T2_Transferencia_aprendizado

November 22, 2020

Trabalho #2 - Transferência de Aprendizado

Nesse trabalho você vai utilizar uma RNA pré-treinada para realizar uma tarefa de classificação de múltiplas classes. A tarefa é determinar o tipo de objeto mostrado em imagens dentro de 10 classes possíveis. A rede pré-treinada que iremos utilizar é a VGG16 que está disponível no Keras.

Coloque o seu nome aqui

Nome: Bruno Rodrigues Silva

Em primeiro lugar é necessário importar alguns pacotes do Python que serão usados ao longo nesse trabalho: - numpy pacote de cálculo científico com Python - matplotlib biblioteca para gerar gráficos em Python - TensorFlow

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

1 Visão geral do problema

Nesse trabalho iremos usar o conjunto dados CIFAR-10. Esses dados consistem de um subconjunto de uma coleção de 80 milhões de imagens pequenas obtidas por Alex Krizhevsky, Vinod Nair e Geoffrey Hinton, disponível em https://www.cs.toronto.edu/~kriz/cifar.html.

O conjunto de dados CIFAR-10 consiste de 60.000 imagens coloridas com dimensão 32x32 divididas em 10 classes. Existem 50.000 imagens de treinamento e 10.000 imagens de teste.

As classes de objetos presentes nas imagens são as seguintes:

0. Avião

1. Automóvel
2. Pássaro
3. Gato
4. Veado
5. Cachorro
6. Rã
7. cavalo

8. Navio
9. Caminhão

Não existe nenhuma imagem que contém objetos de mais de uma classe. Não há sobreposição entre automóveis e caminhões. "Automóvel" inclui sedans, SUVs etc. "Caminhão" inclui apenas caminhões grandes, não inclui picapes.

Assim, o objetivo desse problema é desenvolver um RNA que receba como entrada uma imagem e calcula a probabilidade de mostrar um determinado objeto das 10 classes possíveis.

2 Dados de treinamento

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

Características dos dados:

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

2.1 Leitura dos dados

Para iniciar o trabalho é necessário ler o arquivo de dados. Assim, execute o código da célula abaixo para ler o arquivo de dados.

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

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

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

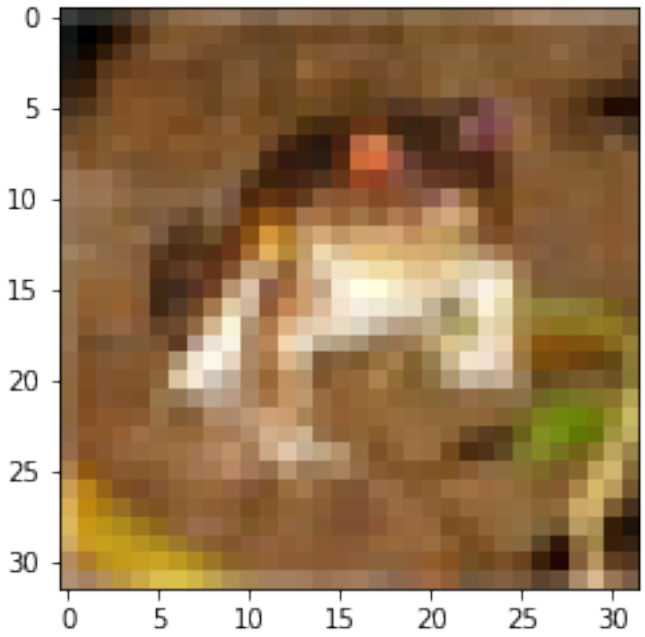
Pela dimensão dos tensores com os dados de treinamento e teste temos:

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

Execute a célula a seguir para visualizar um exemplo de uma imagem do banco de dados juntamente com a sua classe. Altere o valor da variável 'index' e execute a célula novamente para visualizar mais exemplos diferentes.

```
[ ]: # Examplo de uma imagem
index = 0
plt.imshow(x_train_orig[index])
print ("y = " + str(np.squeeze(y_train_orig[index])))
```

```
y = 6
```



2.2 Processamento dos dados

Para os dados poderem ser usados para o desenvolvimento da RNA devemos primeiramente processá-los.

Para isso devemos realizar as seguintes etapas:

- Dividir os dados de treinamento nos conjuntos de treinamento e validação;
- Os valores dos pixels em uma imagem é um número inteiro que deve ser transformado em número real para ser usado em cálculos;
- Normalizar as imagens de forma que os valores dos pixels fique entre 0 e 1.

2.3 Exercício #1: Divisão do conjunto de dados de treinamento

Como o conjunto de imagens é muito grande e demoraria muito tempo para treinar com todos esses dados, nesse trabalho usaremos somente metade dos dados. Assim, crie na célula abaixo um código para:

1. Dividir o conjunto de dados de treinamento nos conjuntos de treinamento e validação. Nessa divisão pegue os primeiros 30.000 exemplos para o conjunto de treinamento e os próximos 6.000 para o conjunto de validação.
2. Selecione os primeiros 6.000 exemplos do conjunto de teste para formar o novo conjunto de teste.
3. Não se esqueça de fazer a mesma selação para as saídas.

[ ]:
```
# PARA VOCÊ FAZER: divisão do conjunto de dados de treinamento

# Dados de entrada
# Inclua seu código aqui
x_train = x_train_orig[:30000]
x_val = x_train_orig[30000:36000]
x_test = x_test_orig[:6000]

# Dados de saída
# Inclua seu código aqui
#
y_train = y_train_orig[:30000]
y_val = y_train_orig[30000:36000]
y_test = y_test_orig[:6000]

print("Dimensão do tensor de dados de entrada de treinamento =", x_train.shape)
print("Dimensão do tensor de dados de entrada de validação =", x_val.shape)
print("Dimensão do tensor de dados de entrada de teste =", x_test.shape)
print("Dimensão do tensor de dados de saída de treinamento =", y_train.shape)
print("Dimensão do tensor de dados de saída de validação =", y_val.shape)
print("Dimensão do tensor de dados de saída de teste =", y_test.shape)
```

```
Dimensão do tensor de dados de entrada de treinamento = (30000, 32, 32, 3)
Dimensão do tensor de dados de entrada de validação = (6000, 32, 32, 3)
Dimensão do tensor de dados de entrada de teste = (6000, 32, 32, 3)
Dimensão do tensor de dados de saída de treinamento = (30000, 1)
Dimensão do tensor de dados de saída de validação = (6000, 1)
Dimensão do tensor de dados de saída de teste = (6000, 1)
```

Saída desejada:

```
Dimensão do tensor de dados de entrada de treinamento = (30000, 32, 32, 3)
Dimensão do tensor de dados de entrada de validação = (6000, 32, 32, 3)
Dimensão do tensor de dados de entrada de teste = (6000, 32, 32, 3)
Dimensão do tensor de dados de saída de treinamento = (30000, 1)
Dimensão do tensor de dados de saída de validação = (6000, 1)
Dimensão do tensor de dados de saída de teste = (6000, 1)
```

2.4 Exercício #2: Normalização dos dados de entrada

na célula abaixo normalize e transforme as imagens em números reais dividindo por 255.

[ ]:
```
# PARA VOCÊ FAZER: normalização dos dados de entrada

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

# Transformação dos dados em números reais
# Inclua seu código aqui
#
x_train = x_train/255.
x_val = x_val/255.
x_test = x_test/255.
# Apresentação de alguns resultados para verificação
print('Alguns elementos de x_train:', x_train[0,0,0,:])
print('Alguns elementos de y_val:', x_val[0,0,0,:])
print('Alguns elementos de x_test:', x_test[0,0,0,:])
```

```
Dimensão das imagens de entrada= (32, 32, 3)
Alguns elementos de x_train: [0.23137255 0.24313725 0.24705882]
Alguns elementos de y_val: [0.69803922 0.69019608 0.74117647]
Alguns elementos de x_test: [0.61960784 0.43921569 0.19215686]
```

Saída esperada:

```
Dimensão das imagens de entrada= (32, 32, 3)
Alguns elementos de x_train: [0.23137255 0.24313726 0.24705882]
Alguns elementos de y_val: [0.69803923 0.6901961  0.7411765 ]
Alguns elementos de x_test: [0.61960787 0.4392157  0.19215687]
```

2.5 Exercício #3: Codificação das classes

As classes dos sobjetos são identificadas por um número inteiro que varia de 0 a 9. Porém, a saída esperada de uma RNA para um problema de classificação de múltiplas classes é um vetor de dimensão igual ao número de classes, que no caso são 10 classes. Cada elemento desse vetor representa a probabilidade da imagem ser um sinal. Assim, devemos transformar as saídas reais do conjunto de dados em um vetor linha de 10elementos, com todos os elementos iguais a zero a menos do correspondente ao da classe do sinal, que deve ser igual a um.

Como já visto, a função que realiza essa transformação é conhecida na literatura de `one-hot-encoding`, que no Keras é chamada de `to_categorical`. Na célula abaixo realize essa transfromação.

[ ]:
```
# PARA VOCÊ FAZER: codificação das classes

# Importa classe de utilidades do Keras
# Inclua seu código aqui
#
from keras.utils import to_categorical
# Transformação das classes de números reais para vetores
```

```
# Inclua seu código aqui
#
y_train_hot = to_categorical(y_train)
y_val_hot = to_categorical(y_val)
y_test_hot = to_categorical(y_test)

print('Dimensão dos dados de saída do conjunto de treinamento: ', y_train_hot.
 ↪shape)
print('Dimensão dos dados de saída do conjunto de validação: ', y_val_hot.shape)
print('Dimensão dos dados de saída do conjunto de teste: ', y_test_hot.shape)
```

```
Dimensão dos dados de saída do conjunto de treinamento:  (30000, 10)
Dimensão dos dados de saída do conjunto de validação:  (6000, 10)
Dimensão dos dados de saída do conjunto de teste:  (6000, 10)
```

Saída esperada:

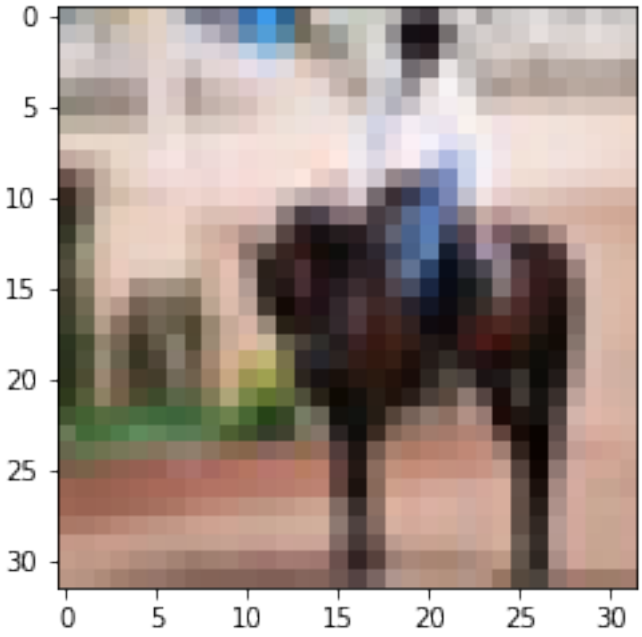
```
Dimensão dos dados de saída do conjunto de treinamento:  (30000, 10)
Dimensão dos dados de saída do conjunto de validação:  (6000, 10)
Dimensão dos dados de saída do conjunto de teste:  (6000, 10)
```

Visualização da entrada e saída correspondente

Execute a célula abaixo para verificar se o programa realizou de fato o que era esperado. No código abaixo index é o número sequencial da imagem. Tente trocar a imagem, mudando o index, usando valores entre 0 e 959, para visualizar outros exemplos.

[ ]:
```
# Exemplo de saída
index = 11
print("Classe numérica: ", y_train[index], ", Vetor de saída correspondentes: ",␣
 ↪y_train_hot[index])
plt.imshow(x_train_orig[index])
plt.show()
```

```
Classe numérica:  [7] , Vetor de saída correspondentes:  [0. 0. 0. 0. 0. 0. 0.
1. 0. 0.]
```



3 RNA convolucional base

Nesse trabalho você irá usar uma RNA convolucional já treinada como base para criar outra RNA para realizar uma tarefa diferente da que a RNA base foi treinada. A RNA que será usada como base é a VGG16 vista em aula (Simonyan & Zisserman, Very deep convolutional networks for large-scale image recognition, 2015).

A VGG16 foi desenvolvida para classificação de múltiplas classes com 1.000 classes. Ela é uitlizada para reconhecimento de objetos em imagens. A arquitetura da VGG é muito simples, sendo composta pela repetição de camadas convolucionais formando blocos. Cada bloco é composto por duas ou três camadas convolucionais, com filtros 3x3, stride = 1 e "same convolution", seguida por uma camada de max-pooling, com janela 2x2 e stride = 2. A VGG16 mantém o mesmo padrão em todos os blocos dobrando o número de filtros a cada bloco. Apesar da VGG16 possuir muitos parâmetros, cerca de 138 milhões, ela é muito simples.

3.1 Exercício #4: Carregar a VGG16

O TensorFlow-Keras possui na sua base de dados a RNA VGG16 treinada com o banco de imagens imagenet. Complete a célula abaixo para carregar a VGG16. Não se esqueça de excluir a parte densa e de definir a dimensão das imagens ao salvar a VGG16.

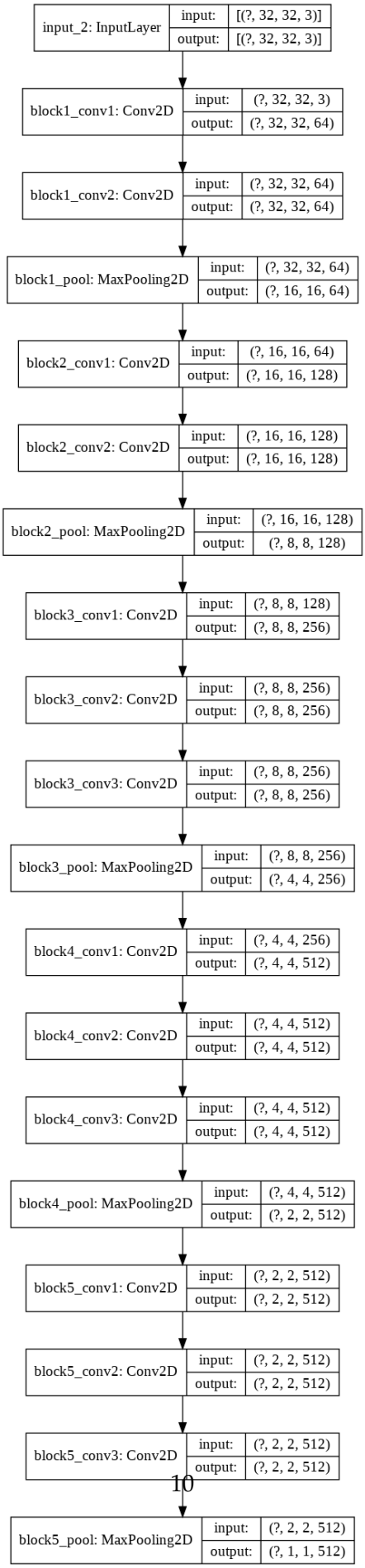
[ ]:
```
# PARA VOCÊ FAZER: carregar e salvar a VGG16 na rna_base

# Importa função para fazer gráfico de RNAs
# Inclua seu código aqui
#
```

```python
from keras.utils import plot_model
from keras.applications import VGG16
# Carrega e salva a VGG16 excluindo suas camadas densas
# Inclua seu código aqui
#
rna_base = VGG16(include_top=False, input_shape=(32, 32, 3))
# Mostra a arquitetura da VGG16
rna_base.summary()

# Cria um arquivo com o esquema da VGG16
plot_model(rna_base, to_file='VGG16.png', show_shapes=True)
```

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 32, 32, 3)]       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 32, 32, 64)        1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 32, 32, 64)        36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 16, 16, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 16, 16, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 16, 16, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 8, 8, 128)         0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 8, 8, 256)         295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 8, 8, 256)         590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 8, 8, 256)         590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 4, 4, 256)         0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 4, 4, 512)         1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 4, 4, 512)         2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 4, 4, 512)         2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 2, 2, 512)         0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 2, 2, 512)         2359808
```

```
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 2, 2, 512)         2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 2, 2, 512)         2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 1, 1, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

[ ]:


input_2: InputLayer
input: [(?, 32, 32, 3)]
output: [(?, 32, 32, 3)]
block1_conv1: Conv2D
input: (?, 32, 32, 3)
output: (?, 32, 32, 64)
block1_conv2: Conv2D
input: (?, 32, 32, 64)
output: (?, 32, 32, 64)
block1_pool: MaxPooling2D
input: (?, 32, 32, 64)
output: (?, 16, 16, 64)
block2_conv1: Conv2D
input: (?, 16, 16, 64)
output: (?, 16, 16, 128)
block2_conv2: Conv2D
input: (?, 16, 16, 128)
output: (?, 16, 16, 128)
block2_pool: MaxPooling2D
input: (?, 16, 16, 128)
output: (?, 8, 8, 128)
block3_conv1: Conv2D
input: (?, 8, 8, 128)
output: (?, 8, 8, 256)
block3_conv2: Conv2D
input: (?, 8, 8, 256)
output: (?, 8, 8, 256)
block3_conv3: Conv2D
input: (?, 8, 8, 256)
output: (?, 8, 8, 256)
block3_pool: MaxPooling2D
input: (?, 8, 8, 256)
output: (?, 4, 4, 256)
block4_conv1: Conv2D
input: (?, 4, 4, 256)
output: (?, 4, 4, 512)
block4_conv2: Conv2D
input: (?, 4, 4, 512)
output: (?, 4, 4, 512)
block4_conv3: Conv2D
input: (?, 4, 4, 512)
output: (?, 4, 4, 512)
block4_pool: MaxPooling2D
input: (?, 4, 4, 512)
output: (?, 2, 2, 512)
block5_conv1: Conv2D
input: (?, 2, 2, 512)
output: (?, 2, 2, 512)
block5_conv2: Conv2D
input: (?, 2, 2, 512)
output: (?, 2, 2, 512)
block5_conv3: Conv2D
input: (?, 2, 2, 512)
output: (?, 2, 2, 512)
block5_pool: MaxPooling2D
input: (?, 2, 2, 512)
output: (?, 1, 1, 512)

Saída esperada:

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 32, 32, 3)]       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 32, 32, 64)        1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 32, 32, 64)        36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 16, 16, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 16, 16, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 16, 16, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 8, 8, 128)         0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 8, 8, 256)         295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 8, 8, 256)         590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 8, 8, 256)         590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 4, 4, 256)         0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 4, 4, 512)         1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 4, 4, 512)         2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 4, 4, 512)         2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 2, 2, 512)         0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 2, 2, 512)         2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 2, 2, 512)         2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 2, 2, 512)         2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 1, 1, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
```

```
Non-trainable params: 0
_________________________________________________________________
```

Observações:

- A parte convolucional da VGG16 possui 5 blocos, sendo que cada bloco possui 2 ou 3 camadas convolucionais seguidas por uma camada de max-pooling.
- A VGG16 não é uma rede muito profunda, possuindo 13 camadas convolucionais.
- Apesar do grande número de parâmetros (138 milhões para a VGG16 completa com as suas camadas densas), a VGG16 é uma rede muito simples e rápida de ser excecutada e treinada, quando comparada com outras RNAs de mesmo desempenho.
- O esquema da VGG16 que você carregou foi gerado no arquivo VGG16.png que está no diretório que você está utilizando. Para ver esse esquema basta abrir esse arquivo.

4 4 - Uso da RNA base para extrair caracteríticas

Nessa parte do trabalho você vai usar a RNA base somente para extrair as caracteríticas das imagens de treinamento, validação e teste. Essas caracterísitcas serão usadas como entrada de uma nova RNA densa que será treinada para realizar a tarefa de classificação dos objetos.

4.1 Exercício #5: Extração das características das imagens

Na célula abaixo crie um código para executar a rna_base na forma de previsão para extrair as características das imagens de treinamento, validação e teste.

[ ]:
```
# PARA VOCÊ FAZER: gerar as características das imagens de treinamento,␣
 ↪validação e teste pela rna_base.

# Inclua seu código aqui
#
train_features = rna_base.predict(x_train)
val_features = rna_base.predict(x_val)
test_features = rna_base.predict(x_test)

print("Dimensão do tensor de características das imagens de treinamento = ",␣
 ↪train_features.shape)
print("Dimensão do tensor de características das imagens de validação = ",␣
 ↪val_features.shape)
print("Dimensão do tensor de características das imagens de teste = ",␣
 ↪test_features.shape)
```

```
Dimensão do tensor de características das imagens de treinamento =  (30000, 1,
1, 512)
Dimensão do tensor de características das imagens de validação =  (6000, 1, 1,
512)
Dimensão do tensor de características das imagens de teste =  (6000, 1, 1, 512)
```

Saída esperada:

```
Dimensão do tensor de características das imagens de treinamento =  (30000, 1, 1, 512)
Dimensão do tensor de características das imagens de validação =  (6000, 1, 1, 512)
Dimensão do tensor de características das imagens de teste =  (6000, 1, 1, 512
```

Observe que as características extraídas de cada imagem tem dimensão (1, 1, 512), que é a dimensão do tensor de saída da rna_base.

4.2 Exercício #6: Redimensionamento dos tensores de características

Como a RNA de classificação é composta por camadas densas, então temos que redimensionar os dados de entrada para transformá-los em um vetor. Crie na célula abaixo um código que realiza o redimensionamento das características usando a função reshape da biblioteca Numpy.

[ ]:
```
# PARA VOCÊ FAZER: redimensionamento dos tensores de características

# Recupera dimensões dos tensores de características
m, nlin, ncol, nfeat = train_features.shape
m_val = val_features.shape[0]
m_test = test_features.shape[0]

# Redimensiona tensores de características
# Inclua seu código aqui
#

train_carac = np.reshape(train_features, (m, nlin*ncol*nfeat))
val_carac = np.reshape(val_features, (m_val, nlin*ncol*nfeat))
test_carac = np.reshape(test_features, (m_test, nlin*ncol*nfeat))

print("Dimensão do tensor de características das imagens de treinamento = ",␣
 ↪train_carac.shape)
print("Dimensão do tensor de características das imagens de validação = ",␣
 ↪val_carac.shape)
print("Dimensão do tensor de características das imagens de teste = ",␣
 ↪test_carac.shape)
print("Cinco primeiros elementos da 1a linha dos dados de treinamento: ",␣
 ↪train_carac[0,0:5])
print("Cinco primeiros elementos da 1a linha dos dados de validação: ",␣
 ↪val_carac[0,0:5])
print("Cinco primeiros elementos da 1a linha dos dados de teste: ",␣
 ↪test_carac[0,0:5])
```

```
Dimensão do tensor de características das imagens de treinamento =  (30000, 512)
Dimensão do tensor de características das imagens de validação =  (6000, 512)
Dimensão do tensor de características das imagens de teste =  (6000, 512)
Cinco primeiros elementos da 1a linha dos dados de treinamento:  [0.10176692 0.
0.8440498  0.5943726  0.01328954]
```

```
Cinco primeiros elementos da 1a linha dos dados de validação:  [0.         0.
0.         0.47797072 0.        ]
Cinco primeiros elementos da 1a linha dos dados de teste:  [0.28653282 0.
1.2196431  0.19872217 0.20980254]
```

Saída esperada:

```
Dimensão do tensor de características das imagens de treinamento =  (30000, 512)
Dimensão do tensor de características das imagens de validação =  (6000, 512)
Dimensão do tensor de características das imagens de teste =  (6000, 512)
Cinco primeiros elementos da 1a linha dos dados de treinamento:  [0.10176882 0.         0.844049
Cinco primeiros elementos da 1a linha dos dados de validação:  [0.         0.         0.
Cinco primeiros elementos da 1a linha dos dados de teste:  [0.28653017 0.         1.2196395  0.1
```

4.3 Exercício #7: Configuração da RNA para classificação

Para realizar a classificação dos objetos nas imagens, você vai usar uma RNA com 2 camadas densas (uma camada intermediária e uma de saída) com as seguintes características:

- Tensor de entrada: tensor com os vetores de características das imagens, criado no exercício #5 anterior;
- Camada intermediária: número de neurônios 32, função de ativação ReLu;
- Camada de saída: número neurônios 10, função de ativação softmax;
- Incluir Dropout após a camada intermediária com limitação da norma dos pesos das ligações igual a 3.0;
- Use uma fração de Dropout de 0.2.

Na célula abaixo crie a sua RNA.

[ ]:
```
# PARA VOCÊ FAZER: criação da RNA para classificação

# Importa do Keras classes de modelos e de camadas
# Inclua seu código aqui
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras.constraints import max_norm

# Define dimensão do vetor de entrada
# Inclua seu código aqui
#
vetor_dim = (nlin*ncol*nfeat, )

# Define fração de Dropout
# Inclua seu código aqui
#
frac = 0.2
# Configuração da RNA
# Inclua seu código aqui
#
```

```
rna_class = models.Sequential()
rna_class.add(layers.Dense(32, activation='relu', input_shape=vetor_dim,␣
 ↪kernel_constraint=max_norm(3.0)))
rna_class.add(layers.Dropout(frac))
rna_class.add(layers.Dense(10, activation='softmax'))
rna_class.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 32)                16416
_________________________________________________________________
dropout_2 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                330
=================================================================
Total params: 16,746
Trainable params: 16,746
Non-trainable params: 0
_________________________________________________________________
```

Saída esperada:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 32)                16416
_________________________________________________________________
dropout (Dropout)            (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                330
=================================================================
Total params: 16,746
Trainable params: 16,746
Non-trainable params: 0
_________________________________________________________________
```

4.4 Exercício #8: Compilação e treinamento da RNA

Agora você vai treinar a sua RNA usando o método de otimização Adams. Assim, na célula abaixo, compile e treine a sua RNA usando os seguinte hiperparâmetros:

- método Adam;
- taxa de aprendizagem = 0.001;
- número de épocas = 50;
- verbose = 1.

[ ]:
```
# PARA VOCÊ FAZER: compilação e treinamento da RNA usando o método Adam

# importa do keras a classe dos otimizadores
# Inclua seu código aqui
#
from keras.optimizers import Adam

# Configuração do otimizador
# Inclua seu código aqui
#
opt = Adam(0.001)
rna_class.compile(opt, 'categorical_crossentropy', metrics='accuracy')
# Treinamento da RNA (salve o resultado do treinamento no dicionário history)
# Inclua seu código aqui
#
history = rna_class.fit(train_carac, y_train_hot, batch_size=32, epochs=50,␣
 ↪validation_data=(val_carac, y_val_hot))
```

```
Epoch 1/50
938/938 [==============================] - 3s 3ms/step - loss: 1.4443 -
accuracy: 0.4877 - val_loss: 1.3505 - val_accuracy: 0.5287
Epoch 2/50
938/938 [==============================] - 3s 3ms/step - loss: 1.3904 -
accuracy: 0.5064 - val_loss: 1.3006 - val_accuracy: 0.5450
Epoch 3/50
938/938 [==============================] - 3s 3ms/step - loss: 1.3542 -
accuracy: 0.5240 - val_loss: 1.2785 - val_accuracy: 0.5512
Epoch 4/50
938/938 [==============================] - 3s 3ms/step - loss: 1.3311 -
accuracy: 0.5307 - val_loss: 1.2640 - val_accuracy: 0.5620
Epoch 5/50
938/938 [==============================] - 3s 3ms/step - loss: 1.3123 -
accuracy: 0.5398 - val_loss: 1.2558 - val_accuracy: 0.5572
Epoch 6/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2963 -
accuracy: 0.5437 - val_loss: 1.2576 - val_accuracy: 0.5630
Epoch 7/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2892 -
accuracy: 0.5496 - val_loss: 1.2342 - val_accuracy: 0.5688
Epoch 8/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2812 -
accuracy: 0.5510 - val_loss: 1.2437 - val_accuracy: 0.5680
Epoch 9/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2676 -
accuracy: 0.5568 - val_loss: 1.2313 - val_accuracy: 0.5695
Epoch 10/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2621 -
```

```
accuracy: 0.5573 - val_loss: 1.2262 - val_accuracy: 0.5702
Epoch 11/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2560 -
accuracy: 0.5617 - val_loss: 1.2325 - val_accuracy: 0.5738
Epoch 12/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2534 -
accuracy: 0.5590 - val_loss: 1.2301 - val_accuracy: 0.5710
Epoch 13/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2466 -
accuracy: 0.5623 - val_loss: 1.2158 - val_accuracy: 0.5782
Epoch 14/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2420 -
accuracy: 0.5633 - val_loss: 1.2230 - val_accuracy: 0.5787
Epoch 15/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2389 -
accuracy: 0.5660 - val_loss: 1.2276 - val_accuracy: 0.5693
Epoch 16/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2369 -
accuracy: 0.5661 - val_loss: 1.2330 - val_accuracy: 0.5678
Epoch 17/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2335 -
accuracy: 0.5697 - val_loss: 1.2141 - val_accuracy: 0.5812
Epoch 18/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2337 -
accuracy: 0.5683 - val_loss: 1.2227 - val_accuracy: 0.5725
Epoch 19/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2275 -
accuracy: 0.5667 - val_loss: 1.2320 - val_accuracy: 0.5727
Epoch 20/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2266 -
accuracy: 0.5701 - val_loss: 1.2298 - val_accuracy: 0.5732
Epoch 21/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2257 -
accuracy: 0.5673 - val_loss: 1.2068 - val_accuracy: 0.5805
Epoch 22/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2273 -
accuracy: 0.5683 - val_loss: 1.2324 - val_accuracy: 0.5673
Epoch 23/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2243 -
accuracy: 0.5683 - val_loss: 1.2182 - val_accuracy: 0.5735
Epoch 24/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2241 -
accuracy: 0.5685 - val_loss: 1.2356 - val_accuracy: 0.5733
Epoch 25/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2210 -
accuracy: 0.5693 - val_loss: 1.2258 - val_accuracy: 0.5730
Epoch 26/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2200 -
```

```
accuracy: 0.5715 - val_loss: 1.2203 - val_accuracy: 0.5738
Epoch 27/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2185 -
accuracy: 0.5721 - val_loss: 1.2379 - val_accuracy: 0.5743
Epoch 28/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2185 -
accuracy: 0.5713 - val_loss: 1.2096 - val_accuracy: 0.5775
Epoch 29/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2224 -
accuracy: 0.5705 - val_loss: 1.2156 - val_accuracy: 0.5770
Epoch 30/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2181 -
accuracy: 0.5701 - val_loss: 1.2158 - val_accuracy: 0.5717
Epoch 31/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2155 -
accuracy: 0.5690 - val_loss: 1.2167 - val_accuracy: 0.5700
Epoch 32/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2171 -
accuracy: 0.5716 - val_loss: 1.2250 - val_accuracy: 0.5808
Epoch 33/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2110 -
accuracy: 0.5753 - val_loss: 1.2206 - val_accuracy: 0.5733
Epoch 34/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2124 -
accuracy: 0.5723 - val_loss: 1.2197 - val_accuracy: 0.5782
Epoch 35/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2069 -
accuracy: 0.5721 - val_loss: 1.2170 - val_accuracy: 0.5803
Epoch 36/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2090 -
accuracy: 0.5738 - val_loss: 1.2247 - val_accuracy: 0.5730
Epoch 37/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2114 -
accuracy: 0.5733 - val_loss: 1.2319 - val_accuracy: 0.5687
Epoch 38/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2140 -
accuracy: 0.5729 - val_loss: 1.2259 - val_accuracy: 0.5705
Epoch 39/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2111 -
accuracy: 0.5742 - val_loss: 1.2100 - val_accuracy: 0.5785
Epoch 40/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2095 -
accuracy: 0.5734 - val_loss: 1.2216 - val_accuracy: 0.5735
Epoch 41/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2120 -
accuracy: 0.5739 - val_loss: 1.2325 - val_accuracy: 0.5662
Epoch 42/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2101 -
```

```
accuracy: 0.5728 - val_loss: 1.2488 - val_accuracy: 0.5605
Epoch 43/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2134 -
accuracy: 0.5739 - val_loss: 1.2115 - val_accuracy: 0.5808
Epoch 44/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2144 -
accuracy: 0.5713 - val_loss: 1.2174 - val_accuracy: 0.5763
Epoch 45/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2126 -
accuracy: 0.5717 - val_loss: 1.2582 - val_accuracy: 0.5618
Epoch 46/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2066 -
accuracy: 0.5748 - val_loss: 1.2235 - val_accuracy: 0.5748
Epoch 47/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2090 -
accuracy: 0.5753 - val_loss: 1.2112 - val_accuracy: 0.5775
Epoch 48/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2066 -
accuracy: 0.5726 - val_loss: 1.2450 - val_accuracy: 0.5608
Epoch 49/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2051 -
accuracy: 0.5749 - val_loss: 1.2220 - val_accuracy: 0.5795
Epoch 50/50
938/938 [==============================] - 3s 3ms/step - loss: 1.2084 -
accuracy: 0.5721 - val_loss: 1.2305 - val_accuracy: 0.5758
```

Saída esperada:

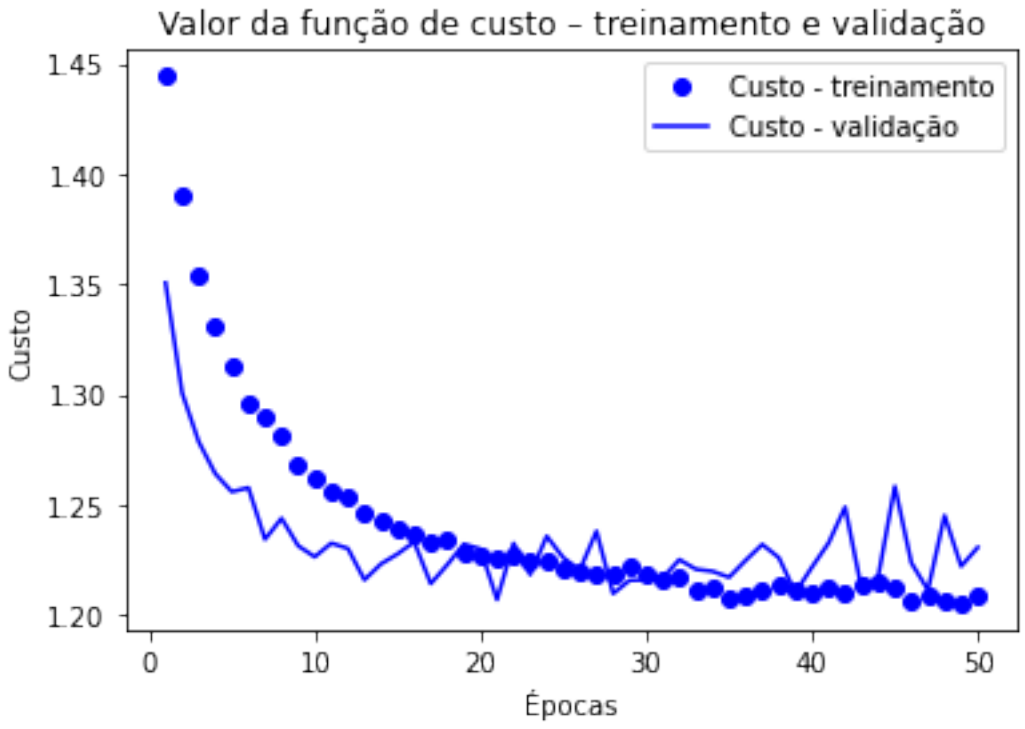
```
Epoch 1/50
938/938 [==============================] - 3s 3ms/step - loss: 1.6224 - accuracy: 0.4290 - val_l
.
.
.
Epoch 50/50
938/938 [==============================] - 3s 3ms/step - loss: 1.0894 - accuracy: 0.6143 - val_l
```

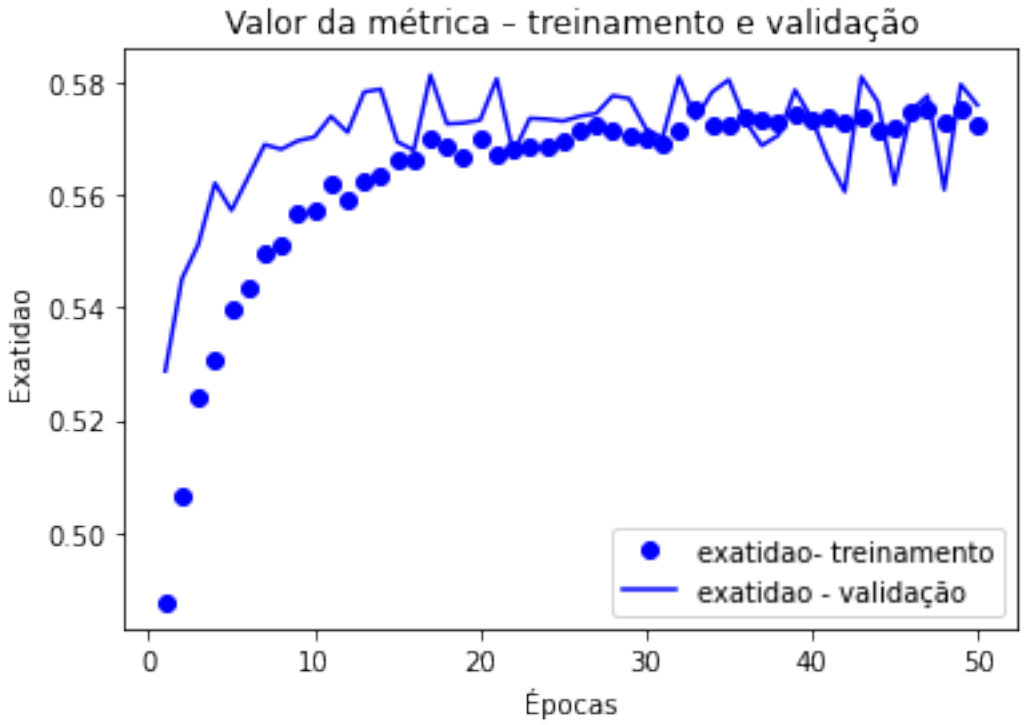
4.5 Visualização dos resultados

Execute a célula a seguir para fazer os gráficos da função de custo e da métrica para os dados de treinamento e validação.

[ ]:
```python
# Define função para fazer os gráficos do treinamento
def plot_results_train(history):
    # Salva treinamento na variável history para visualização
    history_dict = history.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, 'bo', label='Custo - treinamento')
    plt.plot(epocas, val_custo, 'b', label='Custo - validação')
    plt.title('Valor da função de custo - treinamento e validação')
    plt.xlabel('Épocas')
    plt.ylabel('Custo')
    plt.legend()
    plt.show()

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

# Realiza os gráficos chamando função plot_results_train
plot_results_train(history)
```





4.6 Exercício #9: Cálculo dos custo e das métricas

Na célula abaixo crie um código para calcular os valores do custo e da exatidão para os dados de treinamento, validação e teste.

[ ]:
```
# PARA VOCÊ FAZER: Usando o método evaluate calcule o custo e exatidão␣
 ↪resultantes do treinamento

#Calculo do custo e exatidão para os dados de treinamento, validação e teste
# Inclua seu código aqui
#
train_eval = rna_class.evaluate(train_carac, y_train_hot, batch_size=32)
val_eval = rna_class.evaluate(val_carac, y_val_hot, batch_size=32)
test_eval = rna_class.evaluate(test_carac, y_test_hot, batch_size=32)
```

```
938/938 [==============================] - 2s 2ms/step - loss: 1.0644 -
accuracy: 0.6277
188/188 [==============================] - 0s 2ms/step - loss: 1.2305 -
accuracy: 0.5758
188/188 [==============================] - 0s 2ms/step - loss: 1.2179 -
accuracy: 0.5763
```

Saída esperada:

```
938/938 [==============================] - 2s 2ms/step - loss: 0.9416 - accuracy: 0.6722
188/188 [==============================] - 0s 2ms/step - loss: 1.1828 - accuracy: 0.5907
188/188 [==============================] - 0s 2ms/step - loss: 1.1713 - accuracy: 0.5853
```

Análise dos resultados:

Pelos gráficos da função de custo e da métrica você deve observar o seguinte:

- Os resultados obtidos são ruins tanto para os dados de treinamento como para os dados de validação e teste.
- Mesmo com essa RNA tão "pequena" ocorreu um pouco de overfitting no treinamento.

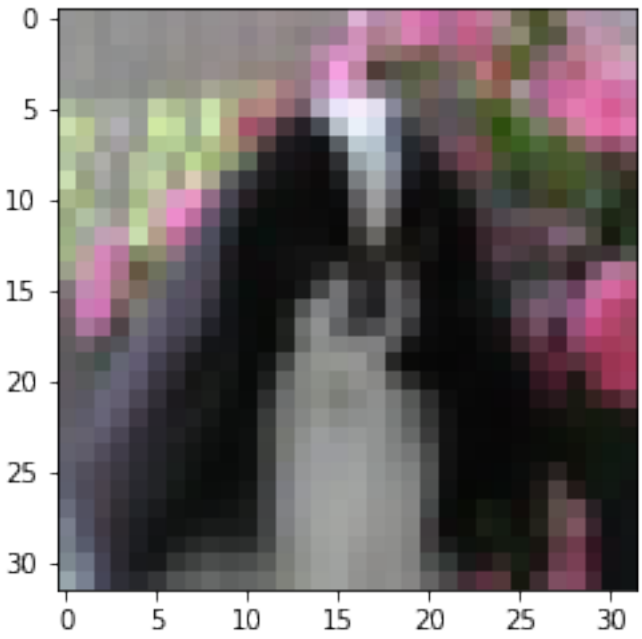
Após entregar o seu trabalho, tente alterar a fração de dropout e treinar por mais épocas para ver o que acontece.

4.7 Verificação dos resultados

Execute a célula abaixo para calcular as previsões da sua RNA para as imagens dos dados de teste e depois verificar se algumas dessas previsões estão corretas. Troque a variável index (variando entre 0 e 5.999) para verificar se a sua RNA consegue classificar corretamente o sinal de mão mostrado nas imagens.

[ ]:
```
# Cálculo das classes previstas

# Calculo das previsões da RNA
y_pred = rna_class.predict(test_carac)

# Cálculo das classes previstas
classe = np.argmax(y_pred, axis=1)

# Exemplo de uma imagem dos dados de teste
index = 42
plt.imshow(x_test_orig[index])
print ("classe prevista = " + str(np.squeeze(classe[index])))
print ("classe real = " + str(np.squeeze(y_test[index])))
```

```
classe prevista = 7
classe real = 5
```



5 RNA completa para processamento das imagens e classificação

O segundo método de realizar transferência de treinamento é mais demorado e computacionalmente mais exigente, mas os resultados são melhores. Esse método consiste em estender a rna_base adicionando as camadas densas para classificação e treinar parcialmente a RNA resultante. A rna_base é adicionada como se fosse uma camada de uma RNA sequencial da mesma forma como adicionamos qualquer tipo de camada. Esse método permite criar uma nova RNA completa e, assim, obter resultados melhores do que somente usar a rna_base para extrair características.

5.1 Exercício #10: Configuração da RNA completa usando a `rna_base`

Na célula abaixo crie um código para configurar uma nova RNA completa tendo como camadas convolucionais iniciais a rna_base e uma camada densas na saída. As características dessa RNA são as seguintes:

- Tensor de entrada: tensor com as imagens;
- Dimensão das imagens de entrada está na variável image_dim;
- Camadas convolucionais: rna_base;
- Camada de saída: número neurônios 10, função de ativação softmax;

Bloco com recuo

[ ]:
```
# PARA VOCÊ FAZER: criação da RNA completa

# Importa classe dos regularizadores
# Inclua seu código aqui
```

```
#
from keras import regularizers
from tensorflow.keras import layers
# Inicia RNA sequencial com a rna_base e adiciona as camadas de flattenning e␣
 ↪densas
# Inclua seu código aqui
#
rna = models.Sequential()
rna.add(rna_base)
rna.add(layers.Flatten())
rna.add(layers.Dense(10, activation='softmax'))
# Visualização da arquitetura da rede
rna.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 1, 1, 512)         14714688
_________________________________________________________________
flatten_1 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_5 (Dense)              (None, 10)                5130
=================================================================
Total params: 14,719,818
Trainable params: 14,719,818
Non-trainable params: 0
_________________________________________________________________
```

Saída esperada:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 1, 1, 512)         14714688
_________________________________________________________________
flatten (Flatten)            (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
=================================================================
Total params: 14,719,818
Trainable params: 14,719,818
Non-trainable params: 0
_________________________________________________________________
```

5.2 Exercício #11: Congelamento dos parâmetros da `rna_base`

Antes de compilar e treinar essa nova RNA é muito importante ‘‘congelar’’ os parâmetros da rna_base e depois descongelar somente as camadas que queremos retreinar. ‘‘Congelar’’ uma camada, ou um conjunto de camadas significa impedir que os seus parâmetros sejam atualizados durante o treinamento.

Na célula abaixo crie um código que congela os parâmetros da rna_base definindo o seu atributo trainable igual a False (ver notas de aula).

[ ]:
```
# PARA VOCÊ FAZER: congelamento dos parâmetros da rna_base

# Número de parâmetros a serem treinados antes do congelamento
print('Número de parâmetros treináveis antes do congelamento =', len(rna.
 ↪trainable_weights))

# Congelamento dos parâmetros da rna_base
# Inclua seu código aqui
#
rna_base.trainable = False

# Número de parâmetros a serem treinados após o congelamento
print('Número de parâmetros treináveis após o congelamento =', len(rna.
 ↪trainable_weights))
```

Número de parâmetros treináveis antes do congelamento = 28
Número de parâmetros treináveis após o congelamento = 2

Saída esperada:

Número de parâmetros treináveis antes do congelamento = 28
Número de parâmetros treináveis após o congelamento = 2

5.3 Exercício #12: Treinamento da parte de classificação da RNA completa

Agora você vai treinar a parte de classifcação da sua RNA usando o método de otimização Adams. Assim, na célula abaixo, compile e treine a sua RNA usando os seguinte hiperparâmetros:

- método Adam;
- taxa de aprendizagem = 0.0005;
- número de épocas = 50;
- verbose = 1.

Observação: esse treinamento deve levar vários minutos.

[ ]:
```
# PARA VOCÊ FAZER: compilação e treinamento da RNA usando o método Adam

# Configuração do otimizador
# Inclua seu código aqui
#
```

```
opt2 = Adam(0.0005)
rna.compile(opt2, 'categorical_crossentropy', 'accuracy')
# Treinamento da RNA
# Inclua seu código aqui
#
history = rna.fit(x_train, y_train_hot, batch_size=32, epochs=50,␣
 ↪validation_data=(x_val, y_val_hot))
```

```
Epoch 1/50
938/938 [==============================] - 10s 10ms/step - loss: 1.7804 -
accuracy: 0.3976 - val_loss: 1.5685 - val_accuracy: 0.4667
Epoch 2/50
938/938 [==============================] - 9s 10ms/step - loss: 1.4707 -
accuracy: 0.5080 - val_loss: 1.4503 - val_accuracy: 0.5058
Epoch 3/50
938/938 [==============================] - 9s 10ms/step - loss: 1.3830 -
accuracy: 0.5327 - val_loss: 1.3882 - val_accuracy: 0.5235
Epoch 4/50
938/938 [==============================] - 9s 10ms/step - loss: 1.3331 -
accuracy: 0.5483 - val_loss: 1.3555 - val_accuracy: 0.5373
Epoch 5/50
938/938 [==============================] - 9s 10ms/step - loss: 1.2991 -
accuracy: 0.5593 - val_loss: 1.3272 - val_accuracy: 0.5462
Epoch 6/50
938/938 [==============================] - 9s 10ms/step - loss: 1.2738 -
accuracy: 0.5687 - val_loss: 1.3096 - val_accuracy: 0.5540
Epoch 7/50
938/938 [==============================] - 9s 10ms/step - loss: 1.2534 -
accuracy: 0.5738 - val_loss: 1.2985 - val_accuracy: 0.5557
Epoch 8/50
938/938 [==============================] - 9s 10ms/step - loss: 1.2371 -
accuracy: 0.5785 - val_loss: 1.2813 - val_accuracy: 0.5620
Epoch 9/50
938/938 [==============================] - 9s 10ms/step - loss: 1.2230 -
accuracy: 0.5824 - val_loss: 1.2729 - val_accuracy: 0.5688
Epoch 10/50
938/938 [==============================] - 9s 10ms/step - loss: 1.2114 -
accuracy: 0.5861 - val_loss: 1.2632 - val_accuracy: 0.5660
Epoch 11/50
938/938 [==============================] - 9s 10ms/step - loss: 1.2004 -
accuracy: 0.5907 - val_loss: 1.2548 - val_accuracy: 0.5683
Epoch 12/50
938/938 [==============================] - 9s 10ms/step - loss: 1.1916 -
accuracy: 0.5930 - val_loss: 1.2472 - val_accuracy: 0.5702
Epoch 13/50
938/938 [==============================] - 9s 10ms/step - loss: 1.1836 -
accuracy: 0.5958 - val_loss: 1.2463 - val_accuracy: 0.5750
```

```
Epoch 14/50
938/938 [==============================] - 9s 10ms/step - loss: 1.1764 -
accuracy: 0.5970 - val_loss: 1.2409 - val_accuracy: 0.5768
Epoch 15/50
938/938 [==============================] - 9s 10ms/step - loss: 1.1696 -
accuracy: 0.5995 - val_loss: 1.2397 - val_accuracy: 0.5690
Epoch 16/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1640 -
accuracy: 0.6012 - val_loss: 1.2338 - val_accuracy: 0.5768
Epoch 17/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1585 -
accuracy: 0.6019 - val_loss: 1.2308 - val_accuracy: 0.5780
Epoch 18/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1535 -
accuracy: 0.6046 - val_loss: 1.2264 - val_accuracy: 0.5793
Epoch 19/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1492 -
accuracy: 0.6060 - val_loss: 1.2276 - val_accuracy: 0.5750
Epoch 20/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1455 -
accuracy: 0.6083 - val_loss: 1.2208 - val_accuracy: 0.5830
Epoch 21/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1410 -
accuracy: 0.6087 - val_loss: 1.2195 - val_accuracy: 0.5773
Epoch 22/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1376 -
accuracy: 0.6097 - val_loss: 1.2169 - val_accuracy: 0.5803
Epoch 23/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1339 -
accuracy: 0.6115 - val_loss: 1.2162 - val_accuracy: 0.5807
Epoch 24/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1310 -
accuracy: 0.6109 - val_loss: 1.2134 - val_accuracy: 0.5827
Epoch 25/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1280 -
accuracy: 0.6134 - val_loss: 1.2150 - val_accuracy: 0.5807
Epoch 26/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1246 -
accuracy: 0.6132 - val_loss: 1.2177 - val_accuracy: 0.5780
Epoch 27/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1222 -
accuracy: 0.6160 - val_loss: 1.2075 - val_accuracy: 0.5827
Epoch 28/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1199 -
accuracy: 0.6147 - val_loss: 1.2094 - val_accuracy: 0.5872
Epoch 29/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1167 -
accuracy: 0.6166 - val_loss: 1.2072 - val_accuracy: 0.5868
```

```
Epoch 30/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1149 -
accuracy: 0.6170 - val_loss: 1.2105 - val_accuracy: 0.5810
Epoch 31/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1124 -
accuracy: 0.6186 - val_loss: 1.2076 - val_accuracy: 0.5817
Epoch 32/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1111 -
accuracy: 0.6187 - val_loss: 1.2046 - val_accuracy: 0.5878
Epoch 33/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1088 -
accuracy: 0.6195 - val_loss: 1.2034 - val_accuracy: 0.5893
Epoch 34/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1076 -
accuracy: 0.6190 - val_loss: 1.2062 - val_accuracy: 0.5847
Epoch 35/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1057 -
accuracy: 0.6213 - val_loss: 1.2032 - val_accuracy: 0.5838
Epoch 36/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1038 -
accuracy: 0.6192 - val_loss: 1.2054 - val_accuracy: 0.5875
Epoch 37/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1021 -
accuracy: 0.6216 - val_loss: 1.2047 - val_accuracy: 0.5860
Epoch 38/50
938/938 [==============================] - 10s 10ms/step - loss: 1.1006 -
accuracy: 0.6226 - val_loss: 1.2046 - val_accuracy: 0.5842
Epoch 39/50
938/938 [==============================] - 10s 10ms/step - loss: 1.0990 -
accuracy: 0.6219 - val_loss: 1.2033 - val_accuracy: 0.5822
Epoch 40/50
938/938 [==============================] - 10s 10ms/step - loss: 1.0979 -
accuracy: 0.6238 - val_loss: 1.2058 - val_accuracy: 0.5837
Epoch 41/50
938/938 [==============================] - 10s 10ms/step - loss: 1.0966 -
accuracy: 0.6237 - val_loss: 1.2045 - val_accuracy: 0.5862
Epoch 42/50
938/938 [==============================] - 10s 10ms/step - loss: 1.0948 -
accuracy: 0.6231 - val_loss: 1.2014 - val_accuracy: 0.5898
Epoch 43/50
938/938 [==============================] - 10s 10ms/step - loss: 1.0940 -
accuracy: 0.6249 - val_loss: 1.2037 - val_accuracy: 0.5857
Epoch 44/50
938/938 [==============================] - 10s 10ms/step - loss: 1.0926 -
accuracy: 0.6245 - val_loss: 1.2025 - val_accuracy: 0.5868
Epoch 45/50
938/938 [==============================] - 10s 10ms/step - loss: 1.0915 -
accuracy: 0.6235 - val_loss: 1.2010 - val_accuracy: 0.5883
```

```
Epoch 46/50
938/938 [==============================] - 10s 10ms/step - loss: 1.0906 -
accuracy: 0.6257 - val_loss: 1.2030 - val_accuracy: 0.5883
Epoch 47/50
938/938 [==============================] - 10s 10ms/step - loss: 1.0893 -
accuracy: 0.6250 - val_loss: 1.2027 - val_accuracy: 0.5853
Epoch 48/50
938/938 [==============================] - 10s 10ms/step - loss: 1.0881 -
accuracy: 0.6257 - val_loss: 1.1994 - val_accuracy: 0.5897
Epoch 49/50
938/938 [==============================] - 10s 10ms/step - loss: 1.0871 -
accuracy: 0.6256 - val_loss: 1.1992 - val_accuracy: 0.5922
Epoch 50/50
938/938 [==============================] - 10s 10ms/step - loss: 1.0859 -
accuracy: 0.6283 - val_loss: 1.2045 - val_accuracy: 0.5865
```

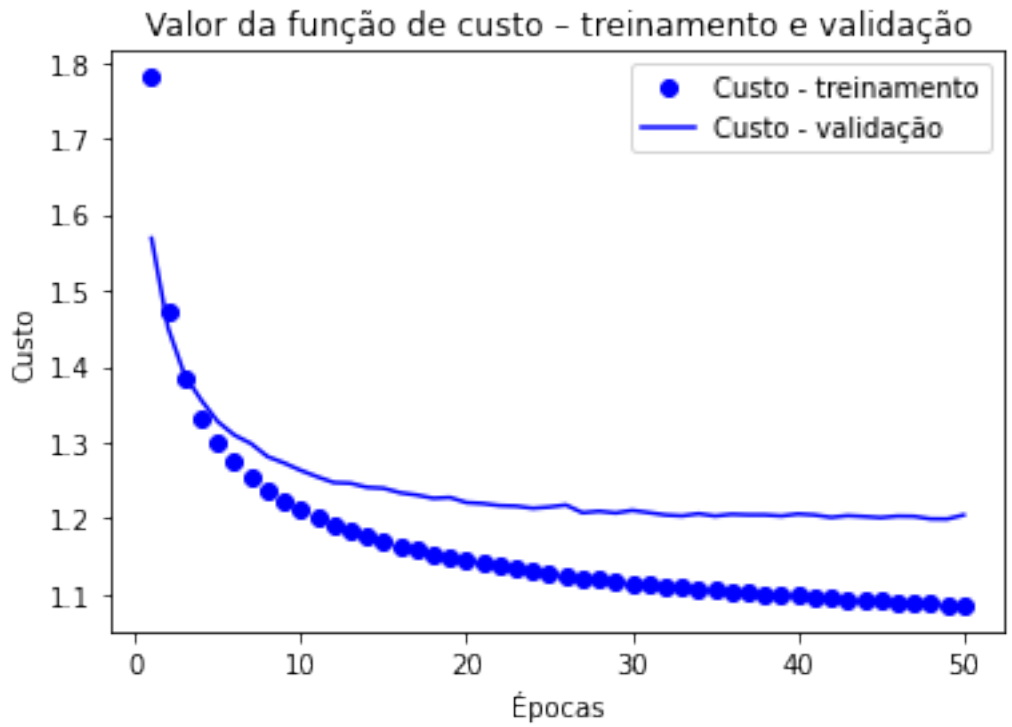
Saída esperada:

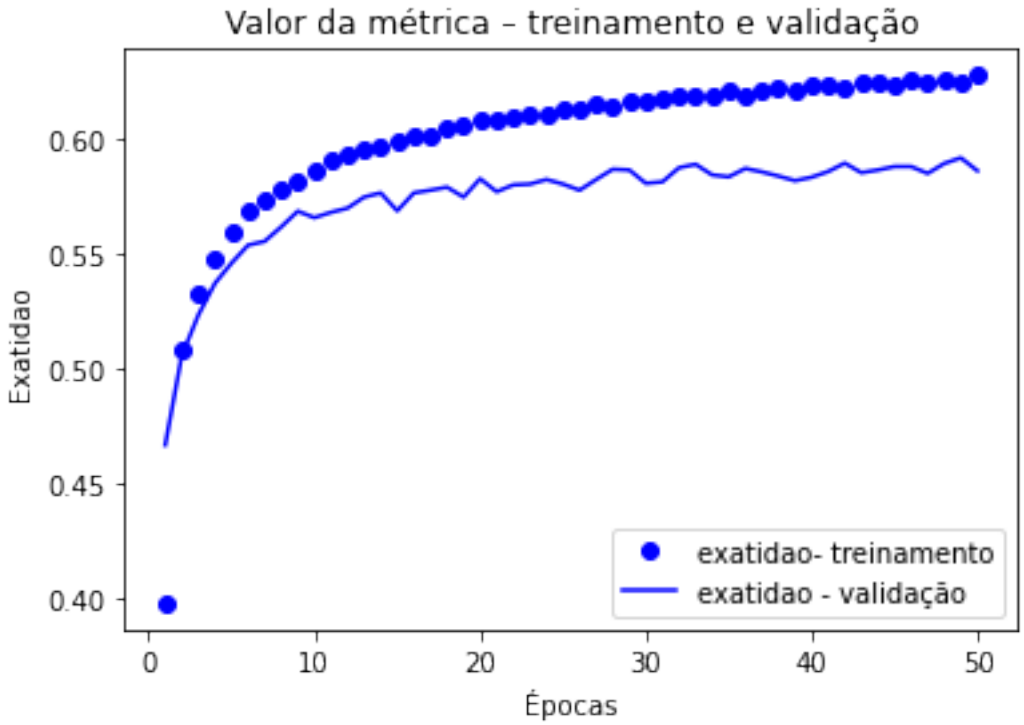
```
Epoch 1/50
938/938 [==============================] - 10s 11ms/step - loss: 1.7721 - accuracy: 0.4693 - val
.
.
.
Epoch 50/50
938/938 [==============================] - 10s 10ms/step - loss: 1.3133 - accuracy: 0.5791 - val
```

5.4 Visualização dos resultados e cálculo do custo e da métrica

Execute a célula abaixo para fazer os gráficos da função de custo e da métrica durante o treinamento e calcular os valores do custo e da exatidão para os dados de treinamento, validação e teste.

[ ]:
```python
# PARA VOCÊ FAZER: visualização e avaliação dos resultados

# Gráfico do processo de treinamento (use a função plot_results_train)
# Inclua seu código aqui
#
plot_results_train(history)
#Calculo do custo e exatidão para os dados de treinamento, validação e teste
# Inclua seu código aqui
#
train_eval = rna.evaluate(x_train, y_train_hot, batch_size=32)
val_eval = rna.evaluate(x_val, y_val_hot, batch_size=32)
test_eval = rna.evaluate(x_test, y_test_hot, batch_size=32)
```


Valor da função de custo – treinamento e validação
Custo - treinamento
Custo - validação
Custo
Épocas
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
0
10
20
30
40
50


Valor da métrica – treinamento e validação
exatidao- treinamento
exatidao - validação
Exatidao
Épocas
0.60
0.55
0.50
0.45
0.40
0
10
20
30
40
50

```
938/938 [==============================] - 8s 8ms/step - loss: 1.0832 -
accuracy: 0.6260
188/188 [==============================] - 2s 8ms/step - loss: 1.2045 -
accuracy: 0.5865
188/188 [==============================] - 2s 8ms/step - loss: 1.2076 -
accuracy: 0.5823
```

Saída esperada:

```
938/938 [==============================] - 8s 8ms/step - loss: 1.2792 - accuracy: 0.5882
188/188 [==============================] - 2s 8ms/step - loss: 1.3486 - accuracy: 0.5675
188/188 [==============================] - 2s 8ms/step - loss: 1.3712 - accuracy: 0.5590
```

5.5 Análise dos resultados

Pelos gráficos e valores da função de custo e da métrica para os dados de treinamento, validação e teste você deve observar que o treinamento e os resultados obtidos com essa RNA é praticamente igual ao obtido pelo método de extração de características. Isso era de se esperar porque não alteramos a parte convolucional da rna_base, assim, as características extraídas das imagens são exatamente iguais nos dois casos e como a parte densa de classificação dos dois métodos tem as mesmas caraterísticas, o resultado não pode ser diferente nos dois casos.

Para que seja possível obter resultados melhores com essa nova RNA temos que fazer a sua sintonia fina, ou seja, temos que retreinar a parte final da rna_base para ela se ajustar melhor aos novos dados.

5.6 Exercício #13: Descongelamento dos parâmetros do último bloco da `rna_base`

Na célula abaixo crie um código que descongela as camadas convolucionais do último bloco da rna_base (block_5). Note que para isso você precisa saber os ‘‘nomes’’ das várias camadas da RNA. Verifique o nome da primmeira camada do último bloco da rna_base para incluir como um sinal para iniciar o descongelamento das camadas (ver notas de aula).

[ ]:
```python
# PARA VOCÊ FAZER: descongelamento das camadas convolucionais do block5.

# Descongela todas as camadas da rna_base
# Inclua seu código aqui
#
rna_base.trainable = True
set_trainable = False

# Percorre camadas da rna_base procurando pelo 5º bloco
# Inclua seu código aqui
#
for layer in rna_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
```

```
        layer.trainable = True
    else:
        layer.trainable = False

# Número de parâmetros a serem treinados após o descongelamento parcial
print('Número de parâmetros treináveis após o descongelamento =', len(rna.
 ↪trainable_weights))
```

Número de parâmetros treináveis após o descongelamento = 8

Saída esperada:

Número de parâmetros treináveis após o descongelamento = 8

Observação: No treinamento da sua RNA 10 tensores de parâmetros serão treinados. Sendo que 6 pretencem às 3 últimas camadas convolucionais da rna_base (3 tensores de pesos dos filtros e 3 tensores de viéses) e 2 pertencem à parte de classificação da rede.

5.7 Exercício #14: Sintonia fina da RNA completa

Agora a RNA está pronta para ser retreinada e sintonizada para o novo problema. Lembre-se de que a sintonia fina deve ser realizada com uma taxa de aprendizado muito pequena, porque se deseja limitar o valor das modificações das camadas convolucionais que estão sendo ajustadas. Atualizações muito grande dos parâmetros podem destruir completamente o treinamento original.

Agora você vai compilar e treinar a sua RNA usando os seguinte hiperparâmetros:

- método Adam;
- taxa de aprendizagem = 1e-05;
- número de épocas = 30;
- verbose = 1.

Observação: Esse treinamento deve levar vários minutos.

[ ]:
```
# PARA VOCÊ FAZER: compilação e treinamento da RNA usando o método Adam

# Configuração do otimizador
# Inclua seu código aqui
opt3 = Adam(1e-5)
rna.compile(opt3, loss='categorical_crossentropy', metrics='accuracy')
# Treinamento da RNA
# Inclua seu código aqui
#
history = rna.fit(x_train, y_train_hot, batch_size=32, epochs=30,␣
 ↪validation_data=(x_val, y_val_hot))
```

```
Epoch 1/30
938/938 [==============================] - 16s 17ms/step - loss: 1.0114 -
accuracy: 0.6482 - val_loss: 1.0187 - val_accuracy: 0.6520
```

```
Epoch 2/30
938/938 [==============================] - 15s 16ms/step - loss: 0.7986 -
accuracy: 0.7233 - val_loss: 0.9599 - val_accuracy: 0.6735
Epoch 3/30
938/938 [==============================] - 15s 16ms/step - loss: 0.6721 -
accuracy: 0.7666 - val_loss: 0.9419 - val_accuracy: 0.6768
Epoch 4/30
938/938 [==============================] - 15s 16ms/step - loss: 0.5716 -
accuracy: 0.8029 - val_loss: 0.9137 - val_accuracy: 0.6957
Epoch 5/30
938/938 [==============================] - 15s 16ms/step - loss: 0.4852 -
accuracy: 0.8364 - val_loss: 0.9027 - val_accuracy: 0.6993
Epoch 6/30
938/938 [==============================] - 16s 17ms/step - loss: 0.4163 -
accuracy: 0.8642 - val_loss: 0.9104 - val_accuracy: 0.7000
Epoch 7/30
938/938 [==============================] - 16s 17ms/step - loss: 0.3521 -
accuracy: 0.8880 - val_loss: 0.9065 - val_accuracy: 0.6995
Epoch 8/30
938/938 [==============================] - 16s 17ms/step - loss: 0.2976 -
accuracy: 0.9107 - val_loss: 0.9113 - val_accuracy: 0.7075
Epoch 9/30
938/938 [==============================] - 16s 17ms/step - loss: 0.2522 -
accuracy: 0.9277 - val_loss: 0.9166 - val_accuracy: 0.7138
Epoch 10/30
938/938 [==============================] - 16s 17ms/step - loss: 0.2089 -
accuracy: 0.9441 - val_loss: 0.9467 - val_accuracy: 0.7135
Epoch 11/30
938/938 [==============================] - 16s 17ms/step - loss: 0.1742 -
accuracy: 0.9571 - val_loss: 0.9542 - val_accuracy: 0.7120
Epoch 12/30
938/938 [==============================] - 16s 17ms/step - loss: 0.1434 -
accuracy: 0.9681 - val_loss: 0.9680 - val_accuracy: 0.7160
Epoch 13/30
938/938 [==============================] - 16s 17ms/step - loss: 0.1192 -
accuracy: 0.9747 - val_loss: 1.0260 - val_accuracy: 0.7125
Epoch 14/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0945 -
accuracy: 0.9833 - val_loss: 1.0216 - val_accuracy: 0.7170
Epoch 15/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0797 -
accuracy: 0.9876 - val_loss: 1.0335 - val_accuracy: 0.7195
Epoch 16/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0624 -
accuracy: 0.9926 - val_loss: 1.0688 - val_accuracy: 0.7195
Epoch 17/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0499 -
accuracy: 0.9948 - val_loss: 1.1006 - val_accuracy: 0.7203
```

```
Epoch 18/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0415 -
accuracy: 0.9966 - val_loss: 1.1288 - val_accuracy: 0.7180
Epoch 19/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0306 -
accuracy: 0.9983 - val_loss: 1.1701 - val_accuracy: 0.7245
Epoch 20/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0333 -
accuracy: 0.9967 - val_loss: 1.1853 - val_accuracy: 0.7185
Epoch 21/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0225 -
accuracy: 0.9983 - val_loss: 1.2404 - val_accuracy: 0.7135
Epoch 22/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0152 -
accuracy: 0.9997 - val_loss: 1.2192 - val_accuracy: 0.7242
Epoch 23/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0126 -
accuracy: 0.9997 - val_loss: 1.3687 - val_accuracy: 0.7093
Epoch 24/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0305 -
accuracy: 0.9944 - val_loss: 1.2582 - val_accuracy: 0.7235
Epoch 25/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0067 -
accuracy: 1.0000 - val_loss: 1.2985 - val_accuracy: 0.7283
Epoch 26/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0201 -
accuracy: 0.9950 - val_loss: 1.4014 - val_accuracy: 0.7038
Epoch 27/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0131 -
accuracy: 0.9987 - val_loss: 1.3316 - val_accuracy: 0.7252
Epoch 28/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0042 -
accuracy: 1.0000 - val_loss: 1.3919 - val_accuracy: 0.7277
Epoch 29/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0032 -
accuracy: 1.0000 - val_loss: 1.4024 - val_accuracy: 0.7265
Epoch 30/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0033 -
accuracy: 0.9999 - val_loss: 1.6000 - val_accuracy: 0.7002
```

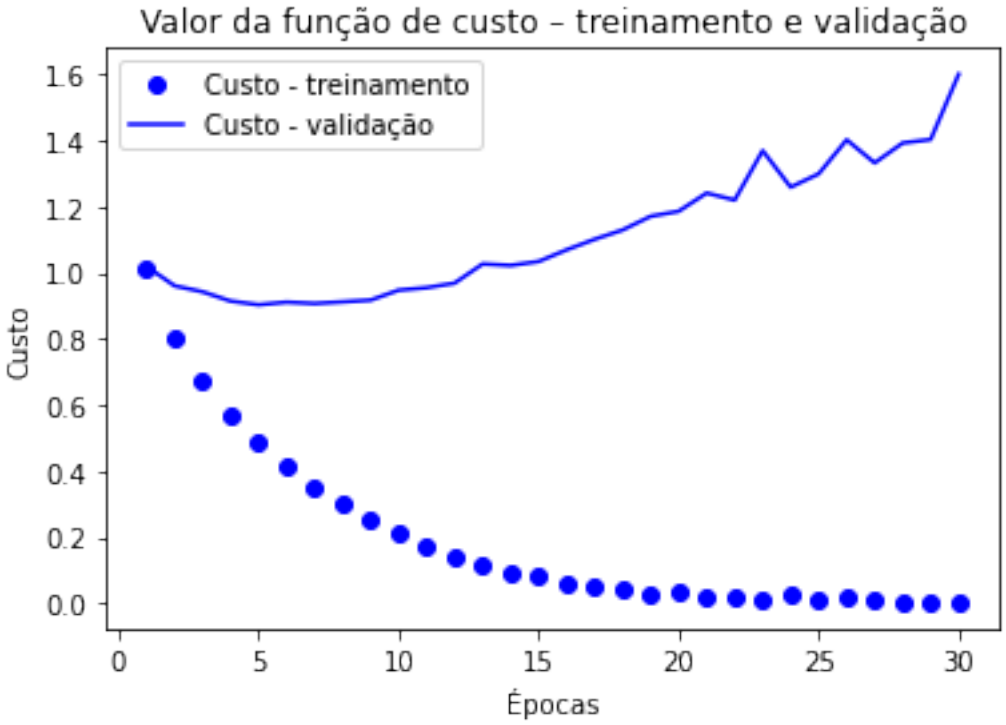
Saída esperada:

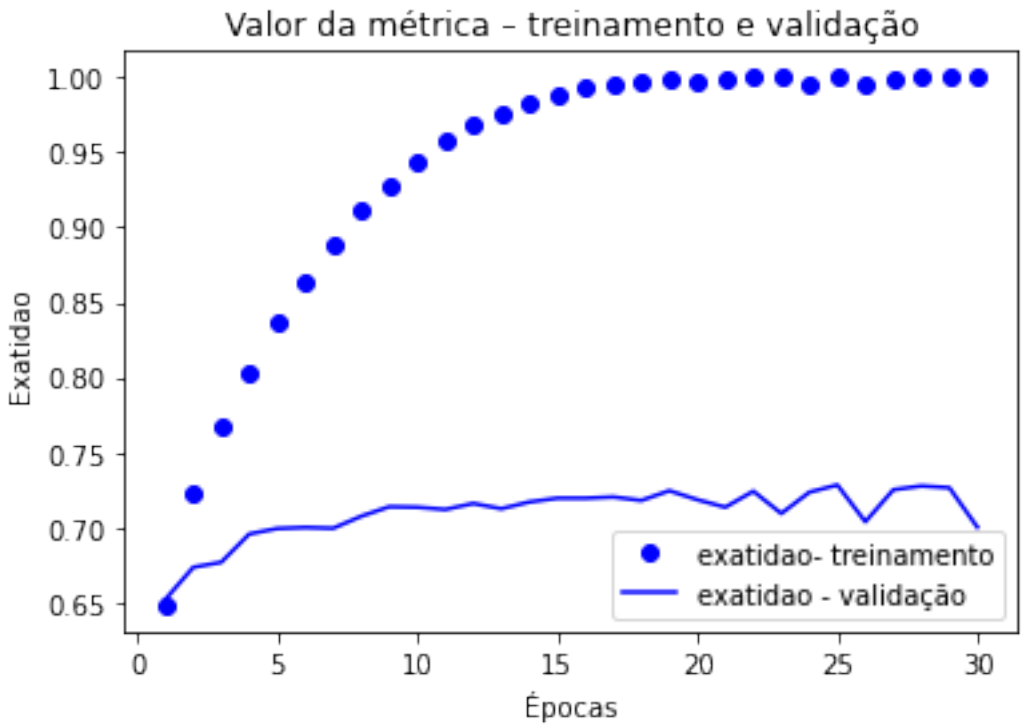
```
Epoch 1/30
938/938 [==============================] - 16s 17ms/step - loss: 1.1222 - accuracy: 0.6430 - val
.
.
.
Epoch 30/30
938/938 [==============================] - 16s 17ms/step - loss: 0.0565 - accuracy: 0.9861 - val
```

5.8 Visualização dos resultados

Execute a célula a seguir para fazer os gráficos da função de custo e da métrica para os dados de treinamento e validação.

[ ]:
```
# Gráfico do processo de treinamento
plot_results_train(history)
```





Execute a célula abaixo para calcular os valores do custo e da exatidão para os dados de treinamento, validação e teste.

[ ]:
```
#Calculo do custo e exatidão para os dados de treinamento, validação e teste
custo_e_metricas_train = rna.evaluate(x_train, y_train_hot)
custo_e_metricas_val = rna.evaluate(x_val, y_val_hot)
custo_e_metricas_test = rna.evaluate(x_test, y_test_hot)
```

```
938/938 [==============================] - 8s 8ms/step - loss: 0.0399 -
accuracy: 0.9889
188/188 [==============================] - 2s 8ms/step - loss: 1.6000 -
accuracy: 0.7002
188/188 [==============================] - 2s 8ms/step - loss: 1.5640 -
accuracy: 0.7163
```

Saída esperada:

```
938/938 [==============================] - 8s 8ms/step - loss: 0.0308 - accuracy: 0.9971
188/188 [==============================] - 2s 8ms/step - loss: 1.5541 - accuracy: 0.7290
188/188 [==============================] - 2s 8ms/step - loss: 1.5556 - accuracy: 0.7380
```

5.9 Análise dos resultados

- Pode-se observar que o treinamento não inicia do zero pois as camadas densas da RNA já foram pré-treinadas e agora somente estão sendo ajustados os

parâmetros das últimas camadas convolucionais da rna_base e retreinando as camadas densas.

- Uma exatidão de cerca de 73% para os dados de validação e de teste represetam um resultado não muito bom.
- Observa-se que os resultados da exatidão para os dados de teste melhoraram bastante, mas não o suficiente em razão do problema de overfitting.

Observa-se que para eliminar esse problema de overfitting teríamos que introduzir regularização L2 e/ou dropout nas camadas convolucionais que retreinamos. Se isso fosse feito certamente conseguiríamos resultados para o conjunto de teste da ordem de 99%, como os obtidos para o conjunto de treinamento.

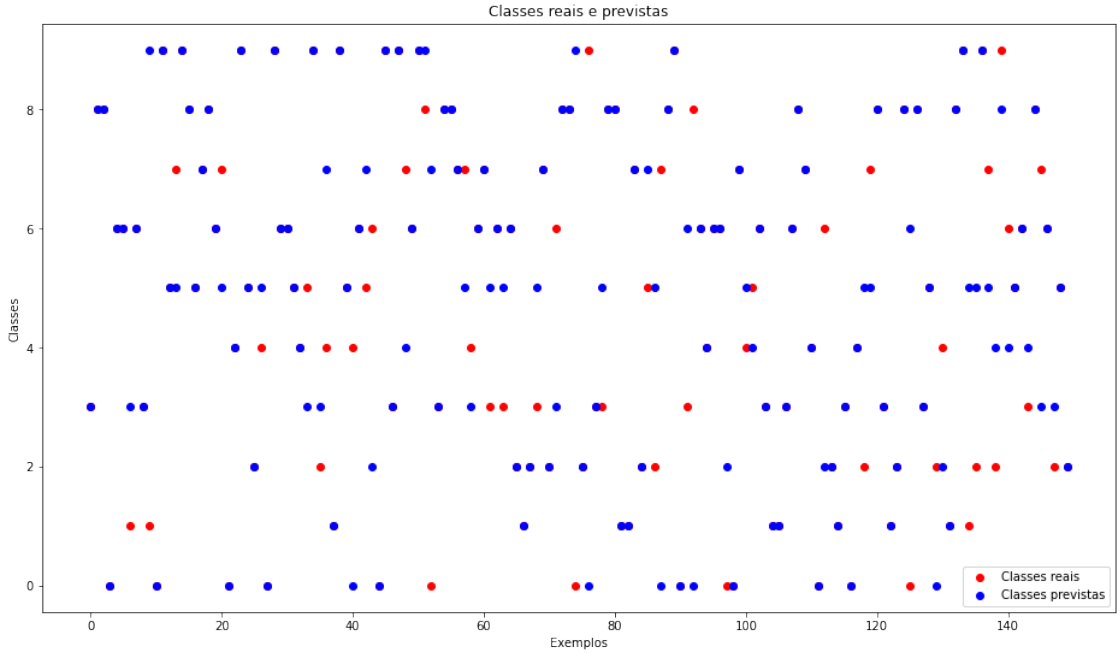
5.10 Exercício #15: Verificação dos resultados

Na célula abaixo calcule a previsões da sua RNA para as imagens dos dados de teste e depois verifique se algumas dessas previsões estão corretas fazendo o gráfico das classes previstas e reais dos primeiros 150 exemplos de teste.

Note que a previsão da RNA é um vetor de 10 elementos com as probabilidades da imagem mostrar os seis sinais. Para determinar a classe prevista deve-se transformar esse vetor em um número inteiro de 0 a 9, que representa o sinal sendo mostrado. Para fazer essa transformação use a função numpy argmax(y_prev, axis=?), onde y_prev é o tensor com as saídas previstas pela RNA. Em qual eixo você deve calcular o índice da maior probabilidade?

[ ]:
```
# PARA VOCÊ FAZER: cálculo das classes previstas pela RNA com dropout

# Calculo das previsões da RNA
# Inclua seu código aqui
#
y_pred = np.argmax(rna.predict(x_test), axis=-1)
# Cálculo das classes previstas
# Inclua seu código aqui
#

# Gráfico das classes reais e previstas
# Fazer o gráfico das classes reais e previstas dos 150 primeiros exemplos de␣
 ↪teste
# Inclua seu código aqui
#
fig = plt.figure(figsize=(16,9))
plt.scatter([i for i in range(150)], y_test[:150], label='Classes reais', c='r')
plt.scatter([i for i in range(150)], y_pred[:150], label='Classes previstas',␣
 ↪c='b')
plt.legend()
plt.title("Classes reais e previstas")
plt.xlabel("Exemplos")
plt.ylabel("Classes")
```

[ ]: Text(0, 0.5, 'Classes')



Saída esperada:

Comentários:

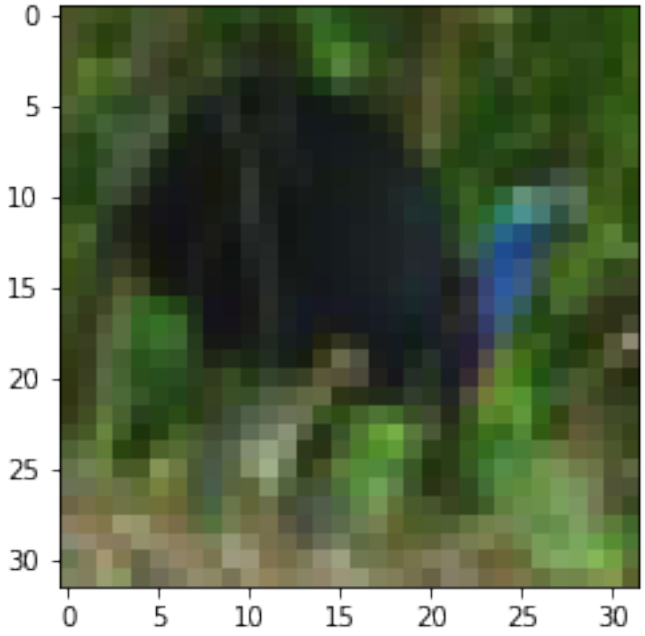
- Uma previsão errada de classe pode ser dectada pelos círculos vermelhos, pois quando a classe prevista é igual à classe real o círculo azul é colocado em cima do vermelho tapando-o.
- No gráfico devem aparecer mais círculos azuis do que vermelhos indicando que existem mais acertos do que erros na previsão das classes.

Visualização de previsões erradas:

Use o gráfico acima e escolha um exemplo onde a previsão da sua RNA está errada. Na célula abaixo troque a variável index e execute-a para verificar se o exemplo tem alguma particularidade que dificulta a sua RNA de prever corretamente a classe desse exemplo.

[ ]:
```
# Cálculo das classes previstas pela RNA com dropout

# Exemplo de uma imagem dos dados de teste
index = 775
plt.imshow(x_test_orig[index])
print ("classe prevista = " + str(np.squeeze(classe[index])))
print ("classe real = " + str(np.squeeze(y_test[index])))
```

```
classe prevista = 5
classe real = 2
```



6 Conclusões

Os resultados obtidos não foram muito bons, por causa de problemas de overfitting. Para melhorar esses resultados teríamos que resolver o problema de overfitting aplicando dropout e/ou regularização nas camadas convolucionais retreinadas.

Apesar dos resultados não muito satisfatórios, observa-se o seguinte:

- Transferência de aprendizado consiste de uma abordagem fácil e altamente eficiente para desenvolver uma nova aplicação.
- Transferência de aprendizado é muito eficiente, principalmente quando o banco de dados disponível é pequeno.
- Existem dois métodos para usar uma RNA pré-treinada. Qualquer um dos dois métodos é eficiente para obter sistemas com bom desempenho, mesmo quando o conjunto de dados é pequeno.
- A transferência de treinamento realizada com treinamento parcial com a técnica de sintonia fina é muito eficiente para obter sistemas com alto desempenho, mesmo quando o conjunto de dados é pequeno.