Trabalho #2 - Rede residual e ajuste de modelo

Nesse trabalho você vai criar uma rede residual para realizar uma tarefa de classificação de múltiplas classes.

A tarefa consiste em classificar objetos do conjunto de dados CIFAR100

Esse trabalho é dividido nas seguintes etapas:

1. Explorar as imagens do conjunto de dados;
2. Construir e treinar uma rede residual para identificar o objeto mostrado na imagem;
3. Avaliar o desempenho da rede;
4. Ajustar o modelo para melhorar o desempenho.

Importação das principais bibliotecas

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

```
'2.17.1'
```

1. Conjunto de dados

O conjunto de dados CIFAR10 é composto por imagens coloridas de dimensão (32, 32, 3).

Esse conjunto de imagens é dedicado à classificçaõ multiclasse com 10 tipos de objetos.

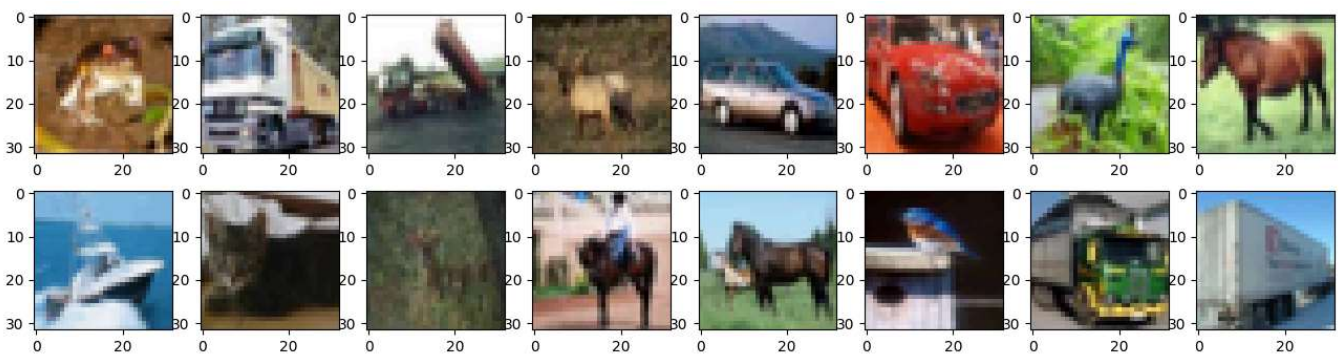
Execute a célula abaixo para carregar o conjunto de dados CIFAR-10, que já está disponível diretamente no Keras.

```
# Importar conjunto de dados
from tensorflow.keras.datasets import cifar10

# Carregar o conjunto de dados CIFAR-10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

Execute a célula abaixo para visualizar algumas imagens.

```
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])
        index += 1
plt.show()
```



Clique duas vezes (ou pressione "Enter") para editar

Exercício #1: Pré-processamento dos dados

Na célula abaixo crie um código para realizar o seguinte:

1. Normalizar as imagens de forma que os seus pixels sejam valores reais entre 0 e 1;
2. Codificação one-hot das saídas.

```
# PARA VOCÊ FAZER: Normalização das imagens e codificação one-hot das saídas
# Importa função to_categorical
from tensorflow.keras.utils import to_categorical

# Normalização das imagens
### COMECE AQUI #### (~2 linhas)
x_train_norm = x_train.astype('float32') / 255.0
x_test_norm = x_test.astype('float32') / 255.0
### TERMINE AQUI ###

# Converter as classe para one-hot encoding
### COMECE AQUI #### (~2 linhas)
y_train_hot = to_categorical(y_train, 10)
y_test_hot = to_categorical(y_test, 10)
### TERMINE AQUI ###

print(f"Dimensão do conjunto de treinamento: {x_train_norm.shape}")
print(f"Dimensão do conjunto de teste: {x_test_norm.shape}")
print(f"Dimensão das saídas de treinamento: {y_train_hot.shape}")
print(f"Dimensão das saídas de teste: {y_test_hot.shape}")
print('Valores máximos e mínimos das imagens de treinamento:', np.max(x_train_norm), ',', np.min(x_train_norm))
print('Valores máximos e mínimos das imagens de teste:', np.max(x_test_norm), ',',  np.min(x_test_norm))
```

```
Dimensão do conjunto de treinamento: (50000, 32, 32, 3)
Dimensão do conjunto de teste: (10000, 32, 32, 3)
Dimensão das saídas de treinamento: (50000, 10)
Dimensão das saídas de teste: (10000, 10)
Valores máximos e mínimos das imagens de treinamento: 1.0 , 0.0
Valores máximos e mínimos das imagens de teste: 1.0 , 0.0
```

Saída esperada:

```
Dimensão do conjunto de treinamento: (50000, 32, 32, 3)
Dimensão do conjunto de teste: (10000, 32, 32, 3)
Dimensão das saídas de treinamento: (50000, 10)
Dimensão das saídas de teste: (10000, 10)
Valores máximos e mínimos das imagens: 1.0 0.0
```

2. Configuração do modelo

Nesse trabalho você vai criar uma rede residual com camadas densas.

Uma rede residual é composta por blocos residuais. Na Figura 1, é mostrado um bloco residual.

ASa equações que impelementam esse bloco são as seguintes:

$$a^{[l+1]} = dense(a^{[l]})$$
$$z^{[l+2]} = dense(a^{[l]})$$
$$a^{[l+2]} = g^{[l+2]}(z^{[l+2]} + a^{[l]}))$$

onde $dense$ é uma camada densa. Observe que a função de ativação da segunda camada densa do bloco somente é aplicada após a soma das ativações $a^{[l]}$ e dos estados $z^{[l+2]}$.

A camada `layers.Add()` do Keras realiza a soma de dois tensores.

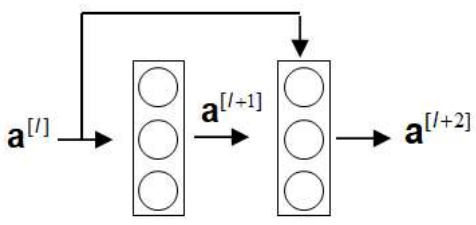


Figura 1 - Esquema de um bloco residual.

Exercício #2: Codificação do bloco residual

Na célula abaixo crie uma função que implementa o bloco residual mostrado na Figura 1. Use a função de ativação relu.

```
### PARA VOCÊ FAZER: Definir o bloco residual
# Importa classe de camadas e modelos
from tensorflow.keras import layers, models

# Importa função de ativação
from tensorflow.keras.activations import relu

def residual_block(x, units):
    # Primeira camada densa
    ### COMECE AQUI ### (1 linha)
    z1 = layers.Dense(units, activation=relu)(x)
    ### TERMINE AQUI ###

    # Segunda camada densa sem ativação
    ### COMECE AQUI ### (1 linha)
    z2 = layers.Dense(units, activation=None)(z1)
    ### TERMINE AQUI ###

    # Soma entrada com z2 com saída da segunda camada
    ### COMECE AQUI ### (1 linha)
    sum = layers.Add()([x, z2])
    ### TERMINE AQUI ###

    # Aplica função de ativação
    ### COMECE AQUI ### (1 linha)
    a2 = layers.Activation('relu')(sum)
    ### TERMINE AQUI ###

    return a2
```

Execute a célula abaixo para testar o seu bloco residual.

```
np.random.seed(3)
x = np.random.randn(3,5)
a2 = residual_block(x, 5)
print('x:', x, '\n')
print('a2:', a2)
```

```
x: [[ 1.78862847  0.43650985  0.09649747 -1.8634927  -0.2773882 ]
 [-0.35475898 -0.08274148 -0.62700068 -0.04381817 -0.47721803]
 [-1.31386475  0.88462238  0.88131804  1.70957306  0.05003364]]

a2: tf.Tensor(
[[2.1368363  0.         0.         0.         0.5600672 ]
 [0.         0.         0.         0.10481881 0.        ]
 [0.         0.8491116  0.7752985  1.7744516  0.1632227 ]], shape=(3, 5), dtype=float32)
```

Saída esperada:

```
x: [[ 1.78862847  0.43650985  0.09649747 -1.8634927  -0.2773882 ]
[-0.35475898 -0.08274148 -0.62700068 -0.04381817 -0.47721803]
[-1.31386475  0.88462238  0.88131804  1.70957306  0.05003364]]

a2: tf.Tensor(
[[1.6887243  0.5611134  0.5987834  0.         0.67116445]
[0.         0.03314844 0.         0.1769045  0.        ]
[0.3389045  0.16169035 0.         0.7670167  0.        ]], shape=(3, 5), dtype=float32)
```

Exercício #3: Configuração da rede

A rede que será utilizada será composta por 2 blocos residuais, sendo que entre eles deve ter um camada densa para ajustar a dimensão dos dados.

Na célula abaixo crie uma função para gerar a rede residual. Essa função deve receber o seguinte:

1. Dimensão das imagens de entrada;
2. lista com número de unidades em cada bloco e cada camada da rede;
3. Número de classes para definir o número de unidades da camada de saída;
4. Exceto na camada de saída utiliza função de ativação relu.

```
### PARA VOCÊ FAZER: Definir a arquitetura do modelo
```

```
def create_residual_model(n1, n2, num_classes, input_shape=(32, 32, 3)):
    ### COMECE AQUI ###
    # Camada de entrada
    inputs = layers.Input(shape=input_shape)

    # Camada para esticar a imagem (flattening)
    x = layers.Flatten()(inputs)

    # Primeira camada densa com dropout
    x = layers.Dense(n1, activation='relu')(x)

    # Primeiro bloco residual
    x = residual_block(x, n1)

    # Camada densa intermediária para ajustar a dimensão
    x = layers.Dense(n2, activation='relu')(x)

    # Segundo bloco residual
    x = residual_block(x, n2)

    # Camada de saída (número de classes)
    outputs = layers.Dense(num_classes, activation='softmax')(x)

    # Construir o modelo
    model = models.Model(inputs=inputs, outputs=outputs)
    ### TERMINE AQUI ###

    return model
```

Para criar o modelo, utilize a função `create_residual_model()` com os seguintes parâmetros:

- n1 = 256;
- n2 = 128;
- num_classes = 10.

```
# PARA VOCE FAZER: Criar modelo

# Definir parâmetros
### PARA VOCE FAZER ### (3 linhas)
n1 = 256
n2 = 128
num_classes = 10
### TERMINE AQUI ###

# Criar modelo
### PARA VOCE FAZER ### (1 linha)
rna = create_residual_model(n1, n2, num_classes)
### TERMINE AQUI ###

rna.summary()
```

Model: "functional_5"

Layer (type)	Output Shape	Param #	Connected to
input_layer_5 (InputLayer)	(None, 32, 32, 3)	0	-
flatten_5 (Flatten)	(None, 3072)	0	input_layer_5[0][0]
dense_44 (Dense)	(None, 256)	786,688	flatten_5[0][0]
dense_45 (Dense)	(None, 256)	65,792	dense_44[0][0]
dense_46 (Dense)	(None, 256)	65,792	dense_45[0][0]
add_12 (Add)	(None, 256)	0	dense_44[0][0], dense_46[0][0]
activation_12 (Activation)	(None, 256)	0	add_12[0][0]
dense_47 (Dense)	(None, 128)	32,896	activation_12[0][0]
dense_48 (Dense)	(None, 128)	16,512	dense_47[0][0]
dense_49 (Dense)	(None, 128)	16,512	dense_48[0][0]
add_13 (Add)	(None, 128)	0	dense_47[0][0], dense_49[0][0]
activation_13 (Activation)	(None, 128)	0	add_13[0][0]
dense_50 (Dense)	(None, 10)	1,290	activation_13[0][0]

Total params: 985,482 (3.76 MB)
Trainable params: 985,482 (3.76 MB)
Non-trainable params: 0 (0.00 B)

Saída esperada:

Model: "functional"

Layer (type)	Output Shape	Param #	Connected to
input_layer_1 (InputLayer)	(None, 32, 32, 3)	0	-
flatten_1 (Flatten)	(None, 3072)	0	input_layer_1[0][0]
dense_12 (Dense)	(None, 256)	786,688	flatten_1[0][0]
dense_13 (Dense)	(None, 256)	65,792	dense_12[0][0]
dense_14 (Dense)	(None, 256)	65,792	dense_13[0][0]
add_6 (Add)	(None, 256)	0	dense_12[0][0], dense_14[0][0]
activation_4 (Activation)	(None, 256)	0	add_6[0][0]
dense_15 (Dense)	(None, 128)	32,896	activation_4[0][0]
dense_16 (Dense)	(None, 128)	16,512	dense_15[0][0]
dense_17 (Dense)	(None, 128)	16,512	dense_16[0][0]
add_7 (Add)	(None, 128)	0	dense_15[0][0], dense_17[0][0]
activation_5 (Activation)	(None, 128)	0	add_7[0][0]
dense_18 (Dense)	(None, 10)	1,290	activation_5[0][0]

Total params: 985,482 (3.76 MB)
Trainable params: 985,482 (3.76 MB)
Non-trainable params: 0 (0.00 B)

3. Compilação e treinamento do modelo

Exercício #4: Compilação do modelo

Agora que o modelo foi criado, você precisa compilá-lo. Vamos usar a função de perda `categorical_crossentropy` para classificação multiclasse e o otimizador `Adam`.

```
# PARA VOCE FAZER: Compilar o modelo
from tensorflow.keras.optimizers import Adam
### COMECE AQUI ### (1 comando)
rna.compile(optimizer=Adam(learning_rate=0.0001), loss='categorical_crossentropy', metrics=['accuracy'])
### TERMINE AQUI ###
```

Exercício #5: Treinar o modelo

Para treinar o modelo use 100 épocas e um lote de 256 exemplos.

```
# PARA VOCE FAZER: Treinar o modelo

### COMECE AQUI ### (1 comando)
history = rna.fit(x_train_norm, y_train_hot, epochs=100, batch_size=256, validation_data=(x_test_norm, y_test_hot))
### TERMINE AQUI ###
```

```
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8785 - loss: 0.3577 - val_accuracy: 0.5164 - val_loss: 2.2041
```

Saída esperada:

```
Epoch 1/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 6s 15ms/step - accuracy: 0.2288 - loss: 2.1627 - val_accuracy: 0.3783 - val_loss: 1.7425
Epoch 2/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.3720 - loss: 1.7419 - val_accuracy: 0.4168 - val_loss: 1.6358
.
.
.
Epoch 99/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9347 - loss: 0.1806 - val_accuracy: 0.4800 - val_loss: 4.0803
Epoch 100/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9350 - loss: 0.1817 - val_accuracy: 0.4942 - val_loss: 4.1091
```

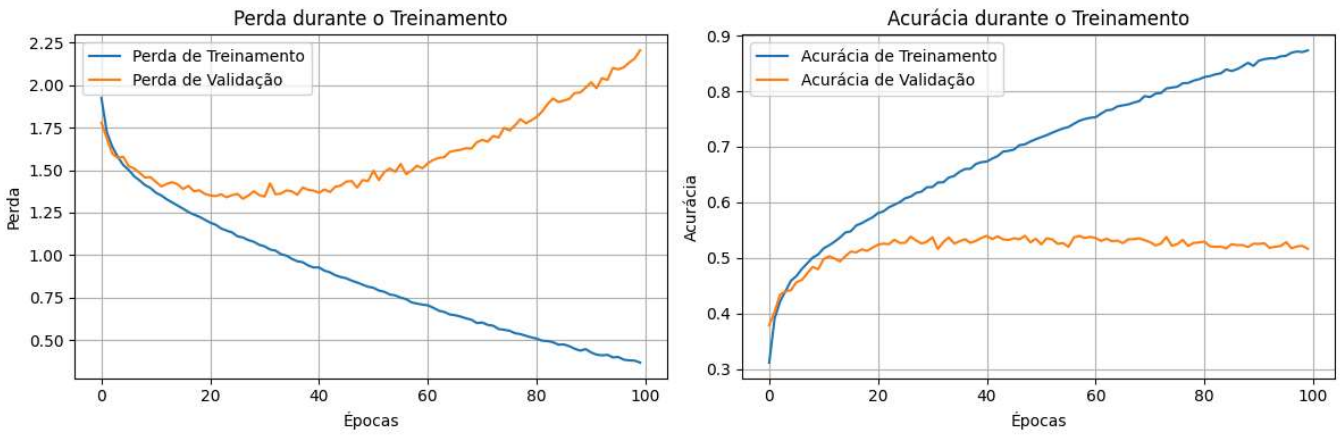
Execute a célula abaixo para vizualizar os gráficos do processo de treinamento.

```
# Plotar a perda e acurácia durante o treinamento
plt.figure(figsize=(12, 4))

# Plotando a perda
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Perda de Treinamento')
plt.plot(history.history['val_loss'], label='Perda de Validação')
plt.title('Perda durante o Treinamento')
plt.xlabel('Épocas')
plt.ylabel('Perda')
plt.grid()
plt.legend()

# Plotando a acurácia
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Acurácia de Treinamento')
plt.plot(history.history['val_accuracy'], label='Acurácia de Validação')
plt.title('Acurácia durante o Treinamento')
plt.xlabel('Épocas')
plt.ylabel('Acurácia')
plt.grid()
plt.legend()

plt.tight_layout()
plt.show()
```



Exercício #6: Avaliar o modelo

Use o método evaluate e calcule a função de custo e a métrica para os dados de treinamento e teste.

```
### PARA VOCE FAZER: Avaliar o modelo

### COMECE AQUI ### (2 linhas)
train_loss, train_accuracy = rna.evaluate(x_train_norm, y_train_hot, verbose=1)
```

```
test_loss, test_accuracy = rna.evaluate(x_test_norm, y_test_hot, verbose=1)
### TERMINE AQUI ###
```

```
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.8709 - loss: 0.3737
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.5160 - loss: 2.1919
```

Saída esperada:

```
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9438 - loss: 0.1639
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.4957 - loss: 4.0715
```

4. Ajuste do modelo

Você agora tem um modelo básico de rede residual para classificação de imagens no conjunto CIFAR100. Por meio da adição de blocos residuais, o modelo pode ser mais profundo sem enfrentar problemas de degradação do desempenho.

Com base nesse modelo, você pode fazer experimentos adicionais para melhorar a performance, como ajustar a arquitetura, adicionar camadas de normalização, ou utilizar técnicas de data augmentation.

Exercício #7: Ajustar o modelo para obter resultados melhores

Nesse exercício você deve fazer ajustes no modelo para melhorar o seu desempenho. As possíveis modificações são: adicionar mais blocos residuais, mudar o número de unidades nos blocos e nas camadas densas, ou tentar diferentes técnicas de regularização (dropout, L2 regularization etc.) Além disso, é possível testar diferentes algoritmos de otimização e taxas de aprendizado.

Implemente as seguintes modificações no modelo:

1. Aumentar número de blocos residuais e aumentar número de unidades nas camadas;
2. Incluir camadas de dropout no modelo maior do item (1);
3. Retirar as camadas de dropout e aplicar regularização L2 no modelo maior do item(1).

Para cada modificação você deve apresentar o novo modelo, a compilação, o treinamento e a avaliação.

A sua avaliação nesse exercício depende dos resultados de exatidão nos dados de teste.

Analise os resultados do ajuste do modelo.

1. Aumentar número de blocos residuais e aumentar número de unidades nas camadas;

```
def modelo_residual_v1(n1, n2, n3, n4, num_classes, input_shape=(32, 32, 3)):
    inputs = layers.Input(shape=input_shape)
    x = layers.Flatten()(inputs)

    x = layers.Dense(n1, activation='relu')(x)
    x = residual_block(x, n1)

    x = layers.Dense(n2, activation='relu')(x)
    x = residual_block(x, n2)

    x = layers.Dense(n3, activation='relu')(x)
    x = residual_block(x, n3)

    x = layers.Dense(n4, activation='relu')(x)
    x = residual_block(x, n4)

    # Camada de saída
    outputs = layers.Dense(num_classes, activation='softmax')(x)

    model = models.Model(inputs=inputs, outputs=outputs)
    return model

rna_v1 = modelo_residual_v1(512, 512, 256, 128, 10)
rna_v1.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history_v1 = rna_v1.fit(x_train_norm, y_train_hot, epochs=100, batch_size=256, validation_data=(x_test_norm, y_test_hot))

train_loss_v1, train_accuracy_v1 = rna_v1.evaluate(x_train_norm, y_train_hot, verbose=1)
test_loss_v1, test_accuracy_v1 = rna_v1.evaluate(x_test_norm, y_test_hot, verbose=1)
```

```
Epoch 75/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9603 - loss: 0.1208 - val_accuracy: 0.4939 - val_loss: 3.7874
Epoch 76/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9686 - loss: 0.0949 - val_accuracy: 0.4780 - val_loss: 3.7664
Epoch 77/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9522 - loss: 0.1410 - val_accuracy: 0.4884 - val_loss: 3.5977
Epoch 78/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9695 - loss: 0.0901 - val_accuracy: 0.4798 - val_loss: 3.5908
Epoch 79/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9562 - loss: 0.1366 - val_accuracy: 0.4889 - val_loss: 3.5947
Epoch 80/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9667 - loss: 0.1003 - val_accuracy: 0.4944 - val_loss: 3.7310
Epoch 81/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.9705 - loss: 0.0852 - val_accuracy: 0.4938 - val_loss: 3.8268
Epoch 82/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9641 - loss: 0.1082 - val_accuracy: 0.4883 - val_loss: 3.5421
Epoch 83/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9575 - loss: 0.1272 - val_accuracy: 0.4913 - val_loss: 3.5582
Epoch 84/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9647 - loss: 0.1060 - val_accuracy: 0.4957 - val_loss: 3.6203
Epoch 85/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9666 - loss: 0.0998 - val_accuracy: 0.4905 - val_loss: 3.5880
Epoch 86/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9648 - loss: 0.1045 - val_accuracy: 0.5010 - val_loss: 3.7322
Epoch 87/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9767 - loss: 0.0697 - val_accuracy: 0.4922 - val_loss: 3.7889
Epoch 88/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9632 - loss: 0.1124 - val_accuracy: 0.4966 - val_loss: 3.6173
Epoch 89/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9595 - loss: 0.1266 - val_accuracy: 0.4906 - val_loss: 3.7174
Epoch 90/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9606 - loss: 0.1175 - val_accuracy: 0.4880 - val_loss: 3.4155
Epoch 91/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9684 - loss: 0.0926 - val_accuracy: 0.4975 - val_loss: 3.7589
Epoch 92/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.9740 - loss: 0.0805 - val_accuracy: 0.4815 - val_loss: 3.6398
Epoch 93/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9599 - loss: 0.1203 - val_accuracy: 0.4929 - val_loss: 3.6864
Epoch 94/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9723 - loss: 0.0856 - val_accuracy: 0.4920 - val_loss: 3.5844
Epoch 95/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9731 - loss: 0.0813 - val_accuracy: 0.4814 - val_loss: 3.4407
Epoch 96/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9610 - loss: 0.1222 - val_accuracy: 0.4921 - val_loss: 3.4769
Epoch 97/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.9700 - loss: 0.0919 - val_accuracy: 0.4938 - val_loss: 3.6466
Epoch 98/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.9739 - loss: 0.0782 - val_accuracy: 0.4883 - val_loss: 3.5508
Epoch 99/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9684 - loss: 0.1000 - val_accuracy: 0.4957 - val_loss: 3.8222
Epoch 100/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9743 - loss: 0.0760 - val_accuracy: 0.4748 - val_loss: 3.6450
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9460 - loss: 0.1634
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.4783 - loss: 3.6218
```

2. Incluir camadas de dropout no modelo maior

```
def modelo_residual_v2_dropout(n1, n2, n3, n4, num_classes, input_shape=(32, 32, 3)):
    inputs = layers.Input(shape=input_shape)
    x = layers.Flatten()(inputs)

    x = layers.Dense(n1, activation='relu')(x)
    x = layers.Dropout(0.2)(x)
    x = residual_block(x, n1)

    x = layers.Dense(n2, activation='relu')(x)
    x = layers.Dropout(0.2)(x)
    x = residual_block(x, n2)

    x = layers.Dense(n3, activation='relu')(x)
    x = layers.Dropout(0.2)(x)
    x = residual_block(x, n3)

    x = layers.Dense(n4, activation='relu')(x)
    x = layers.Dropout(0.2)(x)
    x = residual_block(x, n4)

    # Camada de saída
    outputs = layers.Dense(num_classes, activation='softmax')(x)

    model = models.Model(inputs=inputs, outputs=outputs)
    return model

rna_v2 = modelo_residual_v2_dropout(512, 512, 256, 128, 10)
rna_v2.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
history_v2 = rna_v2.fit(x_train_norm, y_train_hot, epochs=100, batch_size=256, validation_data=(x_test_norm, y_test_hot))

train_loss_v2, train_accuracy_v2 = rna_v2.evaluate(x_train_norm, y_train_hot, verbose=1)
test_loss_v2, test_accuracy_v2 = rna_v2.evaluate(x_test_norm, y_test_hot, verbose=1)
```

```
Epoch 73/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5127 - loss: 1.3607 - val_accuracy: 0.4757 - val_loss: 1.5084
Epoch 74/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.5156 - loss: 1.3541 - val_accuracy: 0.4820 - val_loss: 1.4914
Epoch 75/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.5192 - loss: 1.3450 - val_accuracy: 0.4816 - val_loss: 1.4828
Epoch 76/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.5173 - loss: 1.3465 - val_accuracy: 0.4760 - val_loss: 1.5045
Epoch 77/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.5211 - loss: 1.3425 - val_accuracy: 0.4755 - val_loss: 1.4931
Epoch 78/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.5192 - loss: 1.3364 - val_accuracy: 0.4675 - val_loss: 1.5295
Epoch 79/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.5233 - loss: 1.3344 - val_accuracy: 0.4842 - val_loss: 1.4783
Epoch 80/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.5210 - loss: 1.3377 - val_accuracy: 0.4975 - val_loss: 1.4397
Epoch 81/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5250 - loss: 1.3288 - val_accuracy: 0.4720 - val_loss: 1.4953
Epoch 82/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5211 - loss: 1.3335 - val_accuracy: 0.4939 - val_loss: 1.4567
Epoch 83/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5259 - loss: 1.3257 - val_accuracy: 0.4760 - val_loss: 1.4882
Epoch 84/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5279 - loss: 1.3292 - val_accuracy: 0.4878 - val_loss: 1.4765
Epoch 85/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5227 - loss: 1.3234 - val_accuracy: 0.4775 - val_loss: 1.5066
Epoch 86/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5295 - loss: 1.3289 - val_accuracy: 0.4872 - val_loss: 1.4700
Epoch 87/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5297 - loss: 1.3131 - val_accuracy: 0.4814 - val_loss: 1.4765
Epoch 88/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.5232 - loss: 1.3301 - val_accuracy: 0.4891 - val_loss: 1.4756
Epoch 89/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.5274 - loss: 1.3224 - val_accuracy: 0.4941 - val_loss: 1.4608
Epoch 90/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.5288 - loss: 1.3090 - val_accuracy: 0.4733 - val_loss: 1.5032
Epoch 91/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.5284 - loss: 1.3186 - val_accuracy: 0.4875 - val_loss: 1.4798
Epoch 92/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5323 - loss: 1.3132 - val_accuracy: 0.4657 - val_loss: 1.5280
Epoch 93/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5296 - loss: 1.3059 - val_accuracy: 0.4851 - val_loss: 1.4890
Epoch 94/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5279 - loss: 1.3141 - val_accuracy: 0.4860 - val_loss: 1.4861
Epoch 95/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5347 - loss: 1.3013 - val_accuracy: 0.4672 - val_loss: 1.5172
Epoch 96/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.5318 - loss: 1.3065 - val_accuracy: 0.4858 - val_loss: 1.4848
Epoch 97/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.5329 - loss: 1.3031 - val_accuracy: 0.4822 - val_loss: 1.4976
Epoch 98/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5337 - loss: 1.3080 - val_accuracy: 0.5004 - val_loss: 1.4573
Epoch 99/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.5332 - loss: 1.3048 - val_accuracy: 0.4802 - val_loss: 1.5058
Epoch 100/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5381 - loss: 1.2995 - val_accuracy: 0.4749 - val_loss: 1.5145
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.5827 - loss: 1.2681
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.4764 - loss: 1.5122
```

3. Retirar as camadas de dropout e aplicar regularização L2

```
from tensorflow.keras.regularizers import l2

def modelo_residual_v3_l2(n1, n2, n3, n4, num_classes, input_shape=(32, 32, 3)):
    inputs = layers.Input(shape=input_shape)
    x = layers.Flatten()(inputs)

    x = layers.Dense(n1, activation='relu', kernel_regularizer=l2(0.001))(x)
    x = residual_block(x, n1)

    x = layers.Dense(n2, activation='relu', kernel_regularizer=l2(0.01))(x)
    x = residual_block(x, n2)

    x = layers.Dense(n3, activation='relu', kernel_regularizer=l2(0.01))(x)
    x = residual_block(x, n3)

    x = layers.Dense(n4, activation='relu', kernel_regularizer=l2(0.01))(x)
    x = residual_block(x, n4)
```

```
    # Camada de saída
    outputs = layers.Dense(num_classes, activation='softmax')(x)

    model = models.Model(inputs=inputs, outputs=outputs)
    return model

rna_v3 = modelo_residual_v3_l2(512, 512, 256, 128, 10)
rna_v3.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history_v3 = rna_v3.fit(x_train_norm, y_train_hot, epochs=100, batch_size=256, validation_data=(x_test_norm, y_test_hot)

train_loss_v3, train_accuracy_v3 = rna_v3.evaluate(x_train_norm, y_train_hot, verbose=1)
test_loss_v3, test_accuracy_v3 = rna_v3.evaluate(x_test_norm, y_test_hot, verbose=1)
```

```
Epoch 73/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.7361 - loss: 0.9246 - val_accuracy: 0.5052 - val_loss: 1.8371
Epoch 74/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.7436 - loss: 0.8980 - val_accuracy: 0.5175 - val_loss: 1.8712
Epoch 75/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7542 - loss: 0.8684 - val_accuracy: 0.5058 - val_loss: 1.8811
Epoch 76/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - accuracy: 0.7400 - loss: 0.9127 - val_accuracy: 0.5075 - val_loss: 1.8940
Epoch 77/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.7630 - loss: 0.8501 - val_accuracy: 0.5022 - val_loss: 1.9093
Epoch 78/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.7607 - loss: 0.8589 - val_accuracy: 0.4999 - val_loss: 1.9563
Epoch 79/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7563 - loss: 0.8689 - val_accuracy: 0.5009 - val_loss: 1.9229
Epoch 80/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7592 - loss: 0.8613 - val_accuracy: 0.5048 - val_loss: 1.9862
Epoch 81/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7667 - loss: 0.8438 - val_accuracy: 0.4754 - val_loss: 2.0472
Epoch 82/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7570 - loss: 0.8789 - val_accuracy: 0.5005 - val_loss: 1.9911
Epoch 83/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.7698 - loss: 0.8401 - val_accuracy: 0.5015 - val_loss: 1.9632
Epoch 84/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.7793 - loss: 0.8139 - val_accuracy: 0.5056 - val_loss: 1.9252
Epoch 85/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7807 - loss: 0.8123 - val_accuracy: 0.5068 - val_loss: 2.0324
Epoch 86/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.7647 - loss: 0.8538 - val_accuracy: 0.4973 - val_loss: 1.9942
Epoch 87/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7768 - loss: 0.8283 - val_accuracy: 0.4986 - val_loss: 2.0822
Epoch 88/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7812 - loss: 0.8119 - val_accuracy: 0.5017 - val_loss: 2.0172
Epoch 89/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7890 - loss: 0.7987 - val_accuracy: 0.4989 - val_loss: 2.0114
Epoch 90/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.7855 - loss: 0.8084 - val_accuracy: 0.4954 - val_loss: 2.0652
Epoch 91/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.7807 - loss: 0.8071 - val_accuracy: 0.5069 - val_loss: 2.0160
Epoch 92/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.7839 - loss: 0.8092 - val_accuracy: 0.4926 - val_loss: 2.0934
Epoch 93/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7875 - loss: 0.7995 - val_accuracy: 0.5048 - val_loss: 2.1199
Epoch 94/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8009 - loss: 0.7621 - val_accuracy: 0.5035 - val_loss: 2.1107
Epoch 95/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8022 - loss: 0.7531 - val_accuracy: 0.5005 - val_loss: 2.0969
Epoch 96/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7969 - loss: 0.7702 - val_accuracy: 0.5055 - val_loss: 2.0226
Epoch 97/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8081 - loss: 0.7523 - val_accuracy: 0.5007 - val_loss: 2.1214
Epoch 98/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.7957 - loss: 0.7754 - val_accuracy: 0.4984 - val_loss: 2.0893
Epoch 99/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8107 - loss: 0.7407 - val_accuracy: 0.5001 - val_loss: 2.2239
Epoch 100/100
196/196 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8023 - loss: 0.7600 - val_accuracy: 0.4867 - val_loss: 2.1547
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.7984 - loss: 0.7731
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.4849 - loss: 2.1471
```

Discussão: