Aula 16

Modelos customizados

Eduardo Lobo Lustosa Cabral

1. Objetivos

Apresentar os motivos e os benefícios de usar um modelo customizado no lugar dos modelos sequenciais e funcionais.

Apresentar como criar um modelo customizado na forma de classe com a classe sendo herdada do `ModelClass` do TensorFlow.

Apresentar como criar uma RNA residual com ciclo usando uma classe de modelo customizado.

Importação das bilbiotecas básicas

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

```
2.17.1
```

2. Introdução

Nas aulas anteriores vimos duas formas de criar modelos com o TensorFlow/Keras (modelos sequenciais e modelos funcionai).

Um modelo customizado na forma de classe pode possuir funcionalidades que não existem nas classes de modelos sequencicais e funcionais.

As vantagens de criar modelos na forma de classe são inúmeras, entre elas destacam-se:

- Permitir o reuso do modelo;
- Compor modelos complexos em uma única solução para um problema complexo;
- Criar modelos de forma mais estruturada;
- A definição das camadas é separada do modelo, o que permite realizar cálculos complexos, tais como, ciclos, camadas múltiplas, comandos condicionais etc;
- Permite criar sub-modelos, que são utilizados em grandes modelos.

Veremos alguns exemplos de modelos customizados que ou são difícieis, ou mesmo impossíveis, de serem criados com a classe Funcional do keras.

3. Classe de modelos customizados

Para definir uma classe de modelo customizado, a forma mais fácil é criar uma classe que herda as propriedades de modelos do Keras, que é a classe `Model`.

Criar um modelo com a classe `Model` tem muitos benefícios:

1. Em razão da classe customizada herdar as propriedades da classe `Model`, todos os métodos existentes dessa classe são válidos para o modelo customizado, tais como: `model.fit`, `model.evaluate` e `model.predict`.

2. Métodos para salvar modelos, `model.save()` e `model.save_weights()` podem ser usados.

3. Métodos para apresentar modelos, `model.summary()` e `plot_model()` podem ser usados, porém para usar essas funções precisa de códificação adicional.

4. Limitações dos modelos sequenciais e funcionais são eliminadas:

    - Somente modelos com ligações diretas ou ramificações, são permitidos com a classe funcional;
    - Ciclos (loops) não são permitidos com a classe funcional;
    - Nas classes sequencial e funcional a direção do fluxo de informação é sempre para frente, nunca retorna;
    - A classe Model permite criar arquiteturas exóticas, tais como, recursão, alteração de modelo durante o treinamento ou a inferência.

5. Um modelo customizado com a classe Model não é uma alteração muito drástica em relação aos modelos sequenciais e funcionais → continua-se usando os comandos e funções já vistos.

6. Permite criar arquiteturas muito complexas.

3.1 Classe `Model`

Uma classe de modelo customizado deve ter pelo menos dois métodos:

1. `__init()`: recebe os parâmetros necessários para inicializar o modelo
2. `call()`: recebe as entradas do modelo e retorna as saídas calculadas

Um terceiro método (`build`) pode ser incluído se for desejado criar o modelo ao instanciar um objeto da classe.

Os outros métodos da classe Model, tais como, `fit` e `evaluate` também podem ser modificados.

Como exemplo de criar um modelo customizado, vamos criar o modelo mostrado na Figura 1. Esse modelo tem 2 entradas e duas saídas. A primeira saída é o resultado do processamento da junção das duas entradas e a outra saída é o resultado do processamento de uma das entradas.

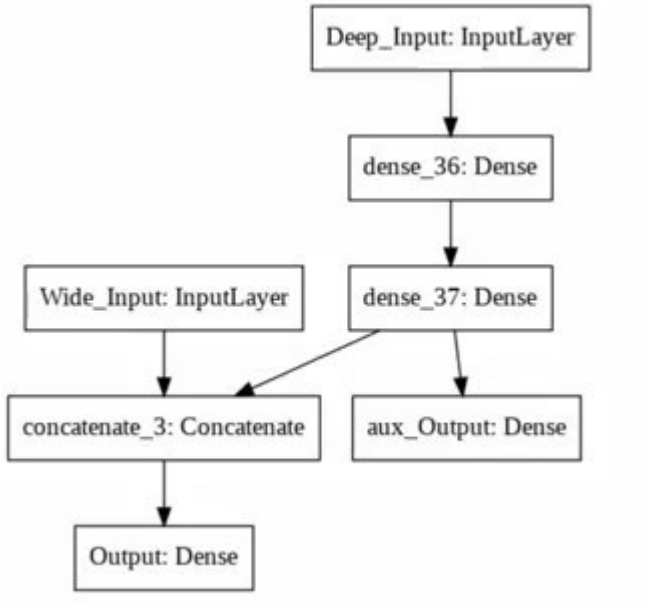


Figura 1 - Exemplo de um modelo simples customizado.

```
# Importa classes de modelo e camadas densa e concatenate
from tensorflow.keras.layers import Dense, concatenate
from tensorflow.keras.models import Model

# Modelo simples customizado na forma de classe
class TwoInputModel(tf.keras.Model):

    # Função de inicialização
    def __init__(self, units=1, activation=None, **kwargs):
        # Incializa classe
        super(TwoInputModel, self).__init__(**kwargs)

        # Cria instâncias das camadas desejadas no modelo
        self.dense1 = Dense(units, activation=activation)
        self.dense2 = Dense(units, activation=activation)
        self.out1 = Dense(1, name='Output')
        self.out2 = Dense(1, name='Aux_Output')

    # Define cálculos a serem realizados na rede. Deve usar camadas
incializadas na função __init__()
    def call(self, inputs):
        # inputs = entrada do modelo, que no caso é uma lista com as
duas entradas
        input1, input2 = inputs

        # Camadas iniciais
```

```
        x1 = self.dense1(input2)
        x2 = self.dense2(x1)

        # Junta entrada 1 com saída da camaada dense3
        concat = concatenate([input1, x2])

        # Calcula saída 1
        y1 = self.out1(concat)

        # Define saída 2
        y2 = self.out2(x2)

        return y1, y2

# Cria modelo da classe TwoInputModel
model = TwoInputModel(units=10, activation='relu')
print(model)
```

```
<TwoInputModel name=two_input_model, built=False>
```

```
model.summary()
```

```
Model: "two_input_model"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━
┓
┃ Layer (type)                         ┃ Output Shape                ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━
┩
│ dense (Dense)                        │ ?                           │
0 (unbuilt) │
├──────────────────────────────────────┼─────────────────────────────┼─
──────────────┤
│ dense_1 (Dense)                      │ ?                           │
0 (unbuilt) │
├──────────────────────────────────────┼─────────────────────────────┼─
──────────────┤
│ Output (Dense)                       │ ?                           │
0 (unbuilt) │
├──────────────────────────────────────┼─────────────────────────────┼─
──────────────┤
│ Aux_Output (Dense)                   │ ?                           │
0 (unbuilt) │
└──────────────────────────────────────┴─────────────────────────────┴─
──────────────┘
```

```
 Total params: 0 (0.00 B)
```

```
 Trainable params: 0 (0.00 B)
```

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

Observe que a dimensão dos tensores das saídas das camadas não estão definidas. Para apresentar o sumário do modelo de forma correta deve-se executá-lo pelo menos uma vez com as entradas desejadas, ou criar o modelo com o método `build` que pertence à `ClassModel`.

Assim, vamos executar o modelo com duas entradas fictícias, como feito na célula a seguir.

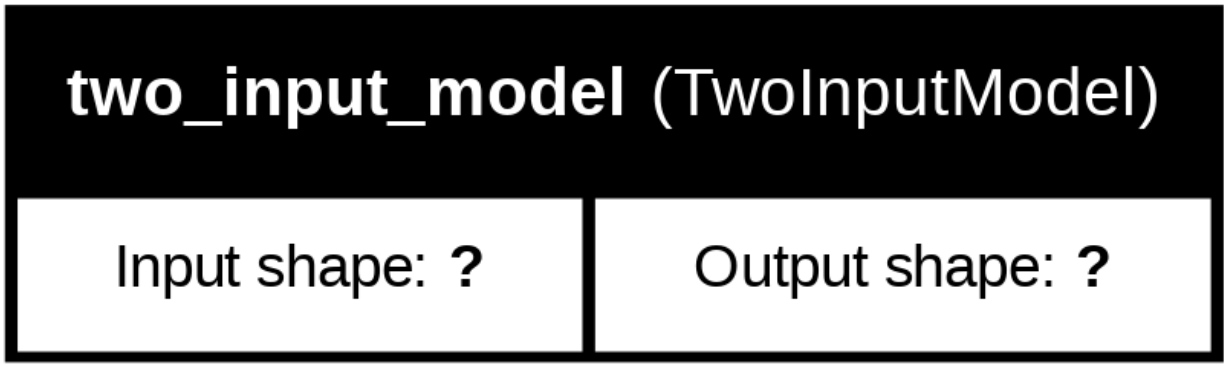
```
# Define entradas
x1 = tf.ones((1,16))
x2 = tf.ones((1,24))

# Executa modelo
y1, y2 = model([x1, x2])

# Mostra  saídas
print('y=', y1)
print('y2=', y2,'\n')

# Mostra sumario do modelo
model.summary()
```

```
y= tf.Tensor([[-0.25266826]], shape=(1, 1), dtype=float32)
y2= tf.Tensor([[0.89538205]], shape=(1, 1), dtype=float32)


Model: "two_input_model"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ dense (Dense)                        │ (1, 10)                     │             250 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_1 (Dense)                      │ (1, 10)                     │             110 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ Output (Dense)                       │ (1, 1)                      │              27 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ Aux_Output (Dense)                   │ (1, 1)                      │              11 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────┘
```

```
 Total params: 398 (1.55 KB)

 Trainable params: 398 (1.55 KB)

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

```
# Mostra diagrama do modelo
from tensorflow.keras.utils import plot_model

plot_model(model, show_shapes=True, show_layer_names=True,
expand_nested=True)
```



3.2 Modelo customizado "completo"

Observa-se que o sumário do modelo não apresenta quase nenhuma informação, somente as camadas existentes, e o diagrama do modelo não mostra de fato o modelo → isso é devido ao fato de que quando se cria uma instância de uma classe de modelo, ele é transformado em um código para executar o modelo, onde não é possível obter informações detalhadas dos seus componentes.

Uma forma de se contornar isso é inicializar a classe após definir seus componentes no método `__init__()`.

Nota-se que o método `build` pertence à classe `Model` e serve para criar o modelo após a criação de uma instaância da classe.

```
# Importa classes de modelo e camadas
from tensorflow.keras import layers
from tensorflow.keras.layers import Dense, concatenate, Input
from tensorflow.keras.models import Model

# Modelo simples customizado na forma de classe
class TwoInputModel2(tf.keras.Model):

    # Função de inicialização
    def __init__(self, dim1, dim2, units=1, activation=None,
**kwargs):
        # Cria instâncias das camadas desejadas no modelo
```

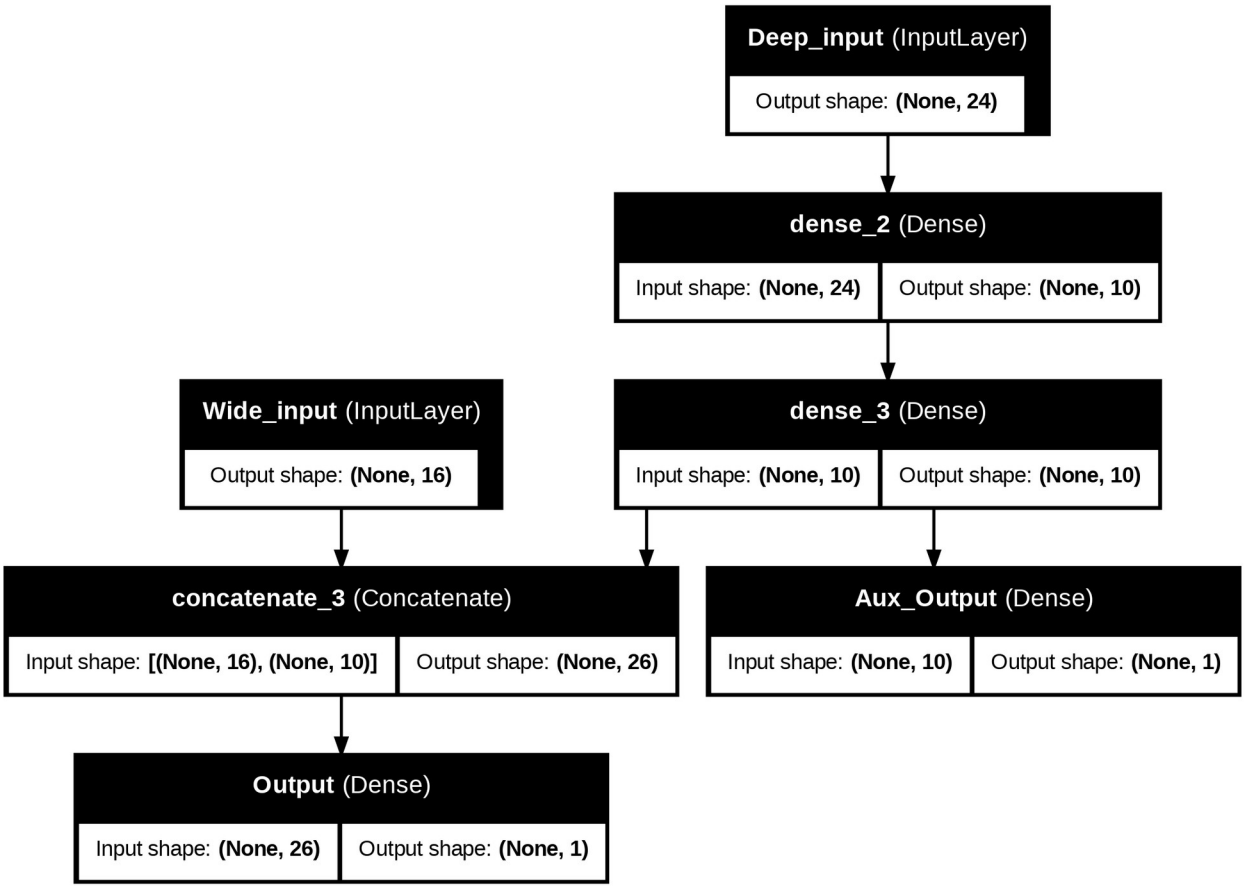
```
        self.input1 = Input(shape=dim1, name='Wide_input')
        self.input2 = Input(shape=dim2, name='Deep_input')
        self.dense1 = Dense(units, activation=activation)
        self.dense2 = Dense(units, activation=activation)
        self.out1 = Dense(1, name='Output')
        self.out2 = Dense(1, name='Aux_Output')

        # Obtém saída do modelo usando a função `call`
        self.out = self.call([self.input1, self.input2])

        # Inicializa modelo com entradas e saídas definidas
        super(TwoInputModel2, self).__init__(
              inputs=[self.input1, self.input2],
              # Chama self.call para obter as saídas
              outputs=self.call([self.input1, self.input2]),
              **kwargs)

    # Define cálculos a serem realizados na rede. Deve usar camadas
incializadas na função __init__()
    def call(self, inputs):
        # inputs = entrada do modelo, que no caso é uma lista com as
duas entradas
        input1, input2 = inputs

        # Camadas iniciais
        x1 = self.dense1(input2)
        x2 = self.dense2(x1)

        # Junta entrada 1 com saída da camaada dense3
        concat = concatenate([input1, x2])

        # Calcula saída 1
        y1 = self.out1(concat)

        # Define saída 2
        y2 = self.out2(x2)

        return [y1, y2]

# Cria modelo da classe TwoInputModel
model2 = TwoInputModel2(dim1=(16,), dim2=(24,), units=10,
activation='relu')

# Mostra sumario do modelo
model2.summary()
```

```
Model: "two_input_model2"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
```

```
 Connected to

 Deep_input (InputLayer)      │ (None, 24)              │            0
 -

 dense_2 (Dense)              │ (None, 10)              │          250
 Deep_input[0][0]

 Wide_input (InputLayer)      │ (None, 16)              │            0
 -

 dense_3 (Dense)              │ (None, 10)              │          110
 dense_2[1][0]

 concatenate_3                │ (None, 26)              │            0
 Wide_input[0][0],
 (Concatenate)                │                         │
 dense_3[1][0]

 Output (Dense)               │ (None, 1)               │           27
 concatenate_3[0][0]

 Aux_Output (Dense)           │ (None, 1)               │           11
 dense_3[1][0]

 Total params: 398 (1.55 KB)

 Trainable params: 398 (1.55 KB)

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

```python
# Mostra diagrama do modelo
from tensorflow.keras.utils import plot_model

plot_model(model2, show_shapes=True, show_layer_names=True,
expand_nested=True)
```



4. Rede residual simples

Vamos criar uma rede residual com dois tipos de blocos residuais para um problema de classificação de imagens.

Os dois blocos residuais são:

- Bloco residual Tipo 1 com camadas convolucionais;
- Bloco residual Tipo 2 com camadas densas.

Na Figura 2 é mostrado o modelo que será criado.

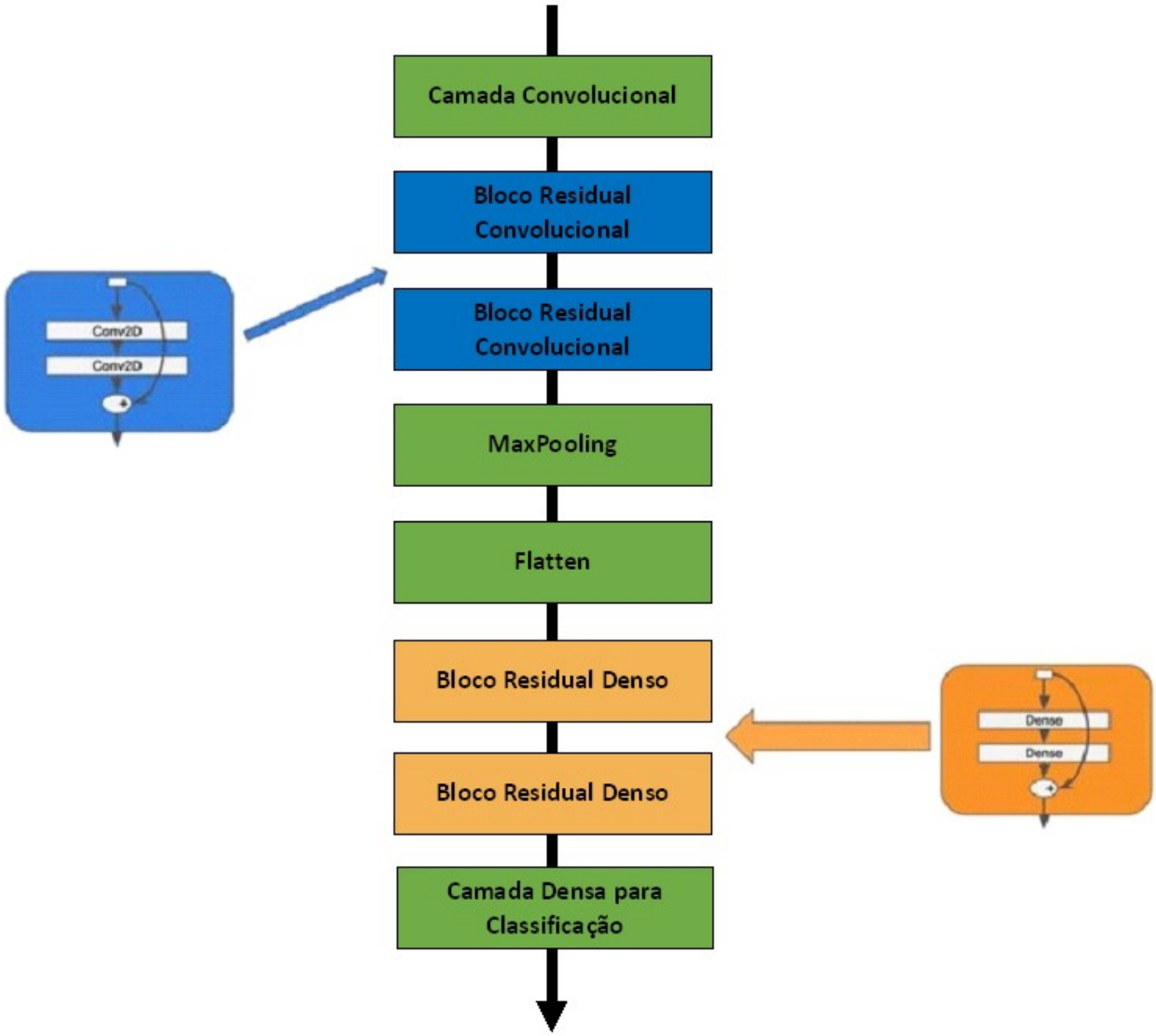


Figura 2 - Rede residual simples customizada.

4.1 Bloco residual convolucional

Vamos criar o bloco residual com camadas convolucionais com as seguintes características:

1. Para facilitar, todas as camadas convolucionais tem a mesma função de ativação (`relu`);
2. Número de camadas convolucionais é definido pelo parâmetro `n_layers` e pode ser em qualquer número.

```
# Importa classes de modelo e camadas
from tensorflow.keras.layers import Conv2D, Add

# Bloco residual com camadas convolucionais
class CVResBlock(tf.keras.Model):

    # Função de inicialização
    def __init__(self, n_layers, n_filters):
```

```
        # Incializa classe
        super(CVResBlock, self).__init__()

        # Cria instâncias das camadas desejadas no modelo
        self.add = Add()
        self.hidden = [Conv2D(n_filters, (3,3), padding='same',
activation='relu') for i in range(n_layers)]

    # Define cálculos a serem realizados na rede. Deve usar camadas
incializadas na função __init__()
    def call(self, inputs):
        # inputs = entrada do bloco
        x = inputs

        # Percorre camadas convolucionais existentes
        for layer in self.hidden:
            x = layer(x)

        return self.add([inputs, x])

cnblock = CVResBlock(2, 16)
print(cnblock)
```

```
<CVResBlock name=cv_res_block, built=False>
```

4.2 Bloco residual denso

Vamos criar o bloco residual com camadas densas com as seguintes características:

1. Para facilitar, todas as camadas densas tem a mesma função de ativação (`relu`);
2. Número de camadas densas é definido pelo parâmetro `n_layers` e pode ser em qualquer número.

```
# Importa classes de modelo e camada densa
from tensorflow.keras.layers import Dense, Add

# Bloco residual com camadas convolucionais
class DenseResBlock(tf.keras.Model):

    # Função de inicialização
    def __init__(self, n_layers, n_units):
        # Incializa classe
        super(DenseResBlock, self).__init__()

        # Cria instâncias das camadas desejadas no modelo
        self.add = Add()
        self.hidden = [Dense(n_units, activation='relu') for i in
range(n_layers)]

    # Define cálculos a serem realizados na rede. Deve usar camadas
incializadas na função __init__()
```

```
    def call(self, inputs):
        # inputs = entrada do bloco
        x = inputs

        # Percorre camadas densas existentes
        for layer in self.hidden:
            x = layer(x)

        return self.add([inputs, x])

dsblock = DenseResBlock(2, 16)
print(dsblock)
```

```
<DenseResBlock name=dense_res_block, built=False>
```

4.3 Rede residual simples

A rede residual da Figura 1 é criada usando os blocos residuais tipo 1 e 2.

```
# Importa camadas
from tensorflow.keras.layers import Flatten, MaxPool2D

# Rede residual da Figura 2
class SimpleResNet(tf.keras.Model):

    # Função de inicialização
    def __init__(self, n_class):
        # Incializa classe
        super(SimpleResNet, self).__init__()

        # Cria instâncias dos blocos e das camadas desejadas no modelo
        self.conv1 = Conv2D(64, (3,3), activation='relu')
        self.maxpool = MaxPool2D(2,2)
        self.flat = Flatten()
        self.blk1 = CVResBlock(2, 64)
        self.blk2 = CVResBlock(2, 64)
        self.dense = Dense(32, activation='relu')
        self.blk3 = DenseResBlock(2, 32)
        self.blk4 = DenseResBlock(2, 32)
        self.out = Dense(n_class, activation='softmax')

    # Define cálculos a serem realizados na rede. Deve usar camadas
incializadas na função __init__()
    def call(self, inputs):
        # Processa imagem com camada convolucional para ajustar
dimensões
        x = self.conv1(inputs)

        # Adicionas blocos convolucionais
        x = self.blk1(x)
```

```
        x = self.blk2(x)

        # Aplica camada MaxPooling
        x = self.maxpool(x)

        # Flatten tenso do bloco convolucional
        x = self.flat(x)

        # Inclui camada densa para ajustar dimensões
        x = self.dense(x)

        # Adiciona blocos densos
        x = self.blk3(x)
        x = self.blk4(x)

        # Calcula saída
        y = self.out(x)

        return y
```

- Observa-se que é necessário definir no `__init__()` todas as camadas (blocos) que serão usadas no modelo.

Vamos criar uma rede dessa classe para realizar uma tarefa de classificação das imagens do conjunto de dados CIFAR10.

```
# Instancia objeto da classe SimpleResNet
rna = SimpleResNet(10)

# Cria imagem fictícia para construir modelo
img = tf.zeros((1,32,32,3))

# Executa rna com imagem fictícia
y = rna(img)

print('Saida:', y)
rna.summary()
```

```
Saida: tf.Tensor([[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]],
shape=(1, 10), dtype=float32)

Model: "simple_res_net"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                          ┃ Output Shape                ┃ Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ conv2d_2 (Conv2D)                     │ (1, 30, 30, 64)             │ 1,792 │
```

```
├──────────────────────────────────────┼──────────────────────────────────┼
─────────────────┤
│ max_pooling2d (MaxPooling2D)         │ (1, 15, 15, 64)                  │
0 │
├──────────────────────────────────────┼──────────────────────────────────┼
─────────────────┤
│ flatten (Flatten)                    │ (1, 14400)                       │
0 │
├──────────────────────────────────────┼──────────────────────────────────┼
─────────────────┤
│ cv_res_block_1 (CVResBlock)          │ ?                                │
73,856 │
├──────────────────────────────────────┼──────────────────────────────────┼
─────────────────┤
│ cv_res_block_2 (CVResBlock)          │ ?                                │
73,856 │
├──────────────────────────────────────┼──────────────────────────────────┼
─────────────────┤
│ dense_6 (Dense)                      │ (1, 32)                          │
460,832 │
├──────────────────────────────────────┼──────────────────────────────────┼
─────────────────┤
│ dense_res_block_1 (DenseResBlock)    │ ?                                │
2,112 │
├──────────────────────────────────────┼──────────────────────────────────┼
─────────────────┤
│ dense_res_block_2 (DenseResBlock)    │ ?                                │
2,112 │
├──────────────────────────────────────┼──────────────────────────────────┼
─────────────────┤
│ dense_11 (Dense)                     │ (1, 10)                          │
330 │
└──────────────────────────────────────┴──────────────────────────────────┴
─────────────────┘

 Total params: 614,890 (2.35 MB)

 Trainable params: 614,890 (2.35 MB)

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

4.4 Conjunto de dados

Vamos carregar o conjunto de dados CIFAR10 diretamente do keras.

```
(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.cifar10.load_data()

print('Dimensão imagens de treinamento:', x_train.shape)
print('Dimensão imagens de teste:', x_test.shape)
```

```
print('Dimensão saídas de treinamento:', y_train.shape)
print('Dimensão saídas de teste:', y_test.shape)
```

```
Dimensão imagens de treinamento: (50000, 32, 32, 3)
Dimensão imagens de teste: (10000, 32, 32, 3)
Dimensão saídas de treinamento: (50000, 1)
Dimensão saídas de teste: (10000, 1)
```

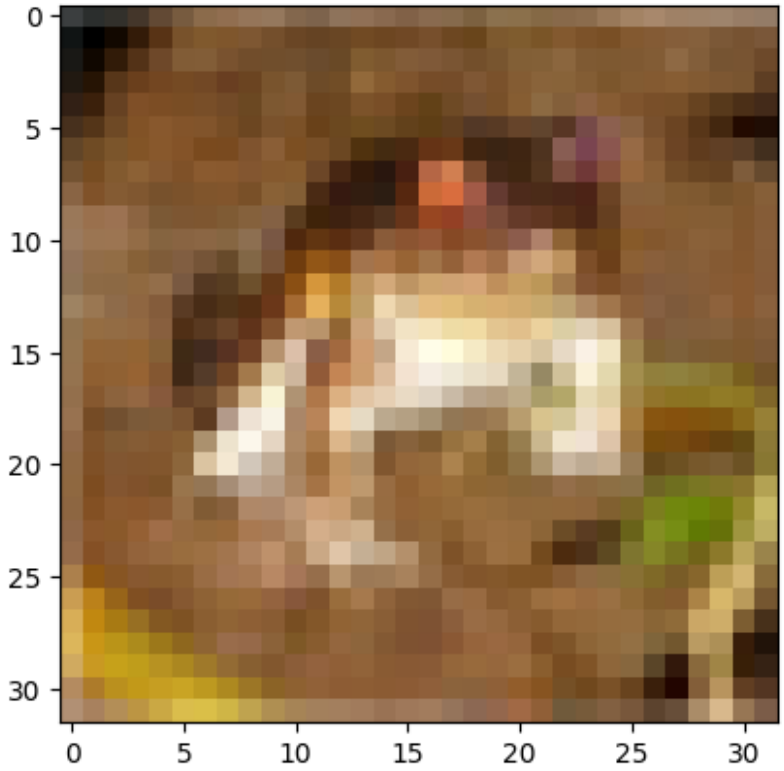
```
# Apresentação de alguns exemplos
plt.figure(figsize=(18, 6))
for i in range(8):
    plt.subplot(1, 8, i+1)
    plt.title(str(y_train[i]))
    plt.imshow(x_train[i])
    plt.axis('off')
plt.show()
```

Vamos criar uma estrutura tipo TFDS para pré-processar e carregar os dados no treinamento.

```
# Cria datasets de treinameto e teste
ds_train = tf.data.Dataset.from_tensor_slices((x_train, y_train))
ds_test = tf.data.Dataset.from_tensor_slices((x_test, y_test))

## Define função para normalizar as imagens codificar saída
def preproc(x, y):
    x_norm = tf.cast(x, dtype=tf.float32)/255.
    y_int = tf.cast(y, dtype=tf.int32)
    #y_hot = tf.one_hot(y_int, 10)
    return x_norm, y_int

# Define tamanho do lote
batch_size = 512

# Cria Dataset com a transformação que normaliza as imagens e codifica
saída
ds_train = ds_train.map(preproc, num_parallel_calls=tf.data.AUTOTUNE)
ds_test = ds_test.map(preproc, num_parallel_calls=tf.data.AUTOTUNE)

# Define tamanho de lotes e embaralha dados
ds_train =
ds_train.cache().batch(batch_size).prefetch(tf.data.AUTOTUNE)
ds_test = ds_test.cache().batch(batch_size).prefetch(tf.data.AUTOTUNE)

# Gera um lote de treinamento
for img, y in ds_train.take(1):
```

```
    print(img.shape, y.shape)

# Mostra primeiro exemplo do lote
print('Classe:', y[0])
plt.imshow(img[0])
plt.show()

# Apresenta dimensão do lote de imagens e alguns pixels da primeira
imagem
print('Dimensão do lote:', img.shape)
print('Valores de alguns pixels:', img[0,14,10:15].numpy())
```

```
(512, 32, 32, 3) (512, 1)
Classe: tf.Tensor([6], shape=(1,), dtype=int32)
```




```
Dimensão do lote: (512, 32, 32, 3)
Valores de alguns pixels: [[0.7529412  0.59607846 0.44313726]
 [0.7254902  0.5803922  0.41960785]
 [0.5686275  0.39607844 0.2       ]
 [0.79607844 0.63529414 0.4745098 ]
 [0.8745098  0.78431374 0.6666667 ]]
```

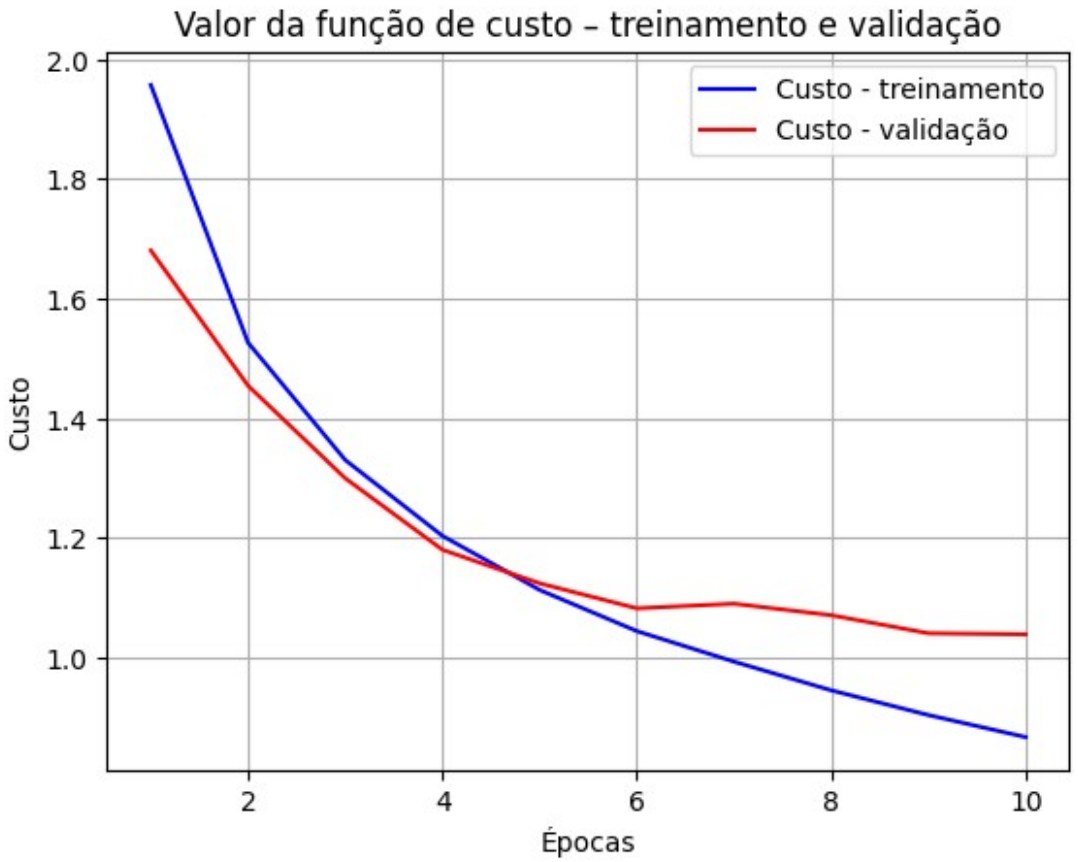
Compilação e treinamento

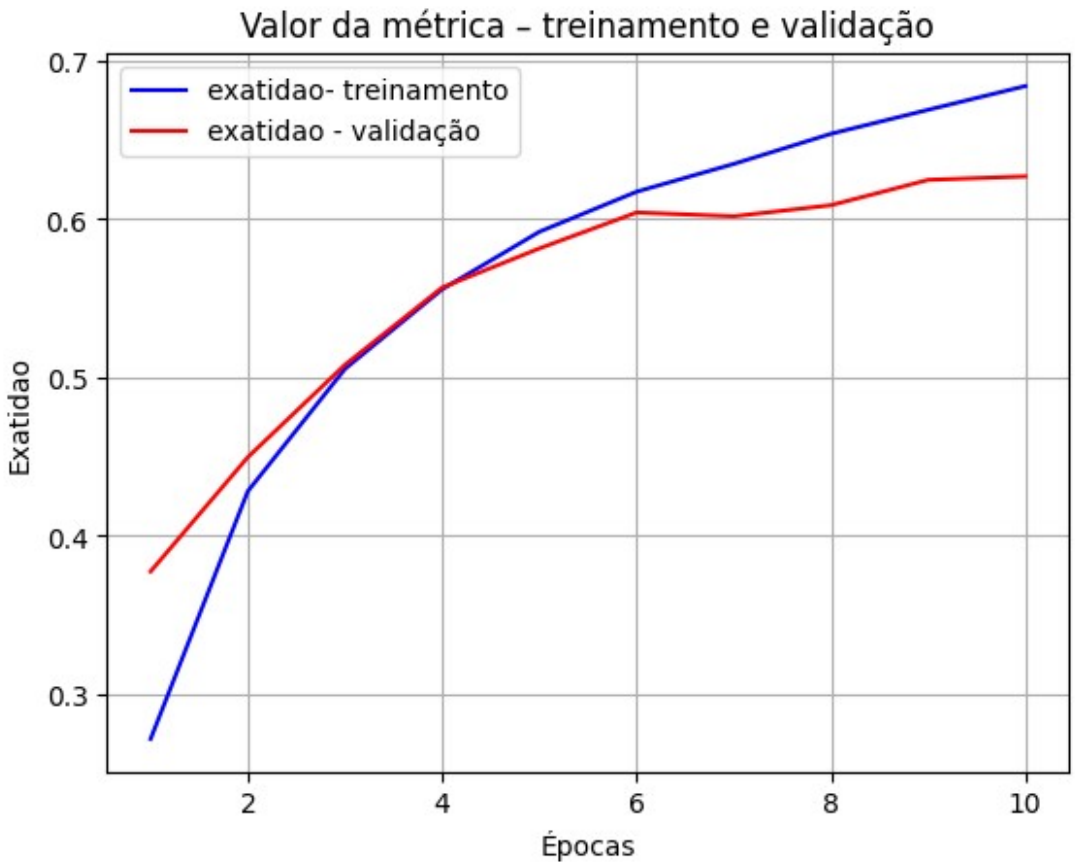
```
# Compila RNA
rna.compile(optimizer='adam', loss= 'sparse_categorical_crossentropy',
metrics=['accuracy'])

# Treinamento
result = rna.fit(ds_train, epochs=10, validation_data=ds_test)
```

```
Epoch 1/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 26s 183ms/step - accuracy: 0.1910 - loss:
2.1661 - val_accuracy: 0.3774 - val_loss: 1.6810
Epoch 2/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 12s 119ms/step - accuracy: 0.4063 - loss:
1.5795 - val_accuracy: 0.4496 - val_loss: 1.4550
Epoch 3/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 20s 116ms/step - accuracy: 0.4906 - loss:
1.3645 - val_accuracy: 0.5080 - val_loss: 1.3003
Epoch 4/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 11s 116ms/step - accuracy: 0.5473 - loss:
1.2240 - val_accuracy: 0.5568 - val_loss: 1.1808
Epoch 5/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 20s 115ms/step - accuracy: 0.5845 - loss:
1.1283 - val_accuracy: 0.5815 - val_loss: 1.1247
Epoch 6/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 11s 117ms/step - accuracy: 0.6092 - loss:
1.0595 - val_accuracy: 0.6042 - val_loss: 1.0829
Epoch 7/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 12s 118ms/step - accuracy: 0.6300 - loss:
1.0042 - val_accuracy: 0.6018 - val_loss: 1.0906
Epoch 8/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 11s 117ms/step - accuracy: 0.6480 - loss:
0.9589 - val_accuracy: 0.6088 - val_loss: 1.0714
Epoch 9/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 20s 115ms/step - accuracy: 0.6634 - loss:
0.9189 - val_accuracy: 0.6248 - val_loss: 1.0412
Epoch 10/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 11s 116ms/step - accuracy: 0.6792 - loss:
0.8781 - val_accuracy: 0.6270 - val_loss: 1.0395
```

```
# Salva treinamento na variável history para visualização
history_dict = result.history

# Salva custos, métricas e epocas em vetores
custo = history_dict['loss']
acc = history_dict['accuracy']
val_custo = history_dict['val_loss']
val_acc = history_dict['val_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.grid()
plt.show()

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






```
# Calcula função de custo e exatidão para todos os dados de teste
loss, accuracy = rna.evaluate(ds_test)

print("Função de custo:", loss)
print("Exatidão:", accuracy)
```

```
20/20 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.6249 - loss:
1.0336
Função de custo: 1.0394773483276367
Exatidão: 0.6269999742507935
```

5. Rede residual com ciclo

Na Figura 2 foi mostrado o modelo com 2 blocos residuais convolucionais e sequenciais em sequência, vamos modificar esse modelo e criar um modelo com um loop, conforme mostrado na Fifura 3.

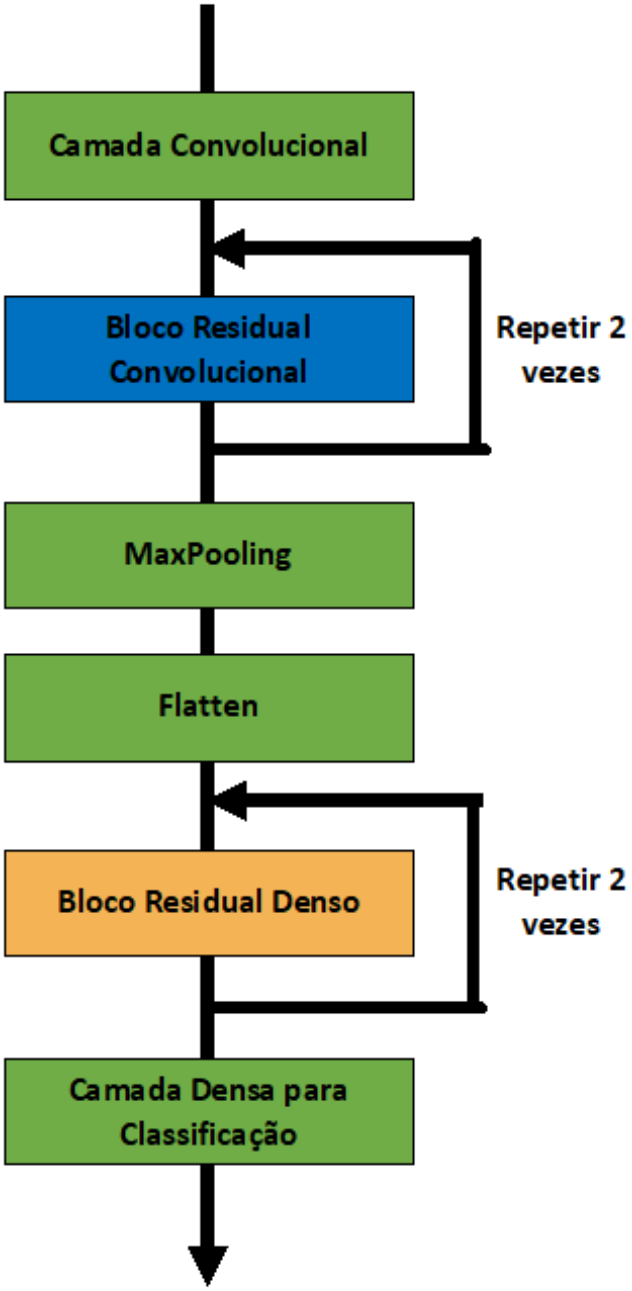


Figura 3 - Rede residual modificada.

Nesse modelo temos:

- No lugar de 2 blocos residuais convolucionais diferentes, temos somente um único bloco que é executado 2 vezes;
- Similarmente, no lugar de 2 blocos residuais densos diferentes, temos somente um único bloco que é executado 2 vezes.

Nesse caso os parâmetros das camadas dos blocos convolucionias (e densos) são os mesmos nas 3 vezes em que é executado, pois é o mesmo bloco.

```
# Rede residual da Figura 3
class CicleResNet(tf.keras.Model):

    # Função de inicialização
    def __init__(self, n_class):
        # Incializa classe
        super(CicleResNet, self).__init__()

        # Cria instâncias dos blocos e das camadas desejadas no modelo
        self.conv1 = Conv2D(64, (3,3), activation='relu')
        self.maxpool = MaxPool2D(2,2)
        self.flat = Flatten()
        self.blk1 = CVResBlock(2, 64)
        self.dense = Dense(32, activation='relu')
        self.blk2 = DenseResBlock(2, 32)
        self.out = Dense(n_class, activation='softmax')

    # Define cálculos a serem realizados na rede. Deve usar camadas
incializadas na função __init__()
    def call(self, inputs):
        # Processa imagem com camada convolucional para ajustar
dimensões
        x = self.conv1(inputs)

        # Executa bloco residual convolucional 3 vezes
        for _ in range(3):
            x = self.blk1(x)

        # Aplica camada MaxPooling
        x = self.maxpool(x)

        # Flatten tenso do bloco convolucional
        x = self.flat(x)

        # Inclui camada densa para ajustar dimensões
        x = self.dense(x)

        # Executa bloco denso 3 vezess
        for _ in range(3):
            x = self.blk2(x)

        # Calcula saída
        y = self.out(x)

        return y

# Instancia objeto da classe SimpleResNet
rnaC = CicleResNet(10)
```

```
# Cria imagem fictícia para construir modelo
img = tf.zeros((1,32,32,3))

# Executa rna com imagem fictícia
y = rnaC(img)

print('Saida:', y)
rnaC.summary()
```

```
Saida: tf.Tensor([[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]],
shape=(1, 10), dtype=float32)
```

```
Model: "cicle_res_net"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃          Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━┩
│ conv2d_7 (Conv2D)                    │ (1, 30, 30, 64)             │            1,792 │
├──────────────────────────────────────┼─────────────────────────────┼──────────────────┤
│ max_pooling2d_1 (MaxPooling2D)       │ (1, 15, 15, 64)             │                0 │
├──────────────────────────────────────┼─────────────────────────────┼──────────────────┤
│ flatten_1 (Flatten)                  │ (1, 14400)                  │                0 │
├──────────────────────────────────────┼─────────────────────────────┼──────────────────┤
│ cv_res_block_3 (CVResBlock)          │ ?                           │           73,856 │
├──────────────────────────────────────┼─────────────────────────────┼──────────────────┤
│ dense_12 (Dense)                     │ (1, 32)                     │          460,832 │
├──────────────────────────────────────┼─────────────────────────────┼──────────────────┤
│ dense_res_block_3 (DenseResBlock)    │ ?                           │            2,112 │
├──────────────────────────────────────┼─────────────────────────────┼──────────────────┤
│ dense_15 (Dense)                     │ (1, 10)                     │              330 │
└──────────────────────────────────────┴─────────────────────────────┴──────────────────┘
```

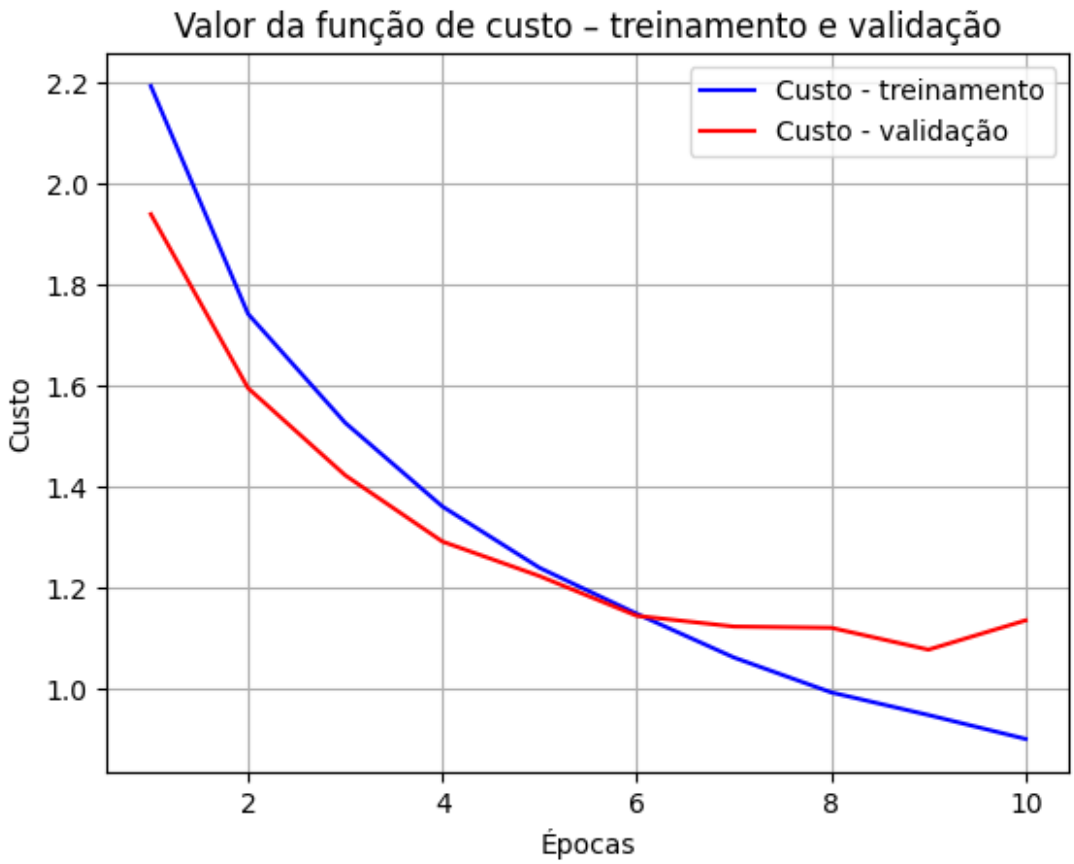
```
 Total params: 538,922 (2.06 MB)

 Trainable params: 538,922 (2.06 MB)

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

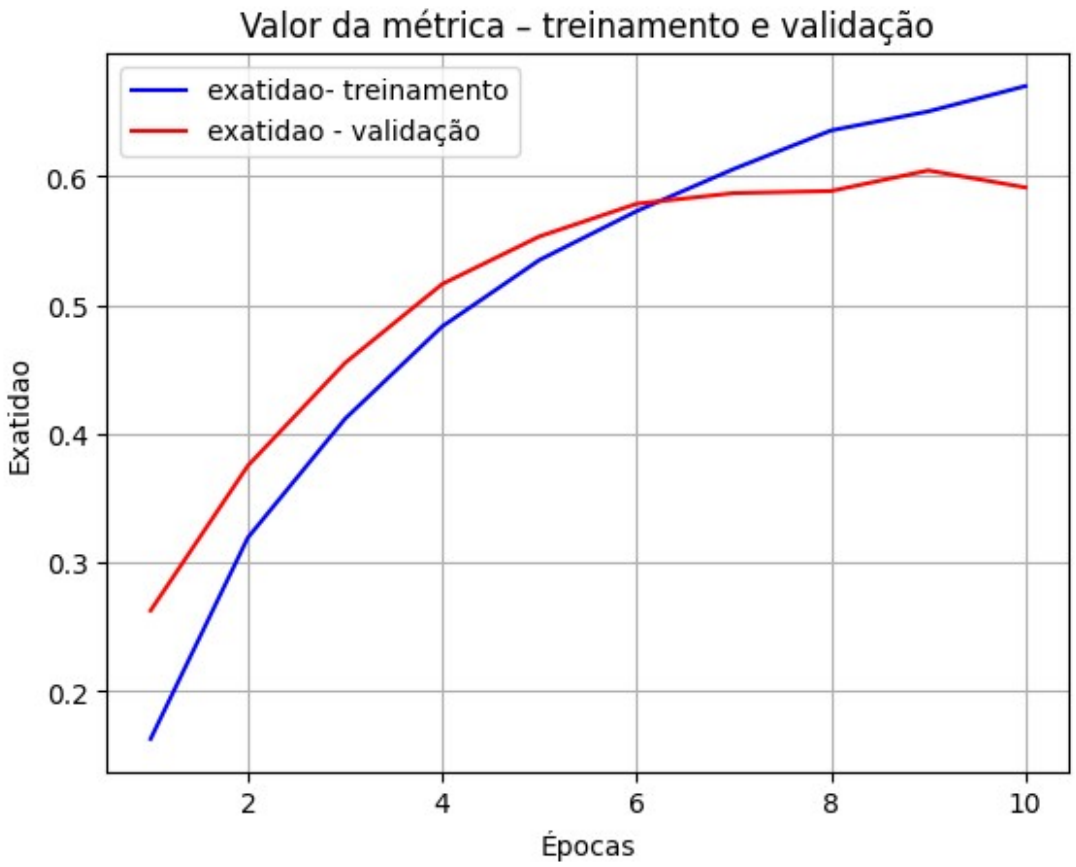
```
# Compila RNA
rnaC.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

# Treinamento
result = rnaC.fit(ds_train, epochs=10, validation_data=ds_test)
```

```
Epoch 1/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 25s 193ms/step - accuracy: 0.1315 - loss:
2.3415 - val_accuracy: 0.2626 - val_loss: 1.9403
Epoch 2/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 16s 166ms/step - accuracy: 0.2913 - loss:
1.8252 - val_accuracy: 0.3751 - val_loss: 1.5958
Epoch 3/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 20s 165ms/step - accuracy: 0.3900 - loss:
1.5677 - val_accuracy: 0.4550 - val_loss: 1.4232
Epoch 4/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 16s 166ms/step - accuracy: 0.4683 - loss:
1.3895 - val_accuracy: 0.5163 - val_loss: 1.2920
Epoch 5/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 16s 167ms/step - accuracy: 0.5251 - loss:
1.2630 - val_accuracy: 0.5533 - val_loss: 1.2236
Epoch 6/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 16s 167ms/step - accuracy: 0.5633 - loss:
1.1670 - val_accuracy: 0.5787 - val_loss: 1.1449
Epoch 7/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 16s 167ms/step - accuracy: 0.5976 - loss:
1.0761 - val_accuracy: 0.5870 - val_loss: 1.1237
Epoch 8/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 17s 172ms/step - accuracy: 0.6290 - loss:
1.0083 - val_accuracy: 0.5886 - val_loss: 1.1213
Epoch 9/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 16s 168ms/step - accuracy: 0.6433 - loss:
0.9613 - val_accuracy: 0.6046 - val_loss: 1.0780
Epoch 10/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 16s 167ms/step - accuracy: 0.6675 - loss:
0.9060 - val_accuracy: 0.5914 - val_loss: 1.1360
```

```
# Salva treinamento na variável history para visualização
history_dict = result.history

# Salva custos, métricas e epocas em vetores
custo = history_dict['loss']
acc = history_dict['accuracy']
val_custo = history_dict['val_loss']
```

```
val_acc = history_dict['val_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.grid()
plt.show()

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


Valor da função de custo – treinamento e validação
Custo - treinamento
Custo - validação
Custo
Épocas
2.2
2.0
1.8
1.6
1.4
1.2
1.0
2
4
6
8
10




```
# Calcula função de custo e exatidão para todos os dados de teste
loss, accuracy = rnaC.evaluate(ds_test)

print("Função de custo:", loss)
print("Exatidão:", accuracy)
```

```
20/20 ━━━━━━━━━━━━━━━━━━━━ 1s 45ms/step - accuracy: 0.5911 - loss:
1.1304
Função de custo: 1.1360113620758057
Exatidão: 0.5914000272750854
```