 15ms/step - loss: 0.0267
- accuracy: 0.9901
Epoch 46/50
1875/1875 [==============================] - 29s 15ms/step - loss: 0.0287
- accuracy: 0.9901
Epoch 47/50
1875/1875 [==============================] - 29s 16ms/step - loss: 0.0265
- accuracy: 0.9906
Epoch 48/50
1875/1875 [==============================] - 30s 16ms/step - loss: 0.0267
- accuracy: 0.9909
Epoch 49/50
1875/1875 [==============================] - 30s 16ms/step - loss: 0.0293
- accuracy: 0.9901
Epoch 50/50
1875/1875 [==============================] - 30s 16ms/step - loss: 0.0260
- accuracy: 0.9911
Out[9]:

<keras.callbacks.History at 0x21da2f3db50>
```

Comparativo

In [30]:

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print('Acuracia do teste:', test_acc)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.7093 - ac
curacy: 0.9077
Acuracia do teste: 0.9077000021934509
```

A acuracia de 90%, praticamente igual ao resultado observado na rede neural convolucional. Comparando a matriz de confusao das duas tecnicas, pode-se observar que ambas apresentaram valores baixos apenas para a setima coluna, sendo: 63% para a CNN 74% para o MLP + Encoder

In [31]:

```
from sklearn.metrics import confusion_matrix
y_pred = model.predict(x_test)
y_pred = np.argmax(y_pred, axis=1)

y_test = np.argmax(y_test, axis=-1)

print('Matriz de Confusao')

confusion_matrix(y_test, y_pred)
```

```
313/313 [==============================] - 1s 2ms/step
Matriz de Confusao
```

Out[31]:

```
array([[843,   1,  25,  10,   7,   1, 110,   0,   3,   0],
       [  0, 984,   2,   7,   3,   0,   2,   0,   2,   0],
       [ 19,   0, 850,  10,  64,   2,  55,   0,   0,   0],
       [ 21,   8,   7, 891,  36,   2,  33,   0,   2,   0],
       [  2,   0,  35,  21, 886,   0,  54,   0,   1,   1],
       [  0,   0,   0,   0,   0, 981,   0,  13,   0,   6],
       [ 88,   1,  67,  27,  64,   1, 745,   0,   7,   0],
       [  0,   1,   0,   0,   0,  14,   0, 959,   0,  26],
       [  6,   0,   5,   3,   4,   2,   2,   2, 975,   1],
       [  0,   0,   1,   0,   0,   6,   0,  30,   0, 963]])
```