Aula 2

Callbacks

Eduardo Lobo Lustosa Cabral

1. Introdução e objetivos

Callbacks são ferramentas que permitem personalizar o processo de treinamento de uma rede neural.

Callbacks são funções chamadas em pontos específicos durante o treinamento, como no início de cada época, no final de cada época, ou após cada lote.

Usando callbacks, pode-se monitorar o treinamento, salvar modelos, ajustar hiperparâmetros etc.

Os objetivos dessa aula são os seguintes:

1. Apresentar os principais callbacks
2. Apresentar como criar um callback customizado

Documentação oficial do Keras sobre callbacks: https://keras.io/api/callbacks/

Importando as bibliotecas necessárias

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
tf.__version__
```

```
{"type":"string"}
```

2. Conjunto de dados

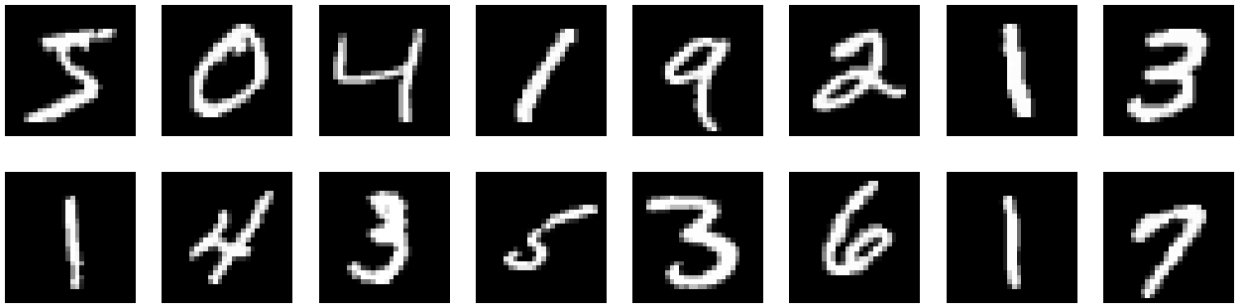
Vamos usar o conjunto de dados de dígitos MNIST, que consiste de imagens em tons de cinza de números de 0 a 9 escritos a mão.

```
(Xtrain, Ytrain), (Xtest, Ytest) = tf.keras.datasets.mnist.load_data()
print('Xtrain.shape: ', Xtrain.shape)
print('Ytrain.shape: ', Ytrain.shape)
print('Xtest.shape: ', Xtest.shape)
print('Ytest.shape: ', Ytest.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Xtrain.shape:  (60000, 28, 28)
Ytrain.shape:  (60000,)
Xtest.shape:  (10000, 28, 28)
Ytest.shape:  (10000,)
```

Normalização dos dados de entrada e codificação one-hot das saídas.

```
# Normalização das entradas
x_train = Xtrain/255.
x_test = Xtest/255.

# Codificação one-hot das saídas
y_train_hot = tf.keras.utils.to_categorical(Ytrain, 10)
y_test_hot = tf.keras.utils.to_categorical(Ytest, 10)

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(Xtrain[index], cmap='gray')
        index += 1
        axs[i,j].axis('off')
plt.show()
```



3. EarlyStopping

O callback EarlyStopping realiza a parada automatica de um treinamento.

O EarlyStopping é um callback útil no treinamento de redes neurais pelos seguintes motivos:

- "Prevenir overfitting": como visto uma forma de prevenir overfitting é parar o treinamento de forma prematura.
- Economizar tempo: é possível evitar treinar por mais épocas do que o necessário, otimizando o uso de recursos computacionais.

- Simplifica a seleção do número de épocas, pois elimina a necessidade de definir manualmente o número ideal de épocas.

Esse callback monitora uma métrica específica durante o treinamento e interrompe o processo quando essa métrica parar de melhorar por um número pré-definido de épocas.

Principais argumentos:

- `monitor`: define o parâmetro do processo de treinamento que é monitorado para realizar a parada. Esse parâmetro pode ser a função de custo ou alguma métrica, tanto dos dados de treinamento como de validação. Opções comuns incluem 'val_loss', 'val_accuracy', 'loss', 'accuracy'.
- `patience`: parâmetro que define o número máximo de épocas que a grandeza escolhida pode permanecer sem melhoria antes que o treinamento seja interrompido.
- `min_delta`: define o que se considera uma melhoria do valor monitorado, esse argumento representa a menor mudança que é considerada significativa.
- `mode`: pode ser 'auto', 'min', ou 'max' → indica se a grandeza monitorada deve ser minimizada ou maximizada.
- `restore_best_weights`: se for `True`, os pesos do modelo serão restaurados para a época com a melhor grandeza monitorada.

Se a métrica não melhorar no mínimo pelo valor definido por `min_delta` pelo número de épocas especificado em `patience`, então, o treinamento é interrompido.

Um exemplo desse callback segue na célula abaixo.

```
from tensorflow.keras.callbacks import EarlyStopping
early_stop = EarlyStopping(monitor='val_accuracy', patience=10,
min_delta=0.001)
```

- Nesse caso está monitorando `val_accuracy` e se ela não diminuir 0,001 por 10 épocas então o treinamento é interrompido

4. ModelCheckpoint

Ele callback permite salvar o modelo durante o processo de treinamento.

Ele permite salvar o modelo em cada época, ou apenas quando uma grandeza específica melhora.

Salvar um modelo é muito importante, por diversos motivos:

1. Recuperar o melhor modelo: no final do treinamento, é possível carregar o modelo salvo com o melhor desempenho, evitando a necessidade de refazer todo o treinamento.

2. Realizar experimentos: pode-se salvar modelos em diferentes pontos do treinamento para comparar resultados e ajustar hiperparâmetros.

3. Continuar o treinamento a partir do último modelo salvo, se o mesmo for interrompido por algum motivo.

Principais argumentos:

- Nome do arquivo: o caminho do arquivo onde o modelo será salvo.
- `monitor`: o callback monitora uma grandeza escolhida, tal como, função de custo ou métrica dos dados de treinamento ou validação.
- `save_best_only`: se for True, salva apenas o melhor modelo até o momento.
- `mode`: pode ser 'auto', 'min', ou 'max' → indica se a grandeza deve ser minimizada ou maximizada.
- `save_weights_only`: se for True, salva apenas os pesos do modelo.
- `verbose`: define se mostra informação quando um modelo é salvo.

Observações:

- Condição de salvamento: quando a métrica satisfazer a condição desejada, por exemplo, atingir o melhor valor até o momento, então, o modelo é salvo em um arquivo.
- Frequência de salvamento: ao definir `save_best_only=False`, o modelo é salvo a cada época.
- Formato do modelo salvo: o formato keras é o atualmente o formato padrão para salvar modelos Keras.

```
from tensorflow.keras.callbacks import ModelCheckpoint
checkpoint = ModelCheckpoint('best_model.keras',
                             monitor='val_accuracy',
                             save_best_only=True,
                             mode='max',
                             verbose=1)
```

- Nesse caso a `val_accuracy`está sendo monitorada e quando ela melhora o modelo é salvo no arquivo 'best_model.keras'. Além disso, todo vez que o modelo é salvo é avisado durante o treinamento.

5. Taxa de aprendizado variável

A variação da taxa de aprendizado durante o treinamento é realizada no TensorFlow/Keras usando o callback **LearningRateScheduler**.

Note que variar a taxa de aprendizado durante o treinamento de uma rede neural pode melhorar a convergência e o desempenho do modelo.

Como vimos, uma taxa de aprendizado muito alta pode levar a oscilações e divergência, enquanto uma taxa muito baixa pode tornar o treinamento muito lento → uma programação de taxa de aprendizado bem definida pode ajudar a encontrar um bom equilíbrio.

O callback LearningRateScheduler recebe uma função que retorna a nova taxa de aprendizado a cada época.

Existem diversas formas de agendar a taxa de aprendizado. Como vimos, algumas opções comuns incluem:

- Decaimento exponencial: a taxa de aprendizado diminui exponencialmente a cada época.
- Decaimento por passo: a taxa de aprendizado é reduzida por um fator fixo a cada certo número de épocas.
- Plateau: a taxa de aprendizado é reduzida quando a métrica de validação para de melhorar.

Na célula abaixo são mostrados dois exemplos de como variar a taxa de aprendizado:

1. Decaimento por passo;
2. Decaimento exponencial.

```
# Importa callback de programaçãp da taxa de aprendizado
from tensorflow.keras.callbacks import LearningRateScheduler

# Função para variar a taxa de aprendizado por passo
def step_scheduler(epoch):
    if epoch < 10:
        return 0.01
    else:
        return 0.001

# Criando o callback
lr_scheduler_step = LearningRateScheduler(step_scheduler)

# Função para variar a taxa de aprendizado exponencialmente
def exponential_decay(lr0, s):
    def decay(epoch):
        return lr0 * 0.1**(epoch / s)
    return decay

# Criando o callback
lr_scheduler_exp = LearningRateScheduler(exponential_decay(0.01, 20))
```

- Função `step_scheduler`: define como a taxa de aprendizado será atualizada a cada época. No exemplo, a taxa é 0.01 para as primeiras 10 épocas e 0.001 para as demais.

- Função `exponential_decay`: define como é realizado o decaimento exponencial da taxa de aprendizado; `lr0` é a taxa de aprendizado inicial e `s` é uma constante usada para definir a velocidade da diminuição da taxa.

- `LearningRateScheduler`: cria um objeto callback que passa a função scheduler para o otimizador.

6. Exemplo

Vamos treinar um modelo simples usando esses 3 callbacks que configurados.

Vamos utilizar um modelo com camadas densas para resolver o problema de classificação multiclasse dos digitos MNIST.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten

# Modelo sequencial
def build_model(input_shape):
    model = Sequential()
    model.add(Flatten(input_shape=input_shape))
    model.add(Dense(256, activation='relu'))
    model.add(Dense(128, activation='relu'))
    model.add(Dense(10, activation='softmax'))
    return model

# Cria modelo
model = build_model(input_shape=(28, 28))

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

model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/reshaping/
flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_dim`
argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

```
Model: "sequential"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃ Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ flatten (Flatten)                    │ (None, 784)                 │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense (Dense)                        │ (None, 256)                 │         200,960 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
```

```
──────────────────┤
│ dense_1 (Dense)                      │ (None, 128)                 │
32,896 │
├──────────────────────────────────────┼─────────────────────────────┼
──────────────────┤
│ dense_2 (Dense)                      │ (None, 10)                  │
1,290 │
└──────────────────────────────────────┴─────────────────────────────┴
──────────────────┘

 Total params: 235,146 (918.54 KB)

 Trainable params: 235,146 (918.54 KB)

 Non-trainable params: 0 (0.00 B)
```

Os callbacks desejados são adicionados à lista de callbacks durante o treinamento.

```
# Treinando o modelo
history = model.fit(x_train, y_train_hot,
                    epochs=100,
                    validation_data=(x_test, y_test_hot),
                    callbacks=[early_stop, checkpoint,
lr_scheduler_exp])

Epoch 1/100
1848/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8821 - loss:
0.3924
Epoch 1: val_accuracy improved from -inf to 0.94720, saving model to
best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 12s 3ms/step - accuracy: 0.8826 - loss:
0.3907 - val_accuracy: 0.9472 - val_loss: 0.1753 - learning_rate:
0.0100
Epoch 2/100
1856/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9551 - loss:
0.1589
Epoch 2: val_accuracy improved from 0.94720 to 0.95460, saving model
to best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9551 - loss:
0.1590 - val_accuracy: 0.9546 - val_loss: 0.1735 - learning_rate:
0.0089
Epoch 3/100
1874/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9623 - loss:
0.1345
Epoch 3: val_accuracy improved from 0.95460 to 0.96720, saving model
to best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9623 - loss:
0.1345 - val_accuracy: 0.9672 - val_loss: 0.1306 - learning_rate:
0.0079
Epoch 4/100
```

```
1869/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9715 - loss:
0.1013
Epoch 4: val_accuracy did not improve from 0.96720
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9715 - loss:
0.1014 - val_accuracy: 0.9561 - val_loss: 0.2102 - learning_rate:
0.0071
Epoch 5/100
1874/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9743 - loss:
0.0930
Epoch 5: val_accuracy improved from 0.96720 to 0.96800, saving model
to best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9743 - loss:
0.0930 - val_accuracy: 0.9680 - val_loss: 0.1398 - learning_rate:
0.0063
Epoch 6/100
1870/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9787 - loss:
0.0737
Epoch 6: val_accuracy improved from 0.96800 to 0.96900, saving model
to best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.9787 - loss:
0.0737 - val_accuracy: 0.9690 - val_loss: 0.1261 - learning_rate:
0.0056
Epoch 7/100
1854/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9839 - loss:
0.0587
Epoch 7: val_accuracy did not improve from 0.96900
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9839 - loss:
0.0587 - val_accuracy: 0.9643 - val_loss: 0.1761 - learning_rate:
0.0050
Epoch 8/100
1868/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9835 - loss:
0.0612
Epoch 8: val_accuracy improved from 0.96900 to 0.96960, saving model
to best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9835 - loss:
0.0612 - val_accuracy: 0.9696 - val_loss: 0.1407 - learning_rate:
0.0045
Epoch 9/100
1873/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9874 - loss:
0.0417
Epoch 9: val_accuracy improved from 0.96960 to 0.97340, saving model
to best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9874 - loss:
0.0417 - val_accuracy: 0.9734 - val_loss: 0.1369 - learning_rate:
0.0040
Epoch 10/100
1852/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9897 - loss:
0.0331
Epoch 10: val_accuracy did not improve from 0.97340
```

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9897 - loss:
0.0332 - val_accuracy: 0.9725 - val_loss: 0.1446 - learning_rate:
0.0035
Epoch 11/100
1858/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9919 - loss:
0.0277
Epoch 11: val_accuracy did not improve from 0.97340
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9919 - loss:
0.0277 - val_accuracy: 0.9712 - val_loss: 0.1660 - learning_rate:
0.0032
Epoch 12/100
1849/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9933 - loss:
0.0240
Epoch 12: val_accuracy improved from 0.97340 to 0.97520, saving model
to best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9933 - loss:
0.0240 - val_accuracy: 0.9752 - val_loss: 0.1463 - learning_rate:
0.0028
Epoch 13/100
1874/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9947 - loss:
0.0178
Epoch 13: val_accuracy improved from 0.97520 to 0.97540, saving model
to best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9947 - loss:
0.0178 - val_accuracy: 0.9754 - val_loss: 0.1596 - learning_rate:
0.0025
Epoch 14/100
1869/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9960 - loss:
0.0135
Epoch 14: val_accuracy improved from 0.97540 to 0.97730, saving model
to best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9960 - loss:
0.0135 - val_accuracy: 0.9773 - val_loss: 0.1567 - learning_rate:
0.0022
Epoch 15/100
1852/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9964 - loss:
0.0106
Epoch 15: val_accuracy did not improve from 0.97730
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9964 - loss:
0.0106 - val_accuracy: 0.9773 - val_loss: 0.1724 - learning_rate:
0.0020
Epoch 16/100
1857/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9970 - loss:
0.0092
Epoch 16: val_accuracy improved from 0.97730 to 0.97740, saving model
to best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9970 - loss:
0.0092 - val_accuracy: 0.9774 - val_loss: 0.1789 - learning_rate:
0.0018
```

```
Epoch 17/100
1861/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9976 - loss:
0.0079
Epoch 17: val_accuracy did not improve from 0.97740
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9976 - loss:
0.0079 - val_accuracy: 0.9764 - val_loss: 0.1777 - learning_rate:
0.0016
Epoch 18/100
1859/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9985 - loss:
0.0055
Epoch 18: val_accuracy did not improve from 0.97740
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9985 - loss:
0.0055 - val_accuracy: 0.9768 - val_loss: 0.2055 - learning_rate:
0.0014
Epoch 19/100
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9987 - loss:
0.0045
Epoch 19: val_accuracy did not improve from 0.97740
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9987 - loss:
0.0045 - val_accuracy: 0.9769 - val_loss: 0.2246 - learning_rate:
0.0013
Epoch 20/100
1853/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9989 - loss:
0.0036
Epoch 20: val_accuracy improved from 0.97740 to 0.97770, saving model
to best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9989 - loss:
0.0036 - val_accuracy: 0.9777 - val_loss: 0.2180 - learning_rate:
0.0011
Epoch 21/100
1860/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9992 - loss:
0.0029
Epoch 21: val_accuracy did not improve from 0.97770
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9992 - loss:
0.0029 - val_accuracy: 0.9775 - val_loss: 0.2391 - learning_rate:
0.0010
Epoch 22/100
1863/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9995 - loss:
0.0024
Epoch 22: val_accuracy did not improve from 0.97770
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9995 - loss:
0.0024 - val_accuracy: 0.9774 - val_loss: 0.2286 - learning_rate:
8.9125e-04
Epoch 23/100
1860/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9998 - loss:
0.0011
Epoch 23: val_accuracy improved from 0.97770 to 0.97820, saving model
to best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9998 - loss:
```

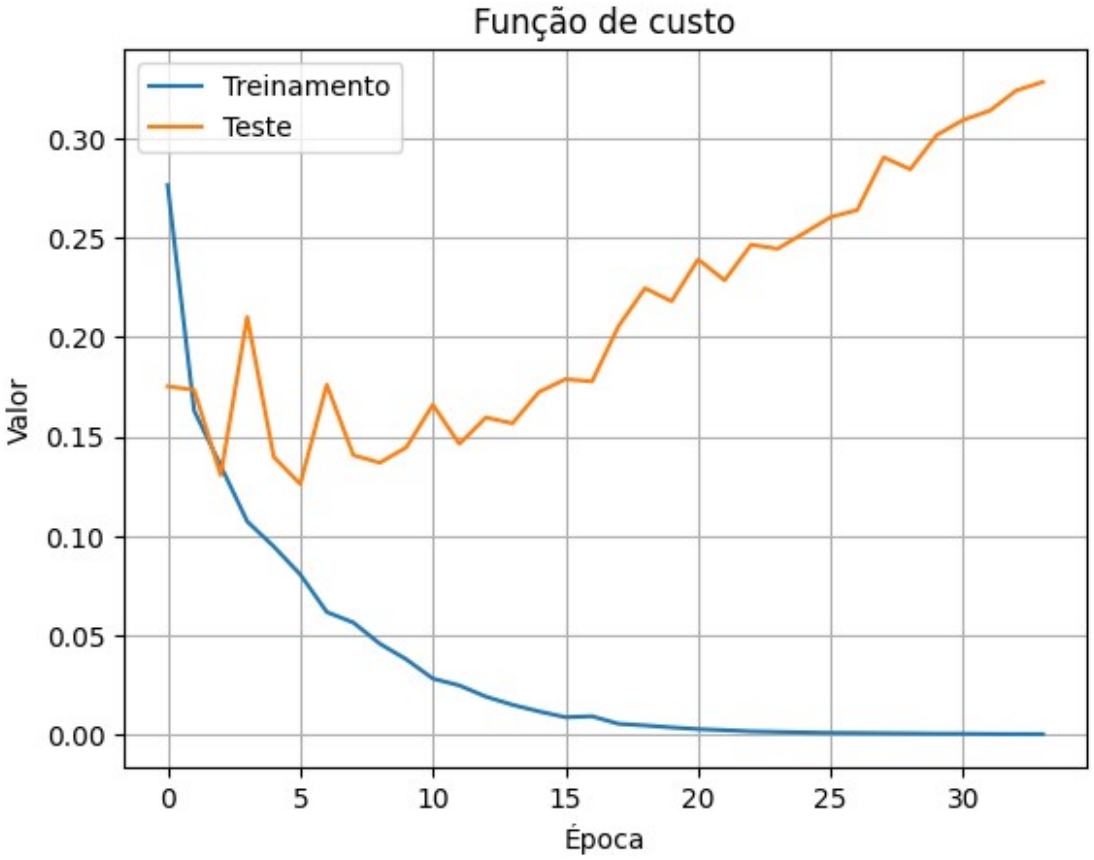
```
0.0011 - val_accuracy: 0.9782 - val_loss: 0.2465 - learning_rate:
7.9433e-04
Epoch 24/100
1849/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9994 - loss:
0.0017
Epoch 24: val_accuracy improved from 0.97820 to 0.97880, saving model
to best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9994 - loss:
0.0017 - val_accuracy: 0.9788 - val_loss: 0.2445 - learning_rate:
7.0795e-04
Epoch 25/100
1859/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9997 - loss:
9.4103e-04
Epoch 25: val_accuracy did not improve from 0.97880
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9997 - loss:
9.4368e-04 - val_accuracy: 0.9786 - val_loss: 0.2524 - learning_rate:
6.3096e-04
Epoch 26/100
1860/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9997 - loss:
9.3625e-04
Epoch 26: val_accuracy did not improve from 0.97880
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9997 - loss:
9.3699e-04 - val_accuracy: 0.9786 - val_loss: 0.2604 - learning_rate:
5.6234e-04
Epoch 27/100
1853/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9997 - loss:
9.5187e-04
Epoch 27: val_accuracy did not improve from 0.97880
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.9997 - loss:
9.5220e-04 - val_accuracy: 0.9785 - val_loss: 0.2639 - learning_rate:
5.0119e-04
Epoch 28/100
1868/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9997 - loss:
8.1462e-04
Epoch 28: val_accuracy improved from 0.97880 to 0.97930, saving model
to best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9997 - loss:
8.1478e-04 - val_accuracy: 0.9793 - val_loss: 0.2904 - learning_rate:
4.4668e-04
Epoch 29/100
1856/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9996 - loss:
9.9134e-04
Epoch 29: val_accuracy did not improve from 0.97930
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9996 - loss:
9.8872e-04 - val_accuracy: 0.9780 - val_loss: 0.2844 - learning_rate:
3.9811e-04
Epoch 30/100
1867/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9997 - loss:
5.8775e-04
```

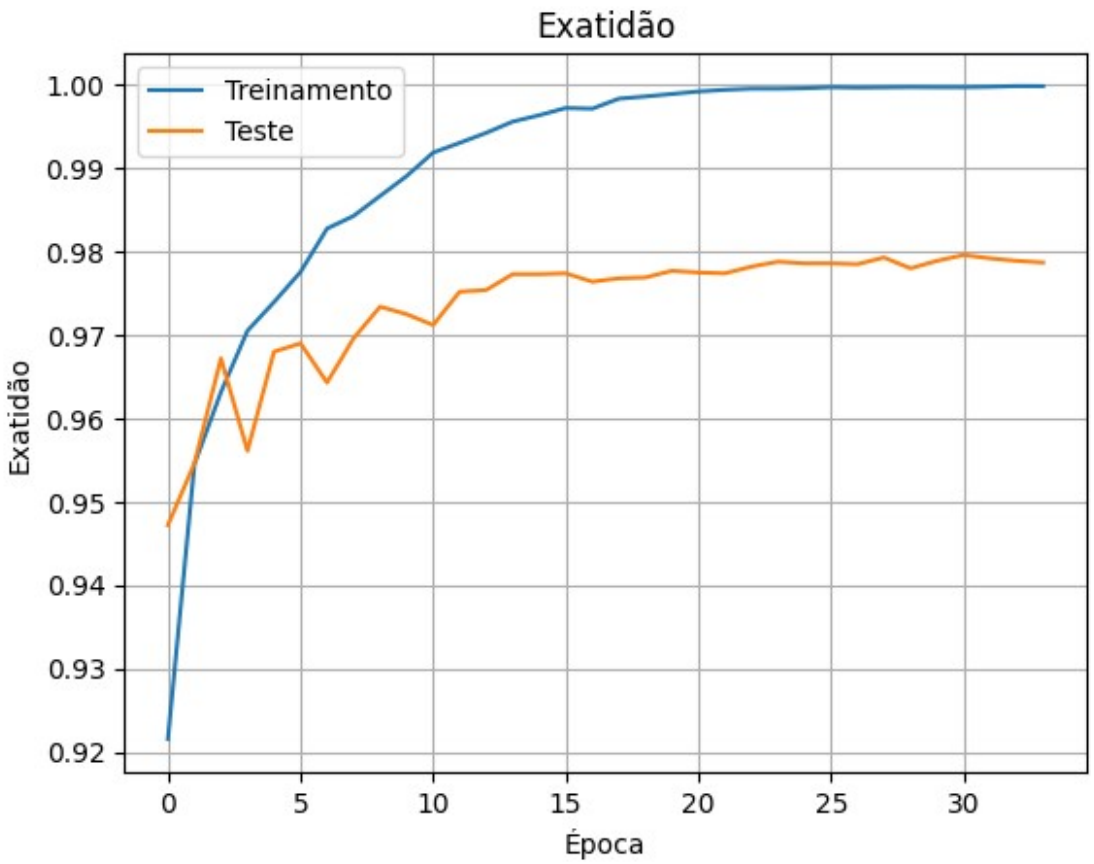
```
Epoch 30: val_accuracy did not improve from 0.97930
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9997 - loss:
5.8794e-04 - val_accuracy: 0.9789 - val_loss: 0.3016 - learning_rate:
3.5481e-04
Epoch 31/100
1844/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9997 - loss:
5.3209e-04
Epoch 31: val_accuracy improved from 0.97930 to 0.97960, saving model
to best_model.keras
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9997 - loss:
5.3336e-04 - val_accuracy: 0.9796 - val_loss: 0.3093 - learning_rate:
3.1623e-04
Epoch 32/100
1856/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9996 - loss:
7.0202e-04
Epoch 32: val_accuracy did not improve from 0.97960
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.9996 - loss:
6.9974e-04 - val_accuracy: 0.9792 - val_loss: 0.3139 - learning_rate:
2.8184e-04
Epoch 33/100
1868/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9998 - loss:
5.7650e-04
Epoch 33: val_accuracy did not improve from 0.97960
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9998 - loss:
5.7599e-04 - val_accuracy: 0.9789 - val_loss: 0.3240 - learning_rate:
2.5119e-04
Epoch 34/100
1845/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9999 - loss:
4.8759e-04
Epoch 34: val_accuracy did not improve from 0.97960
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9999 - loss:
4.8649e-04 - val_accuracy: 0.9787 - val_loss: 0.3283 - learning_rate:
2.2387e-04
```

Visualizando os Resultados

```
# Função de custo
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Função de custo')
plt.ylabel('Valor')
plt.xlabel('Época')
plt.legend(['Treinamento', 'Teste'], loc='upper left')
plt.grid()
plt.show()

# Métrica
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
```

```
plt.title('Exatidão')
plt.ylabel('Exatidão')
plt.xlabel('Época')
plt.legend(['Treinamento', 'Teste'], loc='upper left')
plt.grid()
plt.show()
```






```
rna = tf.keras.models.load_model('best_model.keras')
rna.summary()
```

```
Model: "sequential"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ flatten (Flatten)                    │ (None, 784)                 │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense (Dense)                        │ (None, 256)                 │         200,960 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_1 (Dense)                      │ (None, 128)                 │          32,896 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
```

```
| dense_2 (Dense)                         | (None, 10)                  |
1,290 |
└─────────────────────────────────────────┴─────────────────────────────┴
──────────────────┘

 Total params: 705,440 (2.69 MB)

 Trainable params: 235,146 (918.54 KB)

 Non-trainable params: 0 (0.00 B)

 Optimizer params: 470,294 (1.79 MB)
```

7. Callbacks personalizados

É possível criar callbacks customizados para tarefas mais específicas. Por exemplo, pode-se criar um callback para salvar imagens geradas por um modelo genartivo ou para enviar notificações por e-mail quando o treinamento terminar.

Para criar um callback customizado tem-se que criar uma classe que herda a classe base Callback do Keras e implementar os métodos que se deseja executar em momentos específicos do treinamento.

Existem alguns métodos disponíveis em um Callback que podem utilizados para executar ações em diferentes momentos do treinamento:

- `on_train_begin`: chamado no início do treinamento.
- `on_train_end`: chamado no final do treinamento.
- `on_epoch_begin`: chamado no início de cada época.
- `on_epoch_end`: chamado no final de cada época.
- `on_batch_begin`: chamado no início de cada lote.
- `on_batch_end`: chamado no final de cada lote.

Nas células a seguir seguem alguns exemplos.

7.1 Salvar pesos a cada 10 épocas

O callback da célula abaixo salva os pesos do modelo durante o treinamento a cada `period` épocas.

```
# Importa classe Callback
from tensorflow.keras.callbacks import Callback

# Cria classe de callback para salvar pesos
class SaveWeightsEveryN(Callback):
    def __init__(self, filepath, period):
        super(SaveWeightsEveryN, self).__init__()
        self.filepath = filepath
        self.period = period
```

```
    def on_epoch_end(self, epoch, logs=None):
        if epoch % self.period == 0:
            self.model.save_weights(self.filepath.format(epoch=epoch))

# Cria modelo
model = build_model(input_shape=(28, 28))

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

model.summary()

# Cria objeto de callback
callback_save_weights =
SaveWeightsEveryN(filepath='{epoch:02d}.weights.h5', period=10)

# Trein modelo com o callback
history = model.fit(x_train, y_train_hot, epochs=31,
callbacks=[callback_save_weights])
```

```
Model: "sequential_1"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ flatten_1 (Flatten)                  │ (None, 784)                 │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_3 (Dense)                      │ (None, 256)                 │         200,960 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_4 (Dense)                      │ (None, 128)                 │          32,896 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_5 (Dense)                      │ (None, 10)                  │           1,290 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────┘
```

```
 Total params: 235,146 (918.54 KB)

 Trainable params: 235,146 (918.54 KB)
```

```
 Non-trainable params: 0 (0.00 B)

Epoch 1/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8926 - loss:
0.3682
Epoch 2/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9728 - loss:
0.0873
Epoch 3/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9816 - loss:
0.0565
Epoch 4/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9871 - loss:
0.0398
Epoch 5/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9893 - loss:
0.0327
Epoch 6/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9924 - loss:
0.0236
Epoch 7/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9930 - loss:
0.0201
Epoch 8/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9942 - loss:
0.0189
Epoch 9/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9948 - loss:
0.0163
Epoch 10/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.9959 - loss:
0.0119
Epoch 11/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9961 - loss:
0.0124
Epoch 12/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9956 - loss:
0.0129
Epoch 13/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9975 - loss:
0.0081
Epoch 14/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9953 - loss:
0.0136
Epoch 15/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9966 - loss:
0.0106
Epoch 16/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.9971 - loss:
0.0094
```

```
Epoch 17/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9965 - loss:
0.0115
Epoch 18/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9967 - loss:
0.0099
Epoch 19/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9974 - loss:
0.0088
Epoch 20/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9977 - loss:
0.0090
Epoch 21/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9976 - loss:
0.0081
Epoch 22/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9974 - loss:
0.0077
Epoch 23/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9975 - loss:
0.0087
Epoch 24/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9982 - loss:
0.0052
Epoch 25/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9980 - loss:
0.0065
Epoch 26/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9977 - loss:
0.0069
Epoch 27/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.9979 - loss:
0.0085
Epoch 28/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9984 - loss:
0.0065
Epoch 29/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9979 - loss:
0.0075
Epoch 30/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9975 - loss:
0.0089
Epoch 31/31
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9978 - loss:
0.0074
```

- A classe SaveWeightsEveryN herda as propriedades da classe base Callback.

- Inicialização: no construtor são definidos o caminho do arquivo para salvar os pesos e o período em que os pesos são salvos.

7.2 Salvar previsões intermediárias

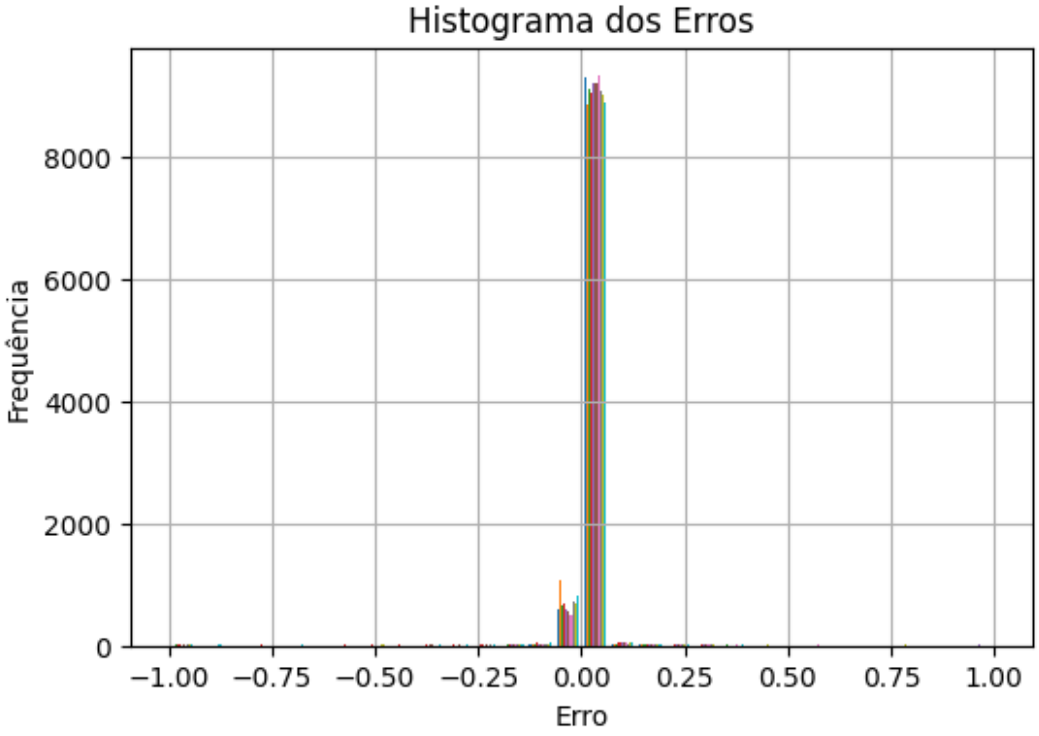
O callback da célula abaixo realiza as seguintes funções:

1. Calcula previsões e métricas no final de algumas épocas (definido por `save_freq`);
2. Calcula uma métrica;
3. Cria um histograma do erro entre as prervisões e as saídas desejadas;
4. Salva gráficos do histograma e das previsões.

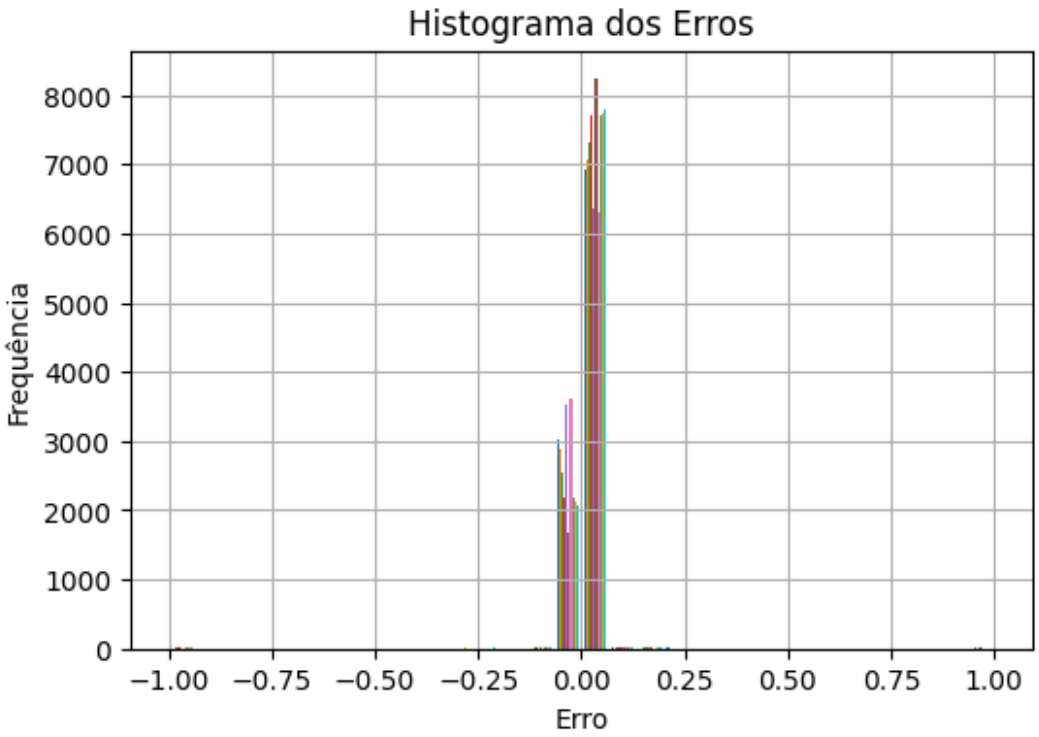
```
class SaveAndPlotPredictions(Callback):
    def __init__(self, x_val, y_val, save_freq=5,
save_path='predictions'):
        super(SaveAndPlotPredictions).__init__()
        self.x_val = x_val
        self.y_val = y_val
        self.save_path = save_path
        self.save_freq = save_freq

    def on_epoch_end(self, epoch, logs=None):
        if epoch % self.save_freq == 0:
            # Fazer previsões no conjunto de teste
            predictions = self.model.predict(self.x_val)

            # Salve as previsões em arquivo
            np.save(f"predictions_epoch_{epoch}.npy", predictions)

            # Calcule métricas de interesse (ex: accuracy, MSE)
            val_metrics = logs.get('val_accuracy')
            print(f"Epoch {epoch}: {val_metrics}")

            # Plotar um histograma da diferença entre as previsões e
os valores reais
            plt.figure(figsize=(6, 4))
            plt.hist(predictions - self.y_val, bins=30)
            plt.xlabel('Erro')
            plt.ylabel('Frequência')
            plt.title('Histograma dos Erros')
            plt.savefig('error_histogram_epoch'+str(epoch)+'.png')
            plt.grid()
            plt.show()

            # Salve um gráfico comparando as previsões com os valores
reais
            plt.figure(figsize=(10, 5))
            plt.plot(np.round(predictions[:100]), 'bo')#,
label='Predictions')
            plt.plot(self.y_val[:100], 'ro')#, label='True Values')
```

```
            plt.grid()
            #plt.legend()
            plt.savefig(self.save_path+'_epoch_'+str(epoch)+'.png')
            plt.close()

# Cria modelo
model = build_model(input_shape=(28, 28))

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

# Cria callback
callback_saveplot = SaveAndPlotPredictions(x_test, y_test_hot,
save_freq=5)

# Treina modelo
model.fit(x_train, y_train_hot,
          epochs=20,
          validation_data=(x_test, y_test_hot),
          callbacks=[callback_saveplot],verbose=0)
```

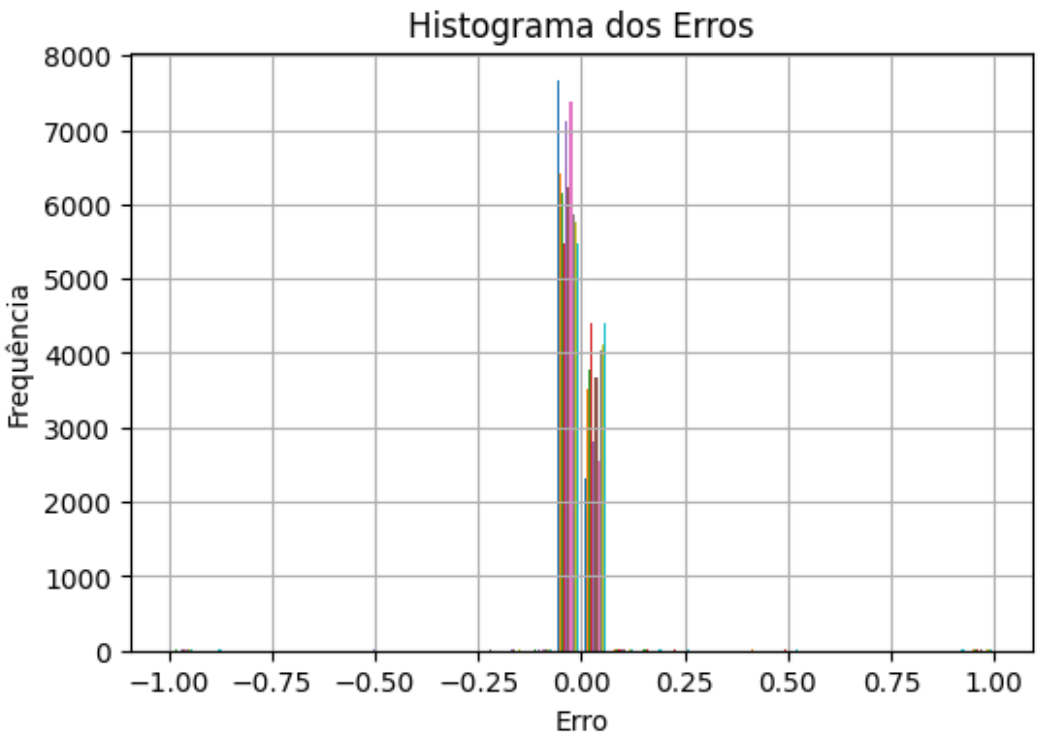
```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Epoch 0: 0.9697999954223633
```



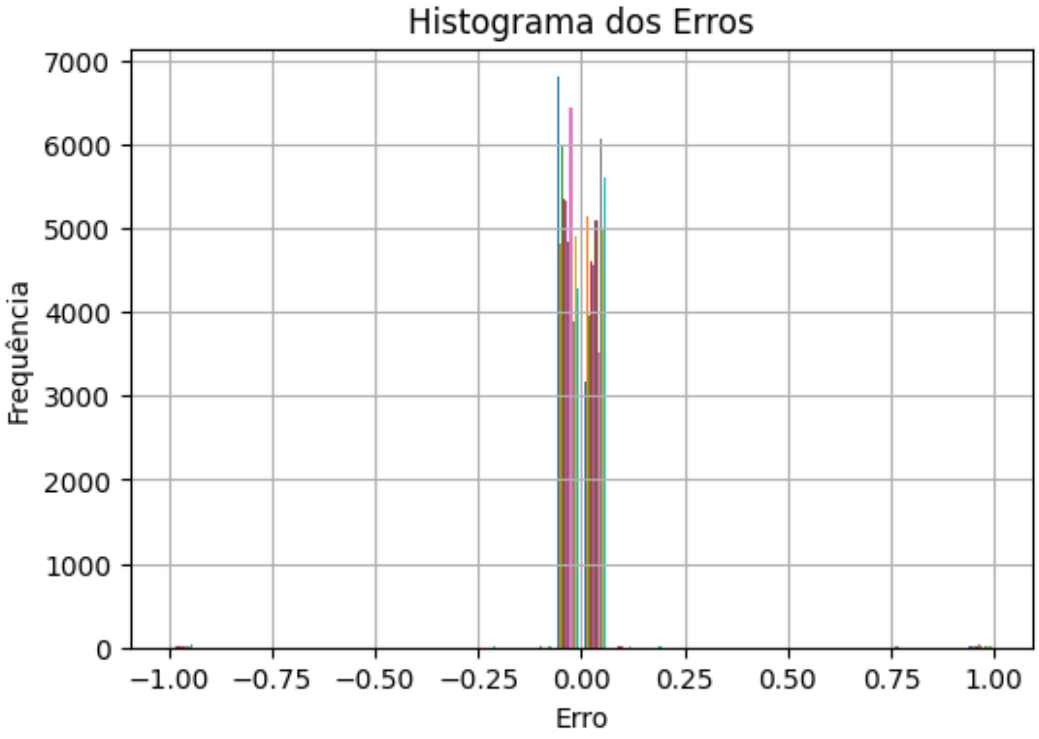
```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Epoch 5: 0.9811999797821045
```




```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Epoch 10: 0.978600025177002
```




```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Epoch 15: 0.9782999753952026
```



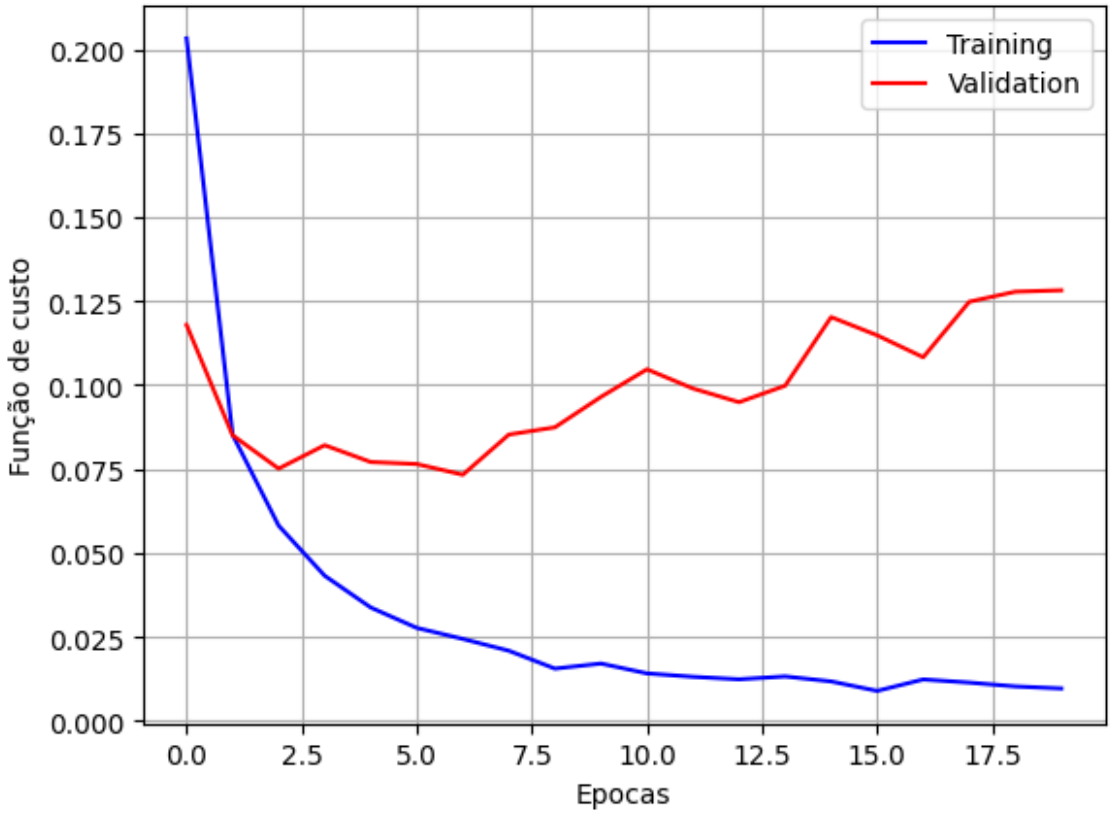
```
<keras.src.callbacks.history.History at 0x7c2d13d95c90>
```

- Inicialização: o construtor recebe o modelo, os dados de validação, o caminho para salvar os gráficos e a frequência de salvamento.
- Método `on_epoch_end`: a cada `save_freq` épocas realiza:
  - Faz previsões no conjunto de validação;
  - Calcula métrica de interesse;
  - Visualiza as previsões;
  - Salva as previsões em um arquivo NumPy;
  - Salva um gráfico comparando as previsões com os valores reais.

7.3 Gráfico do processo de treinamento

```
# Importa função para apagar saída
from IPython.display import clear_output

# Cria calsse para fazer gráfico durante treinamento
class PlotLearningCurves(Callback):
    def __init__(self):
        self.x = []
        self.losses = []
        self.val_losses = []

    def on_epoch_end(self, epoch, logs={}):
        # Atualiza listas
        self.x.append(int(epoch))
        self.losses.append(logs.get('loss'))
        self.val_losses.append(logs.get('val_loss'))

        # Apaga gráfico anterior
        clear_output()

        # Faz o gráfico
        plt.figure()
        plt.plot(self.x, self.losses, 'b', label='Training')
        plt.plot(self.x, self.val_losses, 'r', label='Validation')
        plt.xlabel('Epocas')
        plt.ylabel('Função de custo')
        plt.legend()
        plt.grid()
        plt.show()

# Cria modelo
model = build_model(input_shape=(28, 28))

# Compila modelo
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
```

```
              metrics=['accuracy'])

# Cria callback
plot_callbacks = PlotLearningCurves()

# Treina modelo
model.fit(x_train, y_train_hot,
          epochs=20,
          validation_data=(x_test, y_test_hot),
          callbacks=[plot_callbacks], verbose=0)
```




```
<keras.src.callbacks.history.History at 0x7c2cd5d7b250>
```

- Inicialização: inicializa as listas para armazenar as épocas e as perdas.
- Método `on_epoch_end`: Atualiza os dados do gráfico com a perda de treinamento e validação da época atual e redesenha o gráfico.