Trabalho_Gato_Nao_Gato

June 24, 2023

[ ]:
```
# **Gatos vs Não-Gatos **

### Alunos:
    Antonio Silverio Montagner (19203742)
    Davi Becker da Silva (18206220)

Neste trabalho, será abordado o reconhecimento de padrões por meio do␣
↪treinamento de uma rede neural. O objetivo é desenvolver um sistema capaz de␣
↪identificar e classificar imagens de gatinhos como "gatos" ou "não-gatos".

Para isso, utilizaremos um conjunto de dados contendo imagens RGB,␣
↪representadas por matrizes de 64x64x3 (4096 pixels coloridos). Essas imagens␣
↪serão usadas como parâmetros de entrada para a rede neural.

O objetivo da rede é atribuir um valor de classificação correto para cada␣
↪imagem, sendo que o valor 0 corresponderá a "não-gato" e o valor 1␣
↪corresponderá a "gato".

Para resolver esse problema, serão explorados três modelos diferentes: um␣
↪perceptron simples (regressão logística), uma rede neural de camada rasa e␣
↪uma rede neural convolucional.
Cada modelo terá seu próprio processo de treinamento e teste, visando encontrar␣
↪a melhor abordagem para a classificação precisa das imagens.
```

[ ]:
```
# **Regressão Logística**
Inicialmente, abordaremos o problema utilizando apenas um perceptron, por meio␣
↪de regressão logística.
```

[ ]:
```
## Importando as bibliotecas e lendo os dados
```

[7]:
```
# Importando bibliotecas
import numpy as np
import h5py
import matplotlib.pyplot as plt

# Carregando os dados
train_dataset = h5py.File('./train_catvnoncat.h5', "r")
```

```
train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # train set␣
 ↪features
train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # train set labels

test_dataset = h5py.File('./test_catvnoncat.h5', "r")
test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # test set features
test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # test set labels
```

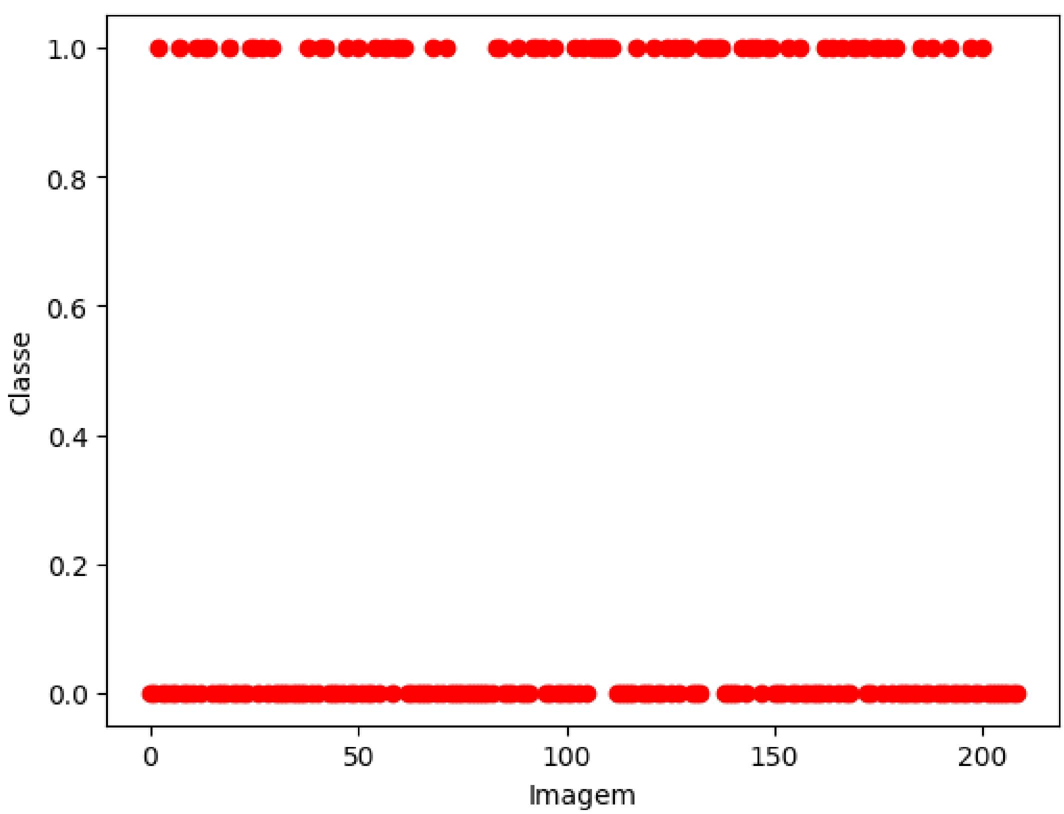
[ ]:
```
Para facilitar a visualização dos dados de treinamento, iremos representá-los␣
 ↪em um gráfico.
```

[8]:
```
n = len(train_set_x_orig) # número de imagens
print('Tamanho da base de dados de treinamento:', n)
print('train set shape:', train_set_x_orig.shape)

plt.scatter(range(n),train_set_y_orig,c='red')
plt.xlabel('Imagem')
plt.ylabel('Classe')
```

```
Tamanho da base de dados de treinamento: 209
train set shape: (209, 64, 64, 3)
```

[8]:
```
Text(0, 0.5, 'Classe')
```



[ ]:
```
Nosso conjunto de dados consiste em 209 imagens de tamanho 64x64x3,␣
 ↪classificadas como gatos (classe 1) ou não-gatos (classe 0).
```

[ ]:
```
## Manipulando os dados
Antes de prosseguirmos, faremos um achatamento (flatten) nos vetores de teste e␣
 ↪treinamento, além de normalizar os dados de entrada.
```

[9]:
```
from sklearn.preprocessing import MinMaxScaler

# Transformando os arrays em arrays de 209 x 12288 (209 x (64 . 64 . 3))
train_set_x = np.array([array.flatten() for array in train_set_x_orig])
test_set_x = np.array([array.flatten() for array in test_set_x_orig])

# Normalizando o array de treino
norm_train_set_x = MinMaxScaler()
norm_train_set_x = norm_train_set_x.fit_transform(train_set_x)
```

[3]:
```
## Modelando a regressão e realizando testes
Com as preparações feitas, iremos modelar a regressão logística e realizar␣
 ↪alguns testes.
```

[10]:
```
from sklearn.linear_model import LogisticRegression

# Treinando o modelo de regressão logística
clf = LogisticRegression(random_state=0, max_iter=10000).fit(norm_train_set_x,␣
 ↪train_set_y_orig)

print(f'Acurácia sobre o arquivo de treino = {clf.score(train_set_x,␣
 ↪train_set_y_orig) * 100}%')
print(f'Acurácia sobre o arquivo de testes = {clf.score(test_set_x,␣
 ↪test_set_y_orig) * 100}%')

print('\nResultados esperados do arquivo de teste:', *test_set_y_orig)
print('Resultados obtidos do arquivo de teste:  ', *clf.predict(test_set_x))
```

```
Acurácia sobre o arquivo de treino = 100.0%
Acurácia sobre o arquivo de testes = 72.0%

Resultados esperados do arquivo de teste: 1 1 1 1 1 0 1 1 1 1 1 1 1 0 0 1 0 1 1
1 1 0 0 1 1 1 1 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1 1 0 0 0 1 1 1 0
Resultados obtidos do arquivo de teste:   1 1 1 1 1 0 0 1 1 1 0 0 1 1 0 1 0 1 0
0 1 0 0 1 0 1 1 0 0 1 0 1 1 1 0 0 0 1 0 0 1 0 1 0 1 1 0 1 1 0
```

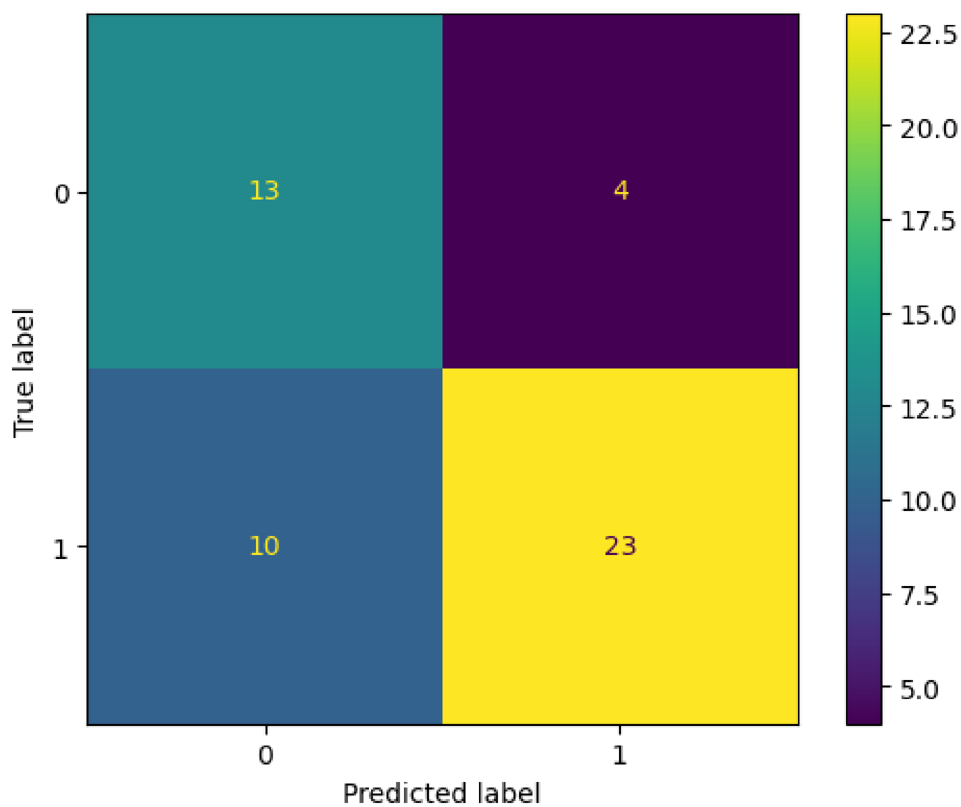
[ ]:
```
Verificamos que o modelo criado obteve resultados satisfatórios, sendo capaz de identificar com precisão se uma determinada imagem é de um gato ou não em 72% das vezes.
```

[ ]:
```
## Matriz de confusão
Com base nos resultados obtidos, construímos uma matriz de confusão para avaliar o desempenho do modelo em relação aos dados de teste.
```

[11]:
```python
from sklearn.metrics import ConfusionMatrixDisplay

ConfusionMatrixDisplay.from_estimator(clf, test_set_x, test_set_y_orig)
```

[11]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at
0x7f9b7ab84910>
```



[ ]:
```
# **Rede de Camada Rasa**
Visando obter um desempenho ainda melhor e resultados mais confiáveis, iremos implementar a classificação de imagens de gatinhos em um modelo mais robusto, com a expectativa de obter respostas mais precisas.
```

[ ]:
```
## Importando as bibliotecas Keras e Tensorflow
```

[14]:
```
from tensorflow import keras          # Importa Keras
from tensorflow.keras import layers  # Ferramentes do Keras mais usadas para␣
 ↪acesso mais rápido
from keras.layers import Dense
from keras.models import Sequential
from tensorflow.keras.utils import to_categorical
```

[ ]:
```
## Definindo as classes de classificação
Vamos utilizar duas classes:

*   0 - não-gatos
*   1 - gatos
```

[15]:
```
train_set_y = to_categorical(train_set_y_orig, 2)
test_set_y = to_categorical(test_set_y_orig, 2)
```

[ ]:
```
## Definindo um novo modelo
Para esta abordagem, vamos utilizar uma rede neural com apenas 3 camadas:
*   uma camada de entrada (*flatten*);
*   uma camada intermediária, composta por 1000 neurônios (com função de␣
 ↪ativação sigmóide)
*   uma camada de saída, composta por 2 neurônios (com função de ativação␣
 ↪*softmax*).


A rede utilizará o decaimento exponencial da taxa de aprendizado, além de pesos␣
 ↪e *bias* aleatórios.
```

[16]:
```
# Decaimento exponencial da taxa de aprendizado
def exp_decay(epoch):
   initial_lrate = 1.0
   k = 0.05
   lrate = initial_lrate * np.exp(-k*epoch)
   return lrate

lrate = keras.callbacks.LearningRateScheduler(exp_decay)
callback = keras.callbacks.EarlyStopping(monitor='val_loss', mode='min',␣
 ↪verbose=1, patience=50)

# Definindo a rede
modelo = keras.Sequential()
modelo.add(layers.Flatten())
modelo.add(layers.Dense(1000, kernel_initializer="random_uniform",␣
 ↪bias_initializer="random_uniform", activation="sigmoid"))
modelo.add(layers.Dense(2, kernel_initializer="random_uniform",␣
 ↪bias_initializer="random_uniform", activation="softmax"))
```

```
opt = keras.optimizers.SGD()
modelo.compile(optimizer=opt, loss="categorical_crossentropy",␣
 ↪metrics=["accuracy"])

input_shape = train_set_x.shape
modelo.build(input_shape)

modelo.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (209, 12288)              0

 dense (Dense)               (209, 1000)               12289000

 dense_1 (Dense)             (209, 2)                  2002

=================================================================
Total params: 12,291,002
Trainable params: 12,291,002
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
## Criando um conjunto de validação
A fim de acompanhar o progresso da rede, faremos uso de um conjunto de␣
 ↪validação composto por 63 fotos.
```

[17]:
```
from sklearn.model_selection import train_test_split

Xtr,Xval,ytr,yval = train_test_split(norm_train_set_x,train_set_y,test_size = 0.
 ↪3)
num_train = np.size(Xtr,0)
print(num_train)
```

```
146
```

[ ]:
```
## Treinando o modelo
Agora, iremos treinar o modelo que foi criado, utilizando um tamanho de lote␣
 ↪(batch size) de 30 e realizando 2000 épocas.
```

[19]:
```
results = modelo.fit(Xtr, ytr, validation_data = (Xval, yval), batch_size = 30,␣
 ↪epochs=2000, callbacks=[callback, lrate], verbose=2)
```

```
Epoch 1/2000
```

```
5/5 - 1s - loss: 5.8668 - accuracy: 0.6575 - val_loss: 14.1048 - val_accuracy:
0.6825 - lr: 1.0000 - 596ms/epoch - 119ms/step
Epoch 2/2000
5/5 - 0s - loss: 10.5108 - accuracy: 0.5548 - val_loss: 18.7739 - val_accuracy:
0.3175 - lr: 0.9512 - 465ms/epoch - 93ms/step
Epoch 3/2000
5/5 - 0s - loss: 9.1147 - accuracy: 0.5411 - val_loss: 13.7885 - val_accuracy:
0.3175 - lr: 0.9048 - 457ms/epoch - 91ms/step
Epoch 4/2000
5/5 - 0s - loss: 9.0811 - accuracy: 0.5342 - val_loss: 2.7697 - val_accuracy:
0.3016 - lr: 0.8607 - 466ms/epoch - 93ms/step
Epoch 5/2000
5/5 - 0s - loss: 7.7188 - accuracy: 0.5342 - val_loss: 2.9836 - val_accuracy:
0.6825 - lr: 0.8187 - 469ms/epoch - 94ms/step
Epoch 6/2000
5/5 - 0s - loss: 7.3719 - accuracy: 0.5068 - val_loss: 7.8240 - val_accuracy:
0.6825 - lr: 0.7788 - 461ms/epoch - 92ms/step
Epoch 7/2000
5/5 - 1s - loss: 4.5361 - accuracy: 0.5959 - val_loss: 8.5230 - val_accuracy:
0.6825 - lr: 0.7408 - 523ms/epoch - 105ms/step
Epoch 8/2000
5/5 - 0s - loss: 6.4005 - accuracy: 0.5479 - val_loss: 8.2829 - val_accuracy:
0.3175 - lr: 0.7047 - 498ms/epoch - 100ms/step
Epoch 9/2000
5/5 - 0s - loss: 5.9708 - accuracy: 0.5274 - val_loss: 2.9454 - val_accuracy:
0.3175 - lr: 0.6703 - 474ms/epoch - 95ms/step
Epoch 10/2000
5/5 - 0s - loss: 5.4576 - accuracy: 0.5068 - val_loss: 1.4133 - val_accuracy:
0.6825 - lr: 0.6376 - 492ms/epoch - 98ms/step
Epoch 11/2000
5/5 - 1s - loss: 4.4021 - accuracy: 0.5068 - val_loss: 7.1116 - val_accuracy:
0.6825 - lr: 0.6065 - 514ms/epoch - 103ms/step
Epoch 12/2000
5/5 - 1s - loss: 5.4454 - accuracy: 0.5000 - val_loss: 7.4105 - val_accuracy:
0.6825 - lr: 0.5769 - 518ms/epoch - 104ms/step
Epoch 13/2000
5/5 - 0s - loss: 4.3719 - accuracy: 0.6027 - val_loss: 3.4601 - val_accuracy:
0.6825 - lr: 0.5488 - 462ms/epoch - 92ms/step
Epoch 14/2000
5/5 - 0s - loss: 2.5947 - accuracy: 0.5822 - val_loss: 5.2047 - val_accuracy:
0.3175 - lr: 0.5220 - 482ms/epoch - 96ms/step
Epoch 15/2000
5/5 - 0s - loss: 3.8297 - accuracy: 0.5616 - val_loss: 0.8873 - val_accuracy:
0.4444 - lr: 0.4966 - 490ms/epoch - 98ms/step
Epoch 16/2000
5/5 - 0s - loss: 2.6242 - accuracy: 0.5890 - val_loss: 0.8713 - val_accuracy:
0.3968 - lr: 0.4724 - 455ms/epoch - 91ms/step
Epoch 17/2000
```

```
5/5 - 0s - loss: 2.6626 - accuracy: 0.5342 - val_loss: 6.0314 - val_accuracy:
0.3175 - lr: 0.4493 - 487ms/epoch - 97ms/step
Epoch 18/2000
5/5 - 1s - loss: 3.3653 - accuracy: 0.5479 - val_loss: 2.6588 - val_accuracy:
0.3016 - lr: 0.4274 - 507ms/epoch - 101ms/step
Epoch 19/2000
5/5 - 0s - loss: 2.4287 - accuracy: 0.5753 - val_loss: 1.0470 - val_accuracy:
0.6825 - lr: 0.4066 - 450ms/epoch - 90ms/step
Epoch 20/2000
5/5 - 1s - loss: 2.1697 - accuracy: 0.5685 - val_loss: 4.3449 - val_accuracy:
0.6825 - lr: 0.3867 - 502ms/epoch - 100ms/step
Epoch 21/2000
5/5 - 0s - loss: 2.0564 - accuracy: 0.6370 - val_loss: 2.6625 - val_accuracy:
0.6825 - lr: 0.3679 - 496ms/epoch - 99ms/step
Epoch 22/2000
5/5 - 1s - loss: 2.3376 - accuracy: 0.5753 - val_loss: 3.9462 - val_accuracy:
0.3016 - lr: 0.3499 - 514ms/epoch - 103ms/step
Epoch 23/2000
5/5 - 0s - loss: 3.1145 - accuracy: 0.5000 - val_loss: 2.9101 - val_accuracy:
0.3016 - lr: 0.3329 - 457ms/epoch - 91ms/step
Epoch 24/2000
5/5 - 0s - loss: 1.6561 - accuracy: 0.5685 - val_loss: 3.1131 - val_accuracy:
0.6825 - lr: 0.3166 - 457ms/epoch - 91ms/step
Epoch 25/2000
5/5 - 0s - loss: 1.7793 - accuracy: 0.5685 - val_loss: 3.6287 - val_accuracy:
0.6825 - lr: 0.3012 - 422ms/epoch - 84ms/step
Epoch 26/2000
5/5 - 1s - loss: 2.0079 - accuracy: 0.6027 - val_loss: 1.0303 - val_accuracy:
0.6825 - lr: 0.2865 - 518ms/epoch - 104ms/step
Epoch 27/2000
5/5 - 0s - loss: 1.5671 - accuracy: 0.6027 - val_loss: 0.8734 - val_accuracy:
0.6825 - lr: 0.2725 - 476ms/epoch - 95ms/step
Epoch 28/2000
5/5 - 1s - loss: 0.5298 - accuracy: 0.7123 - val_loss: 2.9829 - val_accuracy:
0.3651 - lr: 0.2592 - 508ms/epoch - 102ms/step
Epoch 29/2000
5/5 - 0s - loss: 1.7746 - accuracy: 0.5411 - val_loss: 1.9676 - val_accuracy:
0.3810 - lr: 0.2466 - 498ms/epoch - 100ms/step
Epoch 30/2000
5/5 - 1s - loss: 1.2682 - accuracy: 0.5685 - val_loss: 1.9172 - val_accuracy:
0.6825 - lr: 0.2346 - 531ms/epoch - 106ms/step
Epoch 31/2000
5/5 - 0s - loss: 1.1115 - accuracy: 0.6301 - val_loss: 0.9130 - val_accuracy:
0.6825 - lr: 0.2231 - 447ms/epoch - 89ms/step
Epoch 32/2000
5/5 - 0s - loss: 1.0565 - accuracy: 0.6027 - val_loss: 1.0542 - val_accuracy:
0.4444 - lr: 0.2122 - 479ms/epoch - 96ms/step
Epoch 33/2000
```

```
5/5 - 0s - loss: 0.8173 - accuracy: 0.6233 - val_loss: 1.1488 - val_accuracy:
0.6825 - lr: 0.2019 - 467ms/epoch - 93ms/step
Epoch 34/2000
5/5 - 1s - loss: 0.7482 - accuracy: 0.6301 - val_loss: 1.4937 - val_accuracy:
0.3968 - lr: 0.1920 - 507ms/epoch - 101ms/step
Epoch 35/2000
5/5 - 0s - loss: 0.7141 - accuracy: 0.6438 - val_loss: 1.2382 - val_accuracy:
0.6825 - lr: 0.1827 - 474ms/epoch - 95ms/step
Epoch 36/2000
5/5 - 1s - loss: 0.6455 - accuracy: 0.6986 - val_loss: 1.0298 - val_accuracy:
0.6667 - lr: 0.1738 - 531ms/epoch - 106ms/step
Epoch 37/2000
5/5 - 0s - loss: 0.5616 - accuracy: 0.6849 - val_loss: 0.8728 - val_accuracy:
0.7143 - lr: 0.1653 - 471ms/epoch - 94ms/step
Epoch 38/2000
5/5 - 1s - loss: 0.5464 - accuracy: 0.6986 - val_loss: 1.1365 - val_accuracy:
0.4762 - lr: 0.1572 - 526ms/epoch - 105ms/step
Epoch 39/2000
5/5 - 1s - loss: 0.7437 - accuracy: 0.6164 - val_loss: 1.0756 - val_accuracy:
0.6825 - lr: 0.1496 - 568ms/epoch - 114ms/step
Epoch 40/2000
5/5 - 1s - loss: 0.4811 - accuracy: 0.7123 - val_loss: 0.8882 - val_accuracy:
0.6984 - lr: 0.1423 - 530ms/epoch - 106ms/step
Epoch 41/2000
5/5 - 1s - loss: 0.4636 - accuracy: 0.7397 - val_loss: 1.0212 - val_accuracy:
0.6825 - lr: 0.1353 - 544ms/epoch - 109ms/step
Epoch 42/2000
5/5 - 1s - loss: 0.5212 - accuracy: 0.6712 - val_loss: 0.9332 - val_accuracy:
0.6667 - lr: 0.1287 - 526ms/epoch - 105ms/step
Epoch 43/2000
5/5 - 0s - loss: 0.3780 - accuracy: 0.7808 - val_loss: 0.9589 - val_accuracy:
0.6508 - lr: 0.1225 - 465ms/epoch - 93ms/step
Epoch 44/2000
5/5 - 0s - loss: 0.3983 - accuracy: 0.7945 - val_loss: 0.9260 - val_accuracy:
0.6825 - lr: 0.1165 - 487ms/epoch - 97ms/step
Epoch 45/2000
5/5 - 1s - loss: 0.3579 - accuracy: 0.8356 - val_loss: 0.9669 - val_accuracy:
0.6667 - lr: 0.1108 - 519ms/epoch - 104ms/step
Epoch 46/2000
5/5 - 0s - loss: 0.3950 - accuracy: 0.7808 - val_loss: 1.0813 - val_accuracy:
0.5714 - lr: 0.1054 - 497ms/epoch - 99ms/step
Epoch 47/2000
5/5 - 1s - loss: 0.3654 - accuracy: 0.8356 - val_loss: 0.9972 - val_accuracy:
0.6825 - lr: 0.1003 - 512ms/epoch - 102ms/step
Epoch 48/2000
5/5 - 0s - loss: 0.3703 - accuracy: 0.8014 - val_loss: 1.0535 - val_accuracy:
0.6825 - lr: 0.0954 - 496ms/epoch - 99ms/step
Epoch 49/2000
```

```
5/5 - 1s - loss: 0.3701 - accuracy: 0.8493 - val_loss: 0.9798 - val_accuracy:
0.6984 - lr: 0.0907 - 560ms/epoch - 112ms/step
Epoch 50/2000
5/5 - 0s - loss: 0.3396 - accuracy: 0.8425 - val_loss: 0.9852 - val_accuracy:
0.6825 - lr: 0.0863 - 492ms/epoch - 98ms/step
Epoch 51/2000
5/5 - 0s - loss: 0.3483 - accuracy: 0.8219 - val_loss: 0.9980 - val_accuracy:
0.6825 - lr: 0.0821 - 478ms/epoch - 96ms/step
Epoch 52/2000
5/5 - 0s - loss: 0.3074 - accuracy: 0.8493 - val_loss: 0.9666 - val_accuracy:
0.6984 - lr: 0.0781 - 474ms/epoch - 95ms/step
Epoch 53/2000
5/5 - 0s - loss: 0.2804 - accuracy: 0.8836 - val_loss: 1.0521 - val_accuracy:
0.6825 - lr: 0.0743 - 449ms/epoch - 90ms/step
Epoch 54/2000
5/5 - 0s - loss: 0.2949 - accuracy: 0.8493 - val_loss: 1.0505 - val_accuracy:
0.6825 - lr: 0.0707 - 476ms/epoch - 95ms/step
Epoch 55/2000
5/5 - 0s - loss: 0.2959 - accuracy: 0.8425 - val_loss: 1.0176 - val_accuracy:
0.6667 - lr: 0.0672 - 468ms/epoch - 94ms/step
Epoch 56/2000
5/5 - 1s - loss: 0.2786 - accuracy: 0.8836 - val_loss: 1.1549 - val_accuracy:
0.6825 - lr: 0.0639 - 507ms/epoch - 101ms/step
Epoch 57/2000
5/5 - 0s - loss: 0.3509 - accuracy: 0.8288 - val_loss: 1.0024 - val_accuracy:
0.6984 - lr: 0.0608 - 453ms/epoch - 91ms/step
Epoch 58/2000
5/5 - 1s - loss: 0.2768 - accuracy: 0.8699 - val_loss: 0.9680 - val_accuracy:
0.7460 - lr: 0.0578 - 514ms/epoch - 103ms/step
Epoch 59/2000
5/5 - 1s - loss: 0.2604 - accuracy: 0.9110 - val_loss: 1.0276 - val_accuracy:
0.7143 - lr: 0.0550 - 517ms/epoch - 103ms/step
Epoch 60/2000
5/5 - 0s - loss: 0.3033 - accuracy: 0.8425 - val_loss: 1.0434 - val_accuracy:
0.7302 - lr: 0.0523 - 499ms/epoch - 100ms/step
Epoch 61/2000
5/5 - 1s - loss: 0.2767 - accuracy: 0.8562 - val_loss: 0.9844 - val_accuracy:
0.7460 - lr: 0.0498 - 535ms/epoch - 107ms/step
Epoch 62/2000
5/5 - 1s - loss: 0.2401 - accuracy: 0.9315 - val_loss: 1.0034 - val_accuracy:
0.7460 - lr: 0.0474 - 531ms/epoch - 106ms/step
Epoch 63/2000
5/5 - 0s - loss: 0.2744 - accuracy: 0.8699 - val_loss: 1.0253 - val_accuracy:
0.6984 - lr: 0.0450 - 448ms/epoch - 90ms/step
Epoch 64/2000
5/5 - 1s - loss: 0.2320 - accuracy: 0.9247 - val_loss: 1.0330 - val_accuracy:
0.6825 - lr: 0.0429 - 501ms/epoch - 100ms/step
Epoch 65/2000
```

```
5/5 - 0s - loss: 0.2227 - accuracy: 0.9452 - val_loss: 1.0187 - val_accuracy:
0.7143 - lr: 0.0408 - 469ms/epoch - 94ms/step
Epoch 66/2000
5/5 - 1s - loss: 0.2246 - accuracy: 0.9452 - val_loss: 1.0367 - val_accuracy:
0.6667 - lr: 0.0388 - 510ms/epoch - 102ms/step
Epoch 66: early stopping
```

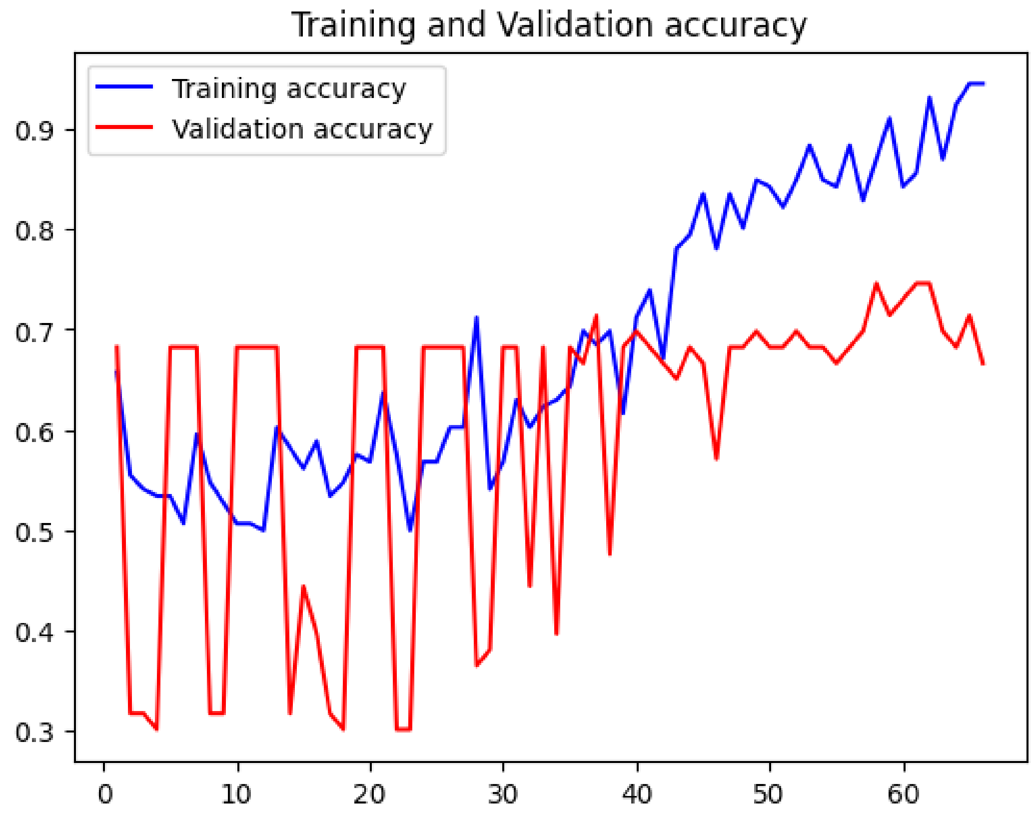
[ ]:
```
## Desempenho do aprendizado
Podemos observar, por meio de um gráfico, o desempenho do aprendizado ao longo␣
↪das épocas, representando a acurácia do modelo para os conjuntos de␣
↪treinamento e validação, bem como o valor da função de custo.
```

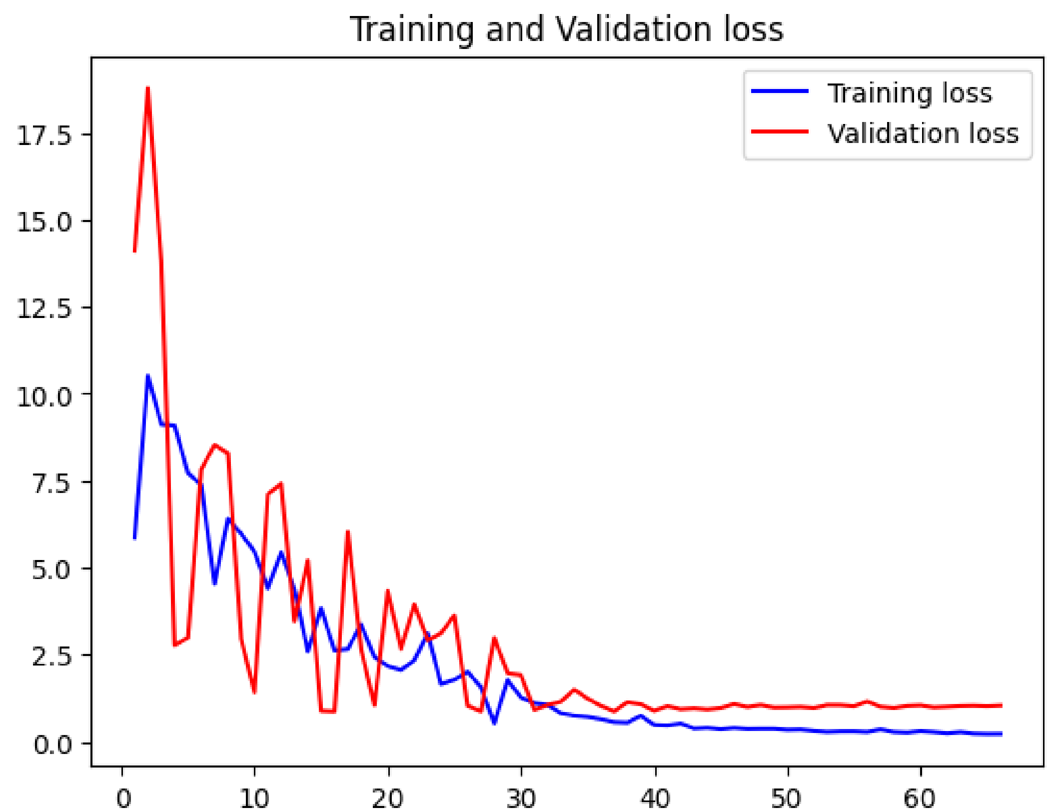
[20]:
```
acc = results.history['accuracy']
val_acc = results.history['val_accuracy']
loss = results.history['loss']
val_loss = results.history['val_loss']
epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'b', label= 'Training accuracy')
plt.plot(epochs, val_acc, 'r', label= 'Validation accuracy')
plt.title('Training and Validation accuracy')
plt.legend()

plt.figure()
plt.plot(epochs, loss, 'b', label= 'Training loss')
plt.plot(epochs, val_loss, 'r', label= 'Validation loss')
plt.title('Training and Validation loss')
plt.legend()

plt.show()
```


Training and Validation accuracy
Training accuracy
Validation accuracy
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
10
20
30
40
50
60



[ ]:
```
## Desempenho da rede
Após configurar e treinar a rede, realizaremos testes de desempenho para␣
↪avaliar o funcionamento do modelo.
```

[21]:
```
from sklearn.metrics import accuracy_score

ytrainpred = modelo.predict(train_set_x)
ytestpred = modelo.predict(test_set_x)

print('Acurácia sobre o arquivo de treino = {:.1f}%'.
↪format(accuracy_score(train_set_y.argmax(axis=1), ytrainpred.argmax(axis=1))␣
↪* 100))
print('Acurácia sobre o arquivo de testes = {:.1f}%'.
↪format(accuracy_score(test_set_y.argmax(axis=1), ytestpred.argmax(axis=1)) *␣
↪100))

print('\nResultados esperados do arquivo de teste:', *test_set_y_orig)
print('Resultados obtidos do arquivo de teste:  ', *[ytestpred[x].argmax() for␣
↪x in range(len(ytestpred))])
```

```
7/7 [==============================] - 0s 22ms/step
```

```
2/2 [==============================] - 0s 19ms/step
Acurácia sobre o arquivo de treino = 88.5%
Acurácia sobre o arquivo de testes = 74.0%

Resultados esperados do arquivo de teste: 1 1 1 1 1 0 1 1 1 1 1 1 1 0 0 1 0 1 1
1 1 0 0 1 1 1 1 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1 1 0 0 0 1 1 1 0
Resultados obtidos do arquivo de teste:   1 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 0 1 0
0 1 0 0 0 1 1 0 1 1 0 1 1 1 1 1 0 0 1 1 0 0 0 1 0 1 0 0 1 1 0
```

[ ]:

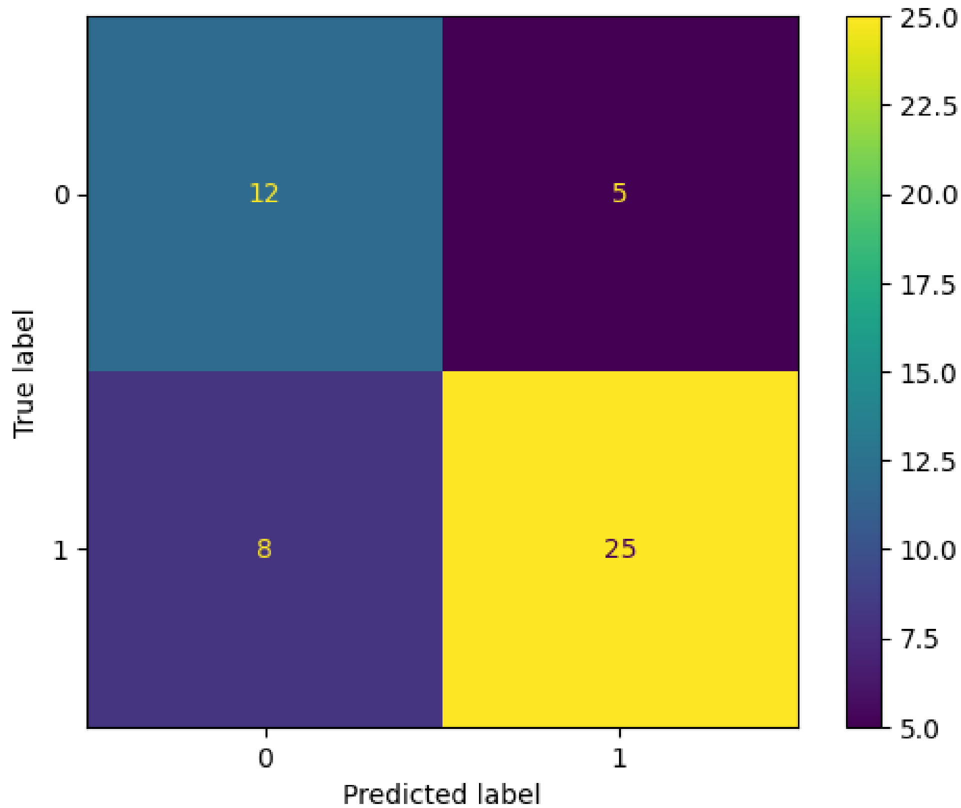
```
## Matriz de confusão
Com base nos resultados obtidos, será possível construir uma matriz de
confusão, que mostrará o desempenho do modelo em relação à base de dados de
teste.
```

[22]:

```python
ConfusionMatrixDisplay.from_predictions(test_set_y.argmax(axis=1), ytestpred.
argmax(axis=1))
```

[22]:

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at
0x7f9b1c1bbeb0>
```



[ ]:
```
# **Rede Convolucional**
Agora, vamos utilizar um modelo mais robusto para identificação de imagens.
```

[ ]:
```
## Criando um conjunto de validação
Novamente, faremos uso de um conjunto de validação, desta vez com 21 fotos.
```

[23]:
```
Xtr,Xval,ytr,yval = train_test_split(train_set_x_orig,train_set_y,test_size = 0.
 ↪1)
num_train = np.size(Xtr,0)
print(num_train)
```

188

[ ]:
```
## Definindo um novo modelo
Para esta abordagem, vamos utilizar uma rede neural com 21 camadas:
*   4 camadas de convolução, com ativação *relu*;
*   9 camadas de normalização do *batch*
*   4 camadas de *pooling*;
*   1 camada *flatten*
*   1 camada de *dropout*;
*   2 camadas NN densas.
```

[24]:
```
model_cnn = keras.Sequential(
    [
        keras.Input(shape=(64,64,3)),
        layers.Conv2D(32, (3,3), activation = 'relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2,2)),
        layers.BatchNormalization(),
        layers.Conv2D(64, (3,3), activation = 'relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2,2)),
        layers.BatchNormalization(),
        layers.Conv2D(128, (3,3), activation = 'relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2,2)),
        layers.BatchNormalization(),
        layers.Conv2D(128, (3,3), activation = 'relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2,2)),
        layers.BatchNormalization(),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(512,activation='relu'),
        layers.BatchNormalization(),
        layers.Dense(2,activation='softmax'),
    ]
```

```
)

model_cnn.compile(
  loss='categorical_crossentropy',
  optimizer='adagrad',
  metrics=['accuracy'],
)
model_cnn.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d (Conv2D)             (None, 62, 62, 32)        896       
                                                                 
 batch_normalization (BatchN  (None, 62, 62, 32)       128       
 ormalization)                                                   
                                                                 
 max_pooling2d (MaxPooling2D  (None, 31, 31, 32)       0         
 )                                                               
                                                                 
 batch_normalization_1 (Batc  (None, 31, 31, 32)       128       
 hNormalization)                                                 
                                                                 
 conv2d_1 (Conv2D)           (None, 29, 29, 64)        18496     
                                                                 
 batch_normalization_2 (Batc  (None, 29, 29, 64)       256       
 hNormalization)                                                 
                                                                 
 max_pooling2d_1 (MaxPooling  (None, 14, 14, 64)       0         
 2D)                                                             
                                                                 
 batch_normalization_3 (Batc  (None, 14, 14, 64)       256       
 hNormalization)                                                 
                                                                 
 conv2d_2 (Conv2D)           (None, 12, 12, 128)       73856     
                                                                 
 batch_normalization_4 (Batc  (None, 12, 12, 128)      512       
 hNormalization)                                                 
                                                                 
 max_pooling2d_2 (MaxPooling  (None, 6, 6, 128)        0         
 2D)                                                             
                                                                 
 batch_normalization_5 (Batc  (None, 6, 6, 128)        512       
 hNormalization)                                                 
                                                                 
 conv2d_3 (Conv2D)           (None, 4, 4, 128)         147584    
```

```
 batch_normalization_6 (Batc  (None, 4, 4, 128)        512
 hNormalization)

 max_pooling2d_3 (MaxPooling  (None, 2, 2, 128)        0
 2D)

 batch_normalization_7 (Batc  (None, 2, 2, 128)        512
 hNormalization)

 flatten_1 (Flatten)         (None, 512)               0

 dropout (Dropout)           (None, 512)               0

 dense_2 (Dense)             (None, 512)               262656

 batch_normalization_8 (Batc  (None, 512)              2048
 hNormalization)

 dense_3 (Dense)             (None, 2)                 1026

=================================================================
Total params: 509,378
Trainable params: 506,946
Non-trainable params: 2,432
_________________________________________________________________
```

[ ]:
```
## Treinando o modelo
Agora treinaremos o modelo criado, utilizando um *batch_size* de tamanho 16 e␣
↪100 épocas.
```

[25]:
```
# Callback
early_stop = keras.callbacks.EarlyStopping(monitor='val_loss', patience=100)

history = model_cnn.fit(
  Xtr,
  ytr,
  epochs=100,
  batch_size=16,
  validation_data=(Xval, yval),
  verbose = 2,
  callbacks=[early_stop]
)
```

```
Epoch 1/100
12/12 - 5s - loss: 0.8450 - accuracy: 0.5745 - val_loss: 0.6738 - val_accuracy:
0.5714 - 5s/epoch - 386ms/step
Epoch 2/100
```

```
12/12 - 2s - loss: 0.6400 - accuracy: 0.7181 - val_loss: 0.6738 - val_accuracy:
0.5714 - 2s/epoch - 128ms/step
Epoch 3/100
12/12 - 2s - loss: 0.6590 - accuracy: 0.6702 - val_loss: 0.6316 - val_accuracy:
0.5714 - 2s/epoch - 130ms/step
Epoch 4/100
12/12 - 1s - loss: 0.6297 - accuracy: 0.7394 - val_loss: 0.6401 - val_accuracy:
0.5714 - 1s/epoch - 117ms/step
Epoch 5/100
12/12 - 1s - loss: 0.5895 - accuracy: 0.7128 - val_loss: 0.6206 - val_accuracy:
0.5714 - 1s/epoch - 118ms/step
Epoch 6/100
12/12 - 2s - loss: 0.4083 - accuracy: 0.8138 - val_loss: 0.6003 - val_accuracy:
0.5238 - 2s/epoch - 130ms/step
Epoch 7/100
12/12 - 2s - loss: 0.4781 - accuracy: 0.7766 - val_loss: 0.5559 - val_accuracy:
0.6190 - 2s/epoch - 131ms/step
Epoch 8/100
12/12 - 2s - loss: 0.4242 - accuracy: 0.8032 - val_loss: 0.5590 - val_accuracy:
0.6667 - 2s/epoch - 141ms/step
Epoch 9/100
12/12 - 2s - loss: 0.3678 - accuracy: 0.8511 - val_loss: 0.5473 - val_accuracy:
0.6667 - 2s/epoch - 131ms/step
Epoch 10/100
12/12 - 2s - loss: 0.3522 - accuracy: 0.8457 - val_loss: 0.5464 - val_accuracy:
0.7143 - 2s/epoch - 128ms/step
Epoch 11/100
12/12 - 1s - loss: 0.3389 - accuracy: 0.8457 - val_loss: 0.5548 - val_accuracy:
0.6667 - 1s/epoch - 114ms/step
Epoch 12/100
12/12 - 2s - loss: 0.3272 - accuracy: 0.8457 - val_loss: 0.5506 - val_accuracy:
0.7143 - 2s/epoch - 132ms/step
Epoch 13/100
12/12 - 2s - loss: 0.3315 - accuracy: 0.8564 - val_loss: 0.5329 - val_accuracy:
0.6190 - 2s/epoch - 126ms/step
Epoch 14/100
12/12 - 1s - loss: 0.3672 - accuracy: 0.8298 - val_loss: 0.5153 - val_accuracy:
0.6667 - 1s/epoch - 123ms/step
Epoch 15/100
12/12 - 1s - loss: 0.3185 - accuracy: 0.8404 - val_loss: 0.5118 - val_accuracy:
0.6667 - 1s/epoch - 120ms/step
Epoch 16/100
12/12 - 2s - loss: 0.2371 - accuracy: 0.8830 - val_loss: 0.5098 - val_accuracy:
0.7143 - 2s/epoch - 132ms/step
Epoch 17/100
12/12 - 1s - loss: 0.2374 - accuracy: 0.8777 - val_loss: 0.5135 - val_accuracy:
0.6667 - 1s/epoch - 120ms/step
Epoch 18/100
```

```
12/12 - 1s - loss: 0.2231 - accuracy: 0.9149 - val_loss: 0.5074 - val_accuracy:
0.6667 - 1s/epoch - 113ms/step
Epoch 19/100
12/12 - 1s - loss: 0.2263 - accuracy: 0.9096 - val_loss: 0.5143 - val_accuracy:
0.7143 - 1s/epoch - 120ms/step
Epoch 20/100
12/12 - 1s - loss: 0.1935 - accuracy: 0.9096 - val_loss: 0.5126 - val_accuracy:
0.7619 - 1s/epoch - 121ms/step
Epoch 21/100
12/12 - 1s - loss: 0.2462 - accuracy: 0.8777 - val_loss: 0.5276 - val_accuracy:
0.6667 - 1s/epoch - 112ms/step
Epoch 22/100
12/12 - 1s - loss: 0.2199 - accuracy: 0.9043 - val_loss: 0.5178 - val_accuracy:
0.6667 - 1s/epoch - 120ms/step
Epoch 23/100
12/12 - 2s - loss: 0.2495 - accuracy: 0.9043 - val_loss: 0.5169 - val_accuracy:
0.6667 - 2s/epoch - 132ms/step
Epoch 24/100
12/12 - 1s - loss: 0.2114 - accuracy: 0.8989 - val_loss: 0.5141 - val_accuracy:
0.7619 - 1s/epoch - 120ms/step
Epoch 25/100
12/12 - 1s - loss: 0.2239 - accuracy: 0.8883 - val_loss: 0.5321 - val_accuracy:
0.7143 - 1s/epoch - 112ms/step
Epoch 26/100
12/12 - 1s - loss: 0.1681 - accuracy: 0.9362 - val_loss: 0.5359 - val_accuracy:
0.7619 - 1s/epoch - 124ms/step
Epoch 27/100
12/12 - 1s - loss: 0.1882 - accuracy: 0.9202 - val_loss: 0.4944 - val_accuracy:
0.7619 - 1s/epoch - 120ms/step
Epoch 28/100
12/12 - 2s - loss: 0.1649 - accuracy: 0.9255 - val_loss: 0.4805 - val_accuracy:
0.7619 - 2s/epoch - 131ms/step
Epoch 29/100
12/12 - 1s - loss: 0.2017 - accuracy: 0.9096 - val_loss: 0.4775 - val_accuracy:
0.8095 - 1s/epoch - 125ms/step
Epoch 30/100
12/12 - 2s - loss: 0.1342 - accuracy: 0.9681 - val_loss: 0.4980 - val_accuracy:
0.7143 - 2s/epoch - 131ms/step
Epoch 31/100
12/12 - 2s - loss: 0.1555 - accuracy: 0.9362 - val_loss: 0.4813 - val_accuracy:
0.7143 - 2s/epoch - 129ms/step
Epoch 32/100
12/12 - 2s - loss: 0.1531 - accuracy: 0.9468 - val_loss: 0.4750 - val_accuracy:
0.7143 - 2s/epoch - 130ms/step
Epoch 33/100
12/12 - 2s - loss: 0.1390 - accuracy: 0.9415 - val_loss: 0.4834 - val_accuracy:
0.7143 - 2s/epoch - 128ms/step
Epoch 34/100
```

```
12/12 - 1s - loss: 0.1227 - accuracy: 0.9628 - val_loss: 0.4960 - val_accuracy:
0.7143 - 1s/epoch - 118ms/step
Epoch 35/100
12/12 - 1s - loss: 0.1253 - accuracy: 0.9521 - val_loss: 0.5204 - val_accuracy:
0.7143 - 1s/epoch - 123ms/step
Epoch 36/100
12/12 - 1s - loss: 0.1330 - accuracy: 0.9468 - val_loss: 0.5035 - val_accuracy:
0.7143 - 1s/epoch - 124ms/step
Epoch 37/100
12/12 - 2s - loss: 0.1416 - accuracy: 0.9574 - val_loss: 0.4945 - val_accuracy:
0.7143 - 2s/epoch - 131ms/step
Epoch 38/100
12/12 - 1s - loss: 0.1492 - accuracy: 0.9468 - val_loss: 0.4953 - val_accuracy:
0.7143 - 1s/epoch - 119ms/step
Epoch 39/100
12/12 - 2s - loss: 0.1616 - accuracy: 0.9362 - val_loss: 0.5117 - val_accuracy:
0.7143 - 2s/epoch - 128ms/step
Epoch 40/100
12/12 - 2s - loss: 0.1188 - accuracy: 0.9628 - val_loss: 0.5203 - val_accuracy:
0.7143 - 2s/epoch - 128ms/step
Epoch 41/100
12/12 - 1s - loss: 0.1007 - accuracy: 0.9734 - val_loss: 0.5202 - val_accuracy:
0.7143 - 1s/epoch - 118ms/step
Epoch 42/100
12/12 - 2s - loss: 0.0992 - accuracy: 0.9787 - val_loss: 0.5624 - val_accuracy:
0.7619 - 2s/epoch - 152ms/step
Epoch 43/100
12/12 - 1s - loss: 0.1342 - accuracy: 0.9521 - val_loss: 0.5912 - val_accuracy:
0.6667 - 1s/epoch - 121ms/step
Epoch 44/100
12/12 - 1s - loss: 0.1737 - accuracy: 0.9149 - val_loss: 0.5637 - val_accuracy:
0.7143 - 1s/epoch - 124ms/step
Epoch 45/100
12/12 - 1s - loss: 0.0951 - accuracy: 0.9734 - val_loss: 0.5191 - val_accuracy:
0.7619 - 1s/epoch - 122ms/step
Epoch 46/100
12/12 - 1s - loss: 0.1178 - accuracy: 0.9628 - val_loss: 0.5059 - val_accuracy:
0.7619 - 1s/epoch - 125ms/step
Epoch 47/100
12/12 - 1s - loss: 0.0819 - accuracy: 0.9787 - val_loss: 0.5151 - val_accuracy:
0.7619 - 1s/epoch - 122ms/step
Epoch 48/100
12/12 - 1s - loss: 0.1390 - accuracy: 0.9309 - val_loss: 0.4912 - val_accuracy:
0.7619 - 1s/epoch - 119ms/step
Epoch 49/100
12/12 - 2s - loss: 0.1217 - accuracy: 0.9681 - val_loss: 0.5182 - val_accuracy:
0.7619 - 2s/epoch - 125ms/step
Epoch 50/100
```

```
12/12 - 2s - loss: 0.1175 - accuracy: 0.9681 - val_loss: 0.4860 - val_accuracy:
0.7619 - 2s/epoch - 127ms/step
Epoch 51/100
12/12 - 1s - loss: 0.1031 - accuracy: 0.9734 - val_loss: 0.5119 - val_accuracy:
0.7619 - 1s/epoch - 120ms/step
Epoch 52/100
12/12 - 1s - loss: 0.1045 - accuracy: 0.9840 - val_loss: 0.4746 - val_accuracy:
0.8095 - 1s/epoch - 120ms/step
Epoch 53/100
12/12 - 1s - loss: 0.0546 - accuracy: 0.9894 - val_loss: 0.5029 - val_accuracy:
0.8095 - 1s/epoch - 119ms/step
Epoch 54/100
12/12 - 1s - loss: 0.0688 - accuracy: 0.9787 - val_loss: 0.5189 - val_accuracy:
0.8095 - 1s/epoch - 119ms/step
Epoch 55/100
12/12 - 1s - loss: 0.1575 - accuracy: 0.9309 - val_loss: 0.4945 - val_accuracy:
0.8095 - 1s/epoch - 121ms/step
Epoch 56/100
12/12 - 2s - loss: 0.0935 - accuracy: 0.9787 - val_loss: 0.5191 - val_accuracy:
0.8095 - 2s/epoch - 133ms/step
Epoch 57/100
12/12 - 2s - loss: 0.0514 - accuracy: 0.9894 - val_loss: 0.5759 - val_accuracy:
0.8095 - 2s/epoch - 130ms/step
Epoch 58/100
12/12 - 1s - loss: 0.0721 - accuracy: 0.9734 - val_loss: 0.5277 - val_accuracy:
0.8095 - 1s/epoch - 112ms/step
Epoch 59/100
12/12 - 1s - loss: 0.0764 - accuracy: 0.9840 - val_loss: 0.5027 - val_accuracy:
0.8095 - 1s/epoch - 118ms/step
Epoch 60/100
12/12 - 1s - loss: 0.0957 - accuracy: 0.9734 - val_loss: 0.5239 - val_accuracy:
0.8095 - 1s/epoch - 118ms/step
Epoch 61/100
12/12 - 1s - loss: 0.1039 - accuracy: 0.9681 - val_loss: 0.4757 - val_accuracy:
0.8095 - 1s/epoch - 118ms/step
Epoch 62/100
12/12 - 1s - loss: 0.0695 - accuracy: 0.9787 - val_loss: 0.4889 - val_accuracy:
0.8095 - 1s/epoch - 115ms/step
Epoch 63/100
12/12 - 2s - loss: 0.0777 - accuracy: 0.9734 - val_loss: 0.5305 - val_accuracy:
0.8095 - 2s/epoch - 132ms/step
Epoch 64/100
12/12 - 2s - loss: 0.1169 - accuracy: 0.9681 - val_loss: 0.5146 - val_accuracy:
0.8095 - 2s/epoch - 129ms/step
Epoch 65/100
12/12 - 1s - loss: 0.0924 - accuracy: 0.9681 - val_loss: 0.4818 - val_accuracy:
0.8571 - 1s/epoch - 122ms/step
Epoch 66/100
```

```
12/12 - 1s - loss: 0.1061 - accuracy: 0.9681 - val_loss: 0.4902 - val_accuracy:
0.8095 - 1s/epoch - 116ms/step
Epoch 67/100
12/12 - 1s - loss: 0.0747 - accuracy: 0.9681 - val_loss: 0.5148 - val_accuracy:
0.8095 - 1s/epoch - 120ms/step
Epoch 68/100
12/12 - 1s - loss: 0.0537 - accuracy: 0.9894 - val_loss: 0.5029 - val_accuracy:
0.8571 - 1s/epoch - 115ms/step
Epoch 69/100
12/12 - 1s - loss: 0.0599 - accuracy: 0.9894 - val_loss: 0.5353 - val_accuracy:
0.8095 - 1s/epoch - 125ms/step
Epoch 70/100
12/12 - 2s - loss: 0.0475 - accuracy: 1.0000 - val_loss: 0.4909 - val_accuracy:
0.8571 - 2s/epoch - 126ms/step
Epoch 71/100
12/12 - 2s - loss: 0.0684 - accuracy: 0.9947 - val_loss: 0.5454 - val_accuracy:
0.8095 - 2s/epoch - 131ms/step
Epoch 72/100
12/12 - 1s - loss: 0.0682 - accuracy: 0.9840 - val_loss: 0.4679 - val_accuracy:
0.8571 - 1s/epoch - 121ms/step
Epoch 73/100
12/12 - 2s - loss: 0.0808 - accuracy: 0.9894 - val_loss: 0.4628 - val_accuracy:
0.8571 - 2s/epoch - 137ms/step
Epoch 74/100
12/12 - 2s - loss: 0.0939 - accuracy: 0.9681 - val_loss: 0.5551 - val_accuracy:
0.8095 - 2s/epoch - 126ms/step
Epoch 75/100
12/12 - 1s - loss: 0.0694 - accuracy: 0.9840 - val_loss: 0.5071 - val_accuracy:
0.8095 - 1s/epoch - 121ms/step
Epoch 76/100
12/12 - 2s - loss: 0.1378 - accuracy: 0.9309 - val_loss: 0.4803 - val_accuracy:
0.8571 - 2s/epoch - 127ms/step
Epoch 77/100
12/12 - 1s - loss: 0.0657 - accuracy: 0.9787 - val_loss: 0.4622 - val_accuracy:
0.8095 - 1s/epoch - 124ms/step
Epoch 78/100
12/12 - 2s - loss: 0.0682 - accuracy: 0.9734 - val_loss: 0.4938 - val_accuracy:
0.8571 - 2s/epoch - 130ms/step
Epoch 79/100
12/12 - 1s - loss: 0.0646 - accuracy: 0.9840 - val_loss: 0.5248 - val_accuracy:
0.8095 - 1s/epoch - 119ms/step
Epoch 80/100
12/12 - 1s - loss: 0.0474 - accuracy: 0.9894 - val_loss: 0.5607 - val_accuracy:
0.8095 - 1s/epoch - 125ms/step
Epoch 81/100
12/12 - 1s - loss: 0.0647 - accuracy: 0.9787 - val_loss: 0.5343 - val_accuracy:
0.7619 - 1s/epoch - 119ms/step
Epoch 82/100
```

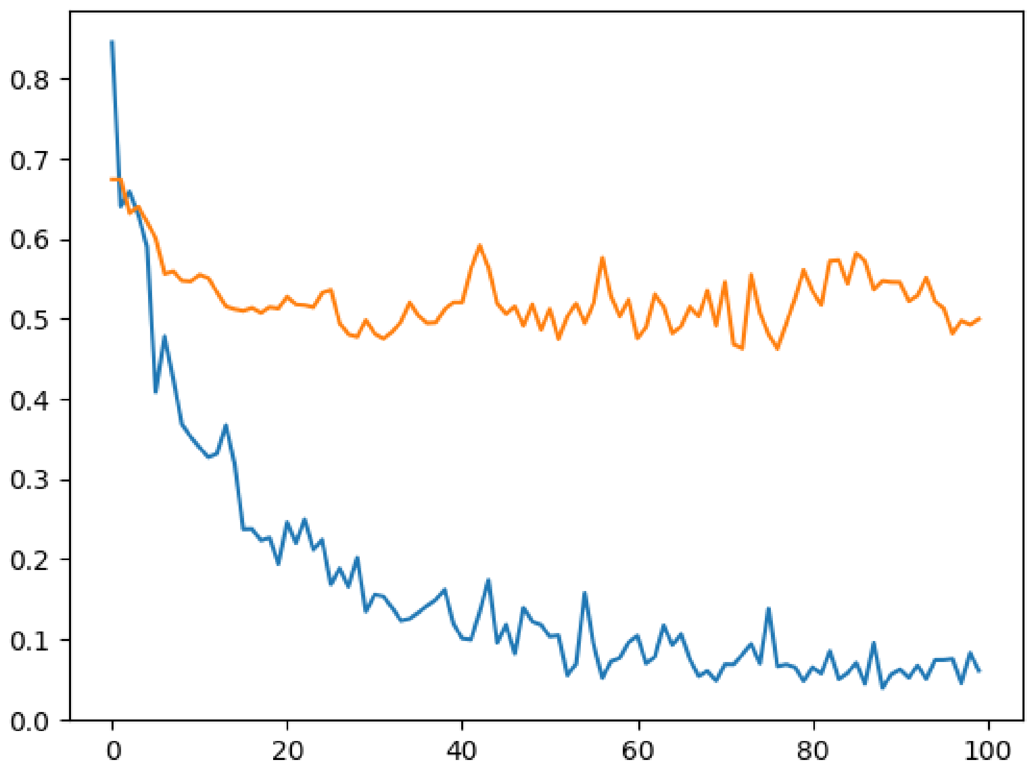
```
12/12 - 2s - loss: 0.0568 - accuracy: 0.9840 - val_loss: 0.5170 - val_accuracy:
0.7619 - 2s/epoch - 129ms/step
Epoch 83/100
12/12 - 2s - loss: 0.0849 - accuracy: 0.9734 - val_loss: 0.5720 - val_accuracy:
0.8095 - 2s/epoch - 129ms/step
Epoch 84/100
12/12 - 1s - loss: 0.0496 - accuracy: 0.9840 - val_loss: 0.5731 - val_accuracy:
0.7619 - 1s/epoch - 116ms/step
Epoch 85/100
12/12 - 1s - loss: 0.0574 - accuracy: 0.9840 - val_loss: 0.5435 - val_accuracy:
0.7619 - 1s/epoch - 123ms/step
Epoch 86/100
12/12 - 2s - loss: 0.0702 - accuracy: 0.9787 - val_loss: 0.5816 - val_accuracy:
0.8095 - 2s/epoch - 127ms/step
Epoch 87/100
12/12 - 1s - loss: 0.0439 - accuracy: 0.9894 - val_loss: 0.5720 - val_accuracy:
0.7619 - 1s/epoch - 119ms/step
Epoch 88/100
12/12 - 2s - loss: 0.0954 - accuracy: 0.9734 - val_loss: 0.5362 - val_accuracy:
0.8095 - 2s/epoch - 127ms/step
Epoch 89/100
12/12 - 2s - loss: 0.0389 - accuracy: 1.0000 - val_loss: 0.5470 - val_accuracy:
0.7619 - 2s/epoch - 140ms/step
Epoch 90/100
12/12 - 2s - loss: 0.0557 - accuracy: 0.9734 - val_loss: 0.5455 - val_accuracy:
0.8095 - 2s/epoch - 147ms/step
Epoch 91/100
12/12 - 2s - loss: 0.0620 - accuracy: 0.9787 - val_loss: 0.5453 - val_accuracy:
0.8095 - 2s/epoch - 126ms/step
Epoch 92/100
12/12 - 2s - loss: 0.0517 - accuracy: 0.9947 - val_loss: 0.5219 - val_accuracy:
0.8095 - 2s/epoch - 129ms/step
Epoch 93/100
12/12 - 2s - loss: 0.0667 - accuracy: 0.9787 - val_loss: 0.5288 - val_accuracy:
0.8571 - 2s/epoch - 127ms/step
Epoch 94/100
12/12 - 1s - loss: 0.0501 - accuracy: 0.9894 - val_loss: 0.5513 - val_accuracy:
0.8095 - 1s/epoch - 118ms/step
Epoch 95/100
12/12 - 2s - loss: 0.0737 - accuracy: 0.9840 - val_loss: 0.5220 - val_accuracy:
0.8571 - 2s/epoch - 132ms/step
Epoch 96/100
12/12 - 2s - loss: 0.0738 - accuracy: 0.9734 - val_loss: 0.5125 - val_accuracy:
0.8571 - 2s/epoch - 158ms/step
Epoch 97/100
12/12 - 2s - loss: 0.0752 - accuracy: 0.9734 - val_loss: 0.4815 - val_accuracy:
0.8095 - 2s/epoch - 126ms/step
Epoch 98/100
```

```
12/12 - 1s - loss: 0.0445 - accuracy: 0.9947 - val_loss: 0.4974 - val_accuracy:
0.8571 - 1s/epoch - 123ms/step
Epoch 99/100
12/12 - 2s - loss: 0.0826 - accuracy: 0.9628 - val_loss: 0.4925 - val_accuracy:
0.8095 - 2s/epoch - 133ms/step
Epoch 100/100
12/12 - 1s - loss: 0.0604 - accuracy: 0.9894 - val_loss: 0.4993 - val_accuracy:
0.7619 - 1s/epoch - 119ms/step
```

[ ]:
```
## Desempenho do aprendizado
Será possível visualizar o desempenho do aprendizado ao longo das épocas por meio de um gráfico.
```

[27]:
```
plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='test')
```

[27]:
```
[<matplotlib.lines.Line2D at 0x7f9b1e79e6a0>]
```



[ ]:
```
## Desempenho da rede
Após configurar e treinar a rede, realizaremos testes de desempenho para avaliar o funcionamento do modelo.
```

[29]:
```python
ytrainpred = model_cnn.predict(train_set_x_orig)
ytestpred = model_cnn.predict(test_set_x_orig)

print('Acurácia sobre o arquivo de treino = {:.1f}%'.format(accuracy_score(train_set_y.argmax(axis=1), ytrainpred.argmax(axis=1)) * 100))
print('Acurácia sobre o arquivo de testes = {:.1f}%'.format(accuracy_score(test_set_y.argmax(axis=1), ytestpred.argmax(axis=1)) * 100))

print('\nResultados esperados do arquivo de teste:', *test_set_y_orig)
print('Resultados obtidos do arquivo de teste:  ', *[ytestpred[x].argmax() for x in range(len(ytestpred))])
```

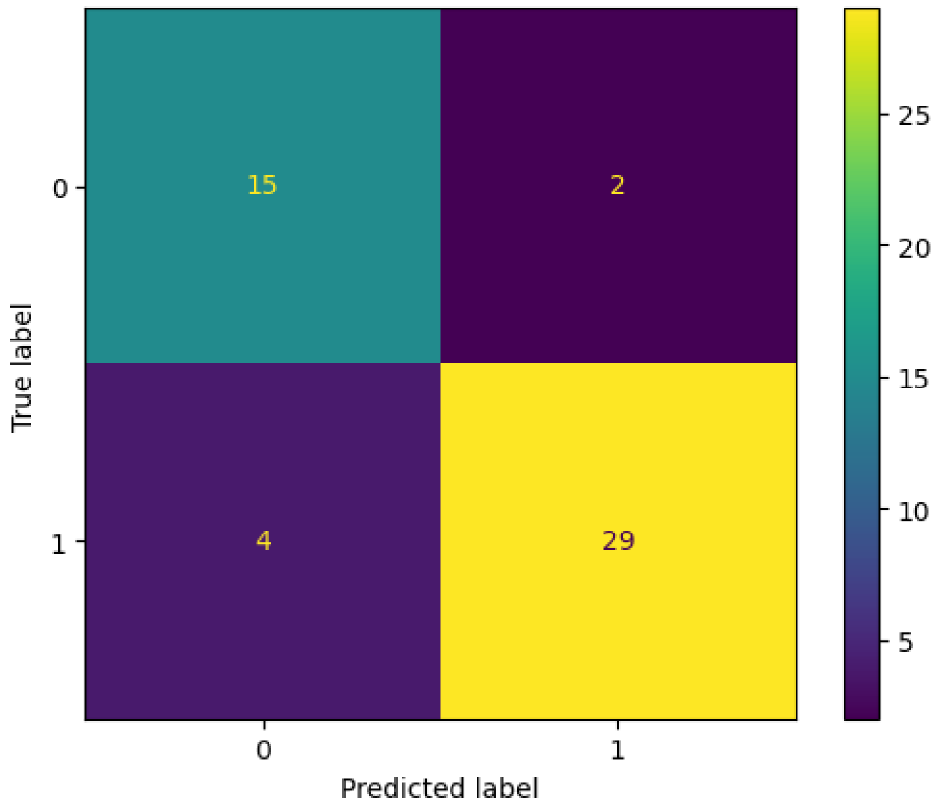
```
7/7 [==============================] - 1s 50ms/step
2/2 [==============================] - 0s 34ms/step
Acurácia sobre o arquivo de treino = 97.6%
Acurácia sobre o arquivo de testes = 88.0%

Resultados esperados do arquivo de teste: 1 1 1 1 1 0 1 1 1 1 1 1 1 0 0 1 0 1 1
1 1 0 0 1 1 1 1 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1 1 0 0 0 1 1 1 0
Resultados obtidos do arquivo de teste:   1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 0 1 0
0 1 1 0 1 1 1 0 0 1 0 1 1 1 1 0 0 0 1 0 0 1 0 1 0 0 0 1 1 1 0
```

[ ]:
```
## Matriz de confusão
Com base nos resultados obtidos, construiremos uma matriz de confusão para analisar o desempenho do modelo em relação aos dados de teste.
```

[30]:
```python
ConfusionMatrixDisplay.from_predictions(test_set_y.argmax(axis=1), ytestpred.argmax(axis=1))
```

[30]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at
0x7f9b1e768eb0>
```


15
2
4
29
True label
Predicted label
0
1
0
1
25
20
15
10
5