Trabalho de Deep Learning

Lucas de Almeida, RA: 1996762
Vinícius Augusto de Souza, RA: 1997530

Uma abordagem dos conceitos de redes neurais convolucionais pré treinadas utilizando o método de *transfer learning*.
A base de dados a ser utilizada será a *Basic Shapes*, que pode ser encontrada neste link (https://www.kaggle.com/cactus3/basicshapes). Ela é bem simples e objetiva, contendo um acervo de 300 imagens: 100 imagens de cada forma geométrica (circulos, quadrados e triangulos) desenhados manualmente por Mark S. (https://www.kaggle.com/cactus3)

Considerações sobre o trabalho:

Para o desenvolvimento deste trabalho, foi escolhido o modelo pré-treinado da literatura *InceptionResNetV2 (https://keras.io/api/applications/inceptionresnetv2/)*, considerando os pesos também pré treinados da *ImageNet*.

In [1]:
```
import tensorflow as tf
import numpy as np
import pandas as pd
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sn
import os
import random
from tqdm import tqdm
import shutil
import pathlib
```

```
INFO:tensorflow:Enabling eager execution
INFO:tensorflow:Enabling v2 tensorshape
INFO:tensorflow:Enabling resource variables
INFO:tensorflow:Enabling tensor equality
INFO:tensorflow:Enabling control flow v2
```

Separação dos arquivos:

Antes de começar a tratar os dados, é necessário realizar a divisão do dataset em 80% para treino e 20% para testes.
O código a seguir realiza essa tarefa, passando como variáveis:

- *data_dir*: o nome do caminho do diretório do dataset escolhido;
- *classes*: as classes das quais os arquivos pertencem (circulos, quadrados e triângulos);

- *output_dir*: o nome do caminho do diretório de saída, após separação;
- *ratio*: a taxa de divisão dos arquivos (i.e. 80% treino e 20% testes);

Em seguida são carregadas essas variávies e todas as imagens *.png* são selecionadas e armazenadas na variável *file*.

Em seguida, é escolhida uma *seed* arbitrária para embaralhar as imagens de forma que seja possível reproduzir essa mesma divisão posteriormente. Os arquivos então são embaralhados e divididos conforme os parâmetros passados anteriormente.

Por fim, os arquivos divididos são enviados cada um para seu respectivo diretório, treino ou teste, e sempre divididos em pastas referentes à sua classe, conforme a estrutura a seguir:

- *test* (80% do dataset)
  - *circles*
  - *triangles*
  - *squares*
- *train* (20% do dataset)
  - *circles*
  - *triangles*
  - *squares*

In [2]:
```
data_dir = "shapes"
classes = ["circles", "squares", "triangles"]
output_dir = "shapes_split"
ratio = [0.8, 0.2]

def split(data_dir, output_dir, ratio):
    for cell in classes:
        cell_path = os.path.join(data_dir, cell)
        files = os.listdir(cell_path)
        files = [os.path.join(cell_path, f) for f in files if f.endswith('.png')]

        random.seed(230)
        files.sort()
        random.shuffle(files)

        split_train = int(ratio[0] * len(files))
        split_test = split_train

        files_train = files[:split_train]
        files_test = files[split_test:]
        files_type = [(files_train, "train"), (files_test, "test")]

        for (files, folder_type) in files_type:
            full_path = os.path.join(output_dir, folder_type)
            full_path = os.path.join(full_path, cell)
            pathlib.Path(full_path).mkdir(parents=True, exist_ok=True)
            for f in files:
                shutil.copy2(f, full_path)
```

Definição dos parâmetros do modelo:

Antes de prosseguir, é necessário compreender em quais parâmetros o modelo se baseia. Dois desses parâmetros terão de ser definidos neste momento, os quais são:

- *epochs*: quantas vezes o algoritmo de treino será executado;
- *batch size*: quantas amostras serão carregadas a cada uma dessas execuções.

Para o treinamento, foi definido que seriam utilizadas 500 *epochs* e 32 como *batch size*.

In [3]:
```
epochs = 500
batch = 32
```

Carregamento do modelo "*InceptionResNetV2*":

Primeiramente o modelo escolhido é carregado juntamente com os pesos aprendidos durante o treino (sem a camada densa) para a variável *base_model*. Em seguida, a variável *x* recebe a saída do modelo carregado.

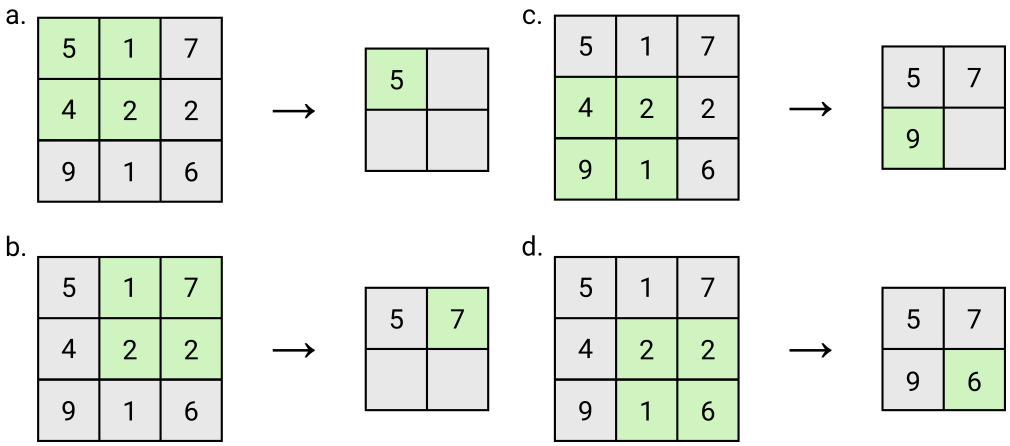
In [4]:
```
base_model = tf.keras.applications.InceptionResNetV2(weights='imagenet', include_
```

In [5]:
```
x=base_model.output
```

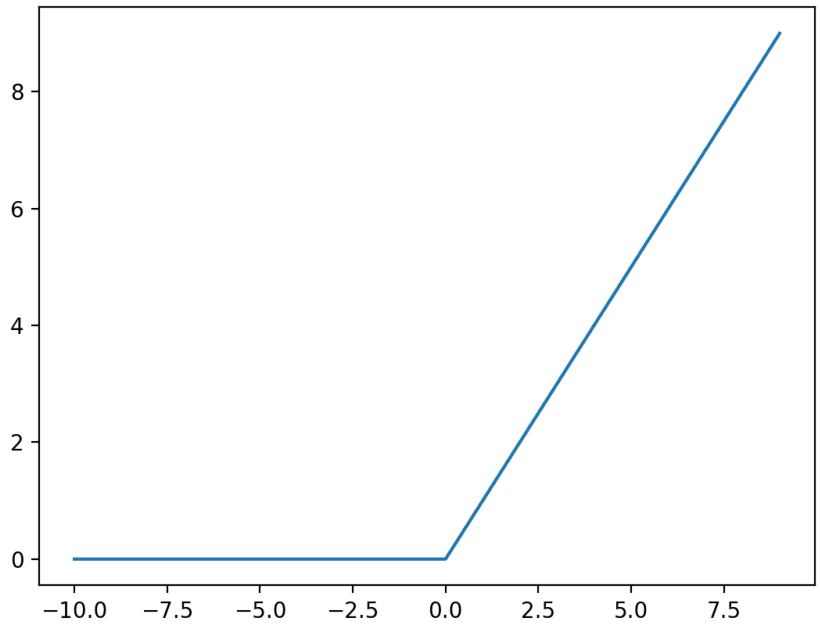
Configuração do modelo "*InceptionResNetV2*":

Nesta etapa é necessário adicionar algumas camadas de nós. Para explicar o que ocorre em cada uma das camadas, é preciso observá-las uma a uma, conforme descrito abaixo:

- *camada GlobalMaxPooling*: Para reduzir a resolução, é utilizada uma operação de *pooling* conhecida como “*pooling máximo*” (ou *max pooling*). Nesta operação de agrupamento, um “bloco” com altura 'A' e largura 'L' desliza sobre os dados de entrada (conforme na figura abaixo, de 'a' até 'd'). A cada iteração (ou seja, o quanto ela avança durante a operação de deslizamento) é muitas vezes igual ao tamanho da *pool*, de modo que seu efeito é igual a uma redução na altura e largura. Para cada bloco, ou “pool”, a operação envolve simplesmente o cálculo do valor máximo. Fazendo isso para cada pool, obtemos um resultado bem reduzido, otimizando muito a quantidade de espaço de que precisamos.



- camada densa com função de ativação "*ReLU*": A função de ativação linear retificada (ou *ReLU*) é uma função linear por partes que produzirá a entrada diretamente se for positiva, caso contrário, ela produzirá zero. Ela se comporta conforme o gráfico abaixo:



  Neste caso, foram adicionadas três camadas densas deste tipo, porém uma com 128 neurônios, outra com 64 neurônios e outra com 32 neurônios.
- *camada dropout*: é feita uma espécide de "regularização", onde alguns neurônios são desligados de forma aleatória, juntamente com suas conexões, apenas durante o período de treinamento, porém durante a predição todos os neurônios são mantidos ativos. O motivo de se fazer isso é evitar *overfitting* no treinamento. A porcentagem escolhida nesse caso foi de 50%, conforme orientação no enunciado do projeto.
- *camada de ativação*: utiliza uma função de ativação ao final das camadas anteriores. Essa função tenta aplicar um comportamento biologicamente análogo às excitações dos neurônios reais, isto é, replicar o comportamento de transmissão de informações dos neurônios, que só conseguem passar adiante a informação após sofrerem um estímulo. O que ocorre na prática é que a função insere um comportamento de "não-linearidade" após a função dos pesos com as entradas. Conforme enunciado, teria de ser feita uma escolha entre a função *sigmóide* e a

função *softmax*. A escolha nesta situação é intuitiva, pois a função sigmóid é mais utilizada no aprendizado de funções lógicas, uma vez que ela tenta encaixar os valores entre 0 e 1. Já a função softmax produz uma distribuição de probabilidades para cada uma das classes das imagens durante a classificação, ao contrário da sigmóid que só consegue lidar com duas classes. Sendo assim, fica definido então que a função a ser utilizada é a *softmax* com 3 classes.

Em seguida é feita a definição do modelo final bem como sua exibição, que pode ser vista na sumário a seguir:
Obs.: em virtude do tamanho do sumário, a linha de código que o exibe foi comentada.

In [6]:
```
x=tf.keras.layers.GlobalMaxPooling2D()(x)

x=tf.keras.layers.Dense(128,activation='relu')(x)

x=tf.keras.layers.Dense(64,activation='relu')(x)

x=tf.keras.layers.Dense(32,activation='relu')(x)

x=tf.keras.layers.Dropout(0.5)(x)

preds=tf.keras.layers.Dense(3,activation='softmax')(x)

model=tf.keras.models.Model(inputs=base_model.input,outputs=preds)

#model.summary()
```

Treinamento e teste do modelo:

Primeiramente, é feito o congelamento dos neurônios já treinados na *ImageNet*, para retreinar somente as camadas densas incluídas no passo anterior. Para fazer isso, é feita a mudança na variável booleana "*treinable*" de cada camada que não inicia com o nome "*dense*" para false.

Em seguida são criados dois objetos:

- *train_data_gen*;
- *test_data_gen*.

Cada um desses objetos irá receber as imagens já processadas com o método da *ResNetV2*, separados em treino e teste conforme a nomenclatura da variável.

Posteriormente, é necessário criar os geradores das imagens (tanto de teste quanto de treino). Isso é feito atravez dos objetos criados anteriormente, através da função "*flow_from_directory*". Seus parâmetros são os que seguem:

- *path*: o caminho do diretório onde estão localizadas as imagens;
- *target_size*: o tamanho da imagem (neste caso, foi escolhido o tamanho 128x128);
- *batch_size*: o mesmo explicado anteriormente, que já foi definido como 32;
- *class_mode*: pode ser "input", caso a imagem de entrada e saída forem as mesmas, "binary" se existirem apenas duas classes para realizar a predição ou "categorical", caso hajam mais classes, que é este caso;

- *shuffle*: *booleano* para ativar ou não o embaralhamento da ordem das imagens que seram utilizadas.

O resultado obtido são dois geradores como saída, armazenados nos objetos:

- *train_generator*;
- *test_generator*.

Com os geradores criados, é preciso definir o otimizador. Conforme enunciado, os otimizadores que apresentam melhores resultados são o *SGD* e o *Adam*. Sendo assim, foi escolhido o *Adam*. Sendo assim, foi compilado o modelo com esses parâmetros, juntamente com a métrica escolhida como sendo por acurácia (*accuracy*), conforme orientação do enunciado.

Por fim foram definidos os steps, e o modelo foi efetivamente treinado e testado. Os resultados podem ser vistos nos próximos tópicos.

In [7]:
```
for l in model.layers:
    if l.name.split('_')[0] != 'dense':
        l.trainable=False
    else:
        l.trainable=True
```

In [8]:
```
train_data_gen = tf.keras.preprocessing.image.ImageDataGenerator(preprocessing_fu

test_data_gen = tf.keras.preprocessing.image.ImageDataGenerator(preprocessing_fur
```

In [9]:
```
train_generator = train_data_gen.flow_from_directory('shapes_split/train',
                                                     target_size=(128, 128), # tamanh
                                                     batch_size=batch,
                                                     class_mode='categorical',
                                                     shuffle=True)

test_generator = test_data_gen.flow_from_directory('shapes_split/test',
                                                     target_size=(128, 128), # tamanh
                                                     batch_size=batch,
                                                     class_mode='categorical',
                                                     shuffle=True)
```

```
Found 240 images belonging to 3 classes.
Found 60 images belonging to 3 classes.
```

In [10]:
```
lr = tf.keras.optimizers.Adam(learning_rate=0.001)

model.compile(optimizer=lr, loss='categorical_crossentropy', metrics=['accuracy']
```

In [11]:
```
step_size_train = train_generator.n//train_generator.batch_size
step_size_test = test_generator.n//test_generator.batch_size
```

In [12]:
```
# history = model.fit_generator(generator=train_generator,
#                     steps_per_epoch=step_size_train,
#                     epochs=epochs,
#                     validation_data=test_generator,
#                     validation_steps=step_size_test)
history = model.fit(train_generator,
                    steps_per_epoch=step_size_train,
                    epochs=epochs,
                    validation_data=test_generator,
                    validation_steps=step_size_test)
```

```
Epoch 1/500
7/7 [==============================] - 25s 2s/step - loss: 1.7027 - accuracy:
0.4486 - val_loss: 0.1233 - val_accuracy: 1.0000
Epoch 2/500
7/7 [==============================] - 6s 930ms/step - loss: 0.3485 - accurac
y: 0.8848 - val_loss: 0.0389 - val_accuracy: 1.0000
Epoch 3/500
7/7 [==============================] - 7s 1s/step - loss: 0.2486 - accuracy:
0.9267 - val_loss: 0.0364 - val_accuracy: 0.9688
Epoch 4/500
7/7 [==============================] - 7s 1s/step - loss: 0.1763 - accuracy:
0.9271 - val_loss: 0.0058 - val_accuracy: 1.0000
Epoch 5/500
7/7 [==============================] - 6s 995ms/step - loss: 0.0697 - accurac
y: 0.9733 - val_loss: 0.0075 - val_accuracy: 1.0000
Epoch 6/500
7/7 [==============================] - 7s 1s/step - loss: 0.0603 - accuracy:
0.9798 - val_loss: 0.0059 - val_accuracy: 1.0000
Epoch 7/500
```

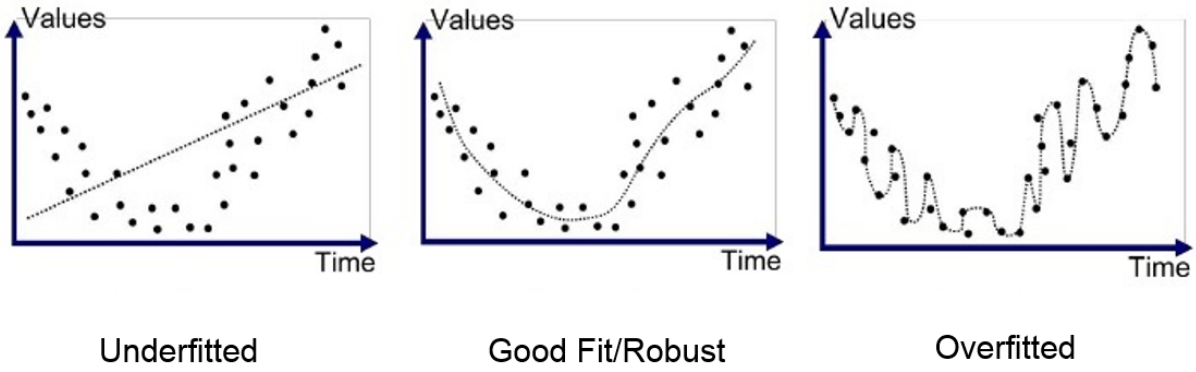
Resultados:

Primeiramente, é feito o cálculo da acurácia do treino e do teste, que podem ser vistas abaixo.

In [29]:
```
loss_train, train_acc = model.evaluate(train_generator, steps=step_size_train)
loss_test, test_acc = model.evaluate_generator(test_generator, steps=step_size_te
print('Acurácia de treino: %.3f ' % (train_acc))
print('Acurácia de teste: %.3f ' % (test_acc))
```

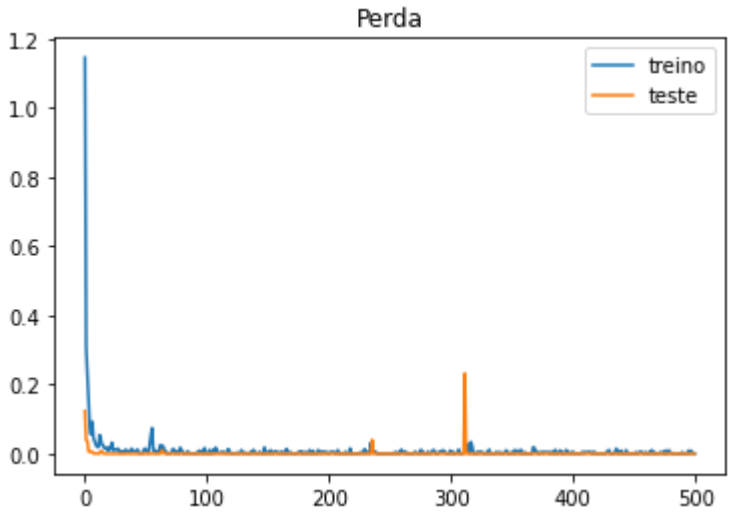
```
7/7 [==============================] - 6s 868ms/step - loss: 0.0000e+00 - accur
acy: 1.0000
Acurácia de treino: 1.000
Acurácia de teste: 1.000
```

Observando os números acima para a acurácia, podemos ver que o resultado foi um número muito atípico, no caso *1.000*, que representa 100% de acurácia no treino e no teste. Quando isso acontece, geralmente se dá o nome de *overfit* (ou *overfitting*), que é quando um modelo estatístico se ajusta bem demais ao conjunto de dados observado, mas se mostra ineficaz para prever novos resultados. Os gráficos abaixo ilustram bem esse comportamento:



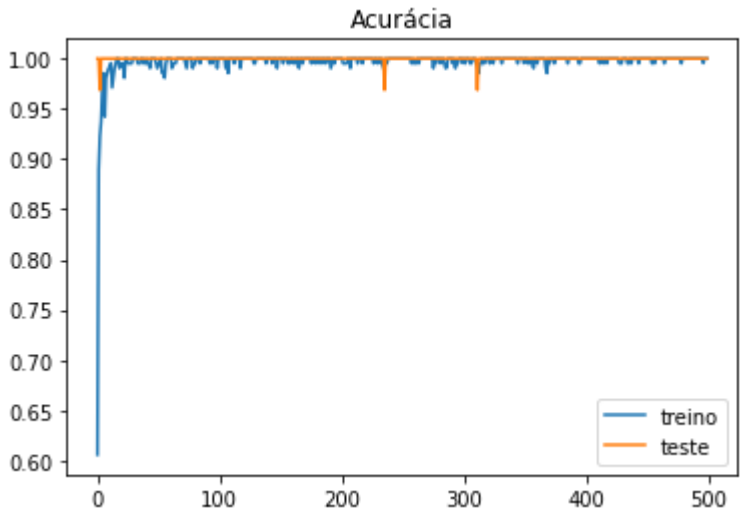
O gráfico seguinte apresenta a taxa de perda de treino e de teste:

In [14]:
```
plt.title('Perda')
plt.plot(history.history['loss'], label='treino')
plt.plot(history.history['val_loss'], label='teste')
plt.legend()
plt.show()
```



O gráfico seguinte apresenta a acurácia de treino e de teste:

In [15]:
```
plt.title('Acurácia')
plt.plot(history.history['accuracy'], label='treino')
plt.plot(history.history['val_accuracy'], label='teste')
plt.legend()
plt.show()
```



Por fim, são feitas as classificações finais, onde são criadas as estruturas para as métricas de avaliação.

Classification Report

- TN / True Negative: o caso foi negativo e previsto como negativo
- TP / True Positive: o caso foi positivo e positivo previsto
- FN / False Negative: o caso era positivo, mas previsto como negativo
- FP / False Positive: o caso foi negativo, mas previsto como positivo

Precisão - Qual porcentagem de suas previsões estava correta?

É a capacidade de um classificador de não marcar uma instância como positiva (na verdade, negativa). Para cada categoria, é definido como a proporção de verdadeiros positivos para a soma de verdadeiros positivos e falsos positivos.

Formula (Precisão de previsões positivas): *Precision = TP/(TP + FP)*

Recall - Qual a porcentagem de casos positivos que você pegou?

É a capacidade do classificador de encontrar todas as instâncias positivas. Para cada categoria, é definido como a proporção de verdadeiros positivos para a soma de verdadeiros positivos e falsos negativos.

Fórmula (Fração de positivos que foram identificados corretamente): *Recall = TP/(TP+FN)*

F1 Score - Qual porcentagem de previsões positivas estavam corretas?

É a média harmônica ponderada de acurácia e recordação, portanto, a pontuação mais alta é 1,0 e a pior diferença é 0,0. As pontuações F1 são mais baixas do que as medições de precisão porque incorporam precisão e recall no cálculo. Geralmente, a média ponderada F1 deve ser usada para comparar os modelos do classificador, ao invés da precisão geral.

Fórmula: *F1 Score = 2*(Recall * Precision) / (Recall + Precision)*

Support

É o número real de ocorrências da classe no conjunto de dados especificado. O suporte desequilibrado nos dados de treinamento pode indicar fraquezas estruturais nas pontuações do classificador relatadas e pode indicar a necessidade de amostragem estratificada ou rebalanceamento. O suporte não mudará entre os modelos, mas sim um processo de avaliação diagnóstica.

In [32]:
```
labels = os.listdir('shapes_split/test')
print('Rótulos', labels)
#criando estruturas para métricas de avaliação, processo um pouco mais demorado
Y_pred = model.predict(test_generator)
print('Preds Created')
y_pred = np.argmax(Y_pred, axis=1)
print('Preds 1D created')
```

```
Rótulos ['circles', 'squares', 'triangles']
Preds Created
Preds 1D created
```

In [48]:
```
classification = classification_report(test_generator.classes, y_pred, target_nam
print('----------------CLASSIFICATION--------------')
print(classification)
matrix = confusion_matrix(test_generator.classes, y_pred)
df_cm = pd.DataFrame(matrix, index = [i for i in range(3)],
                  columns = [i for i in range(3)])
plt.figure(figsize = (10,7))
print('----------------MATRIX--------------')
sn.heatmap(df_cm, annot=True, linewidths=2.5, xticklabels=labels, yticklabels=lab
```

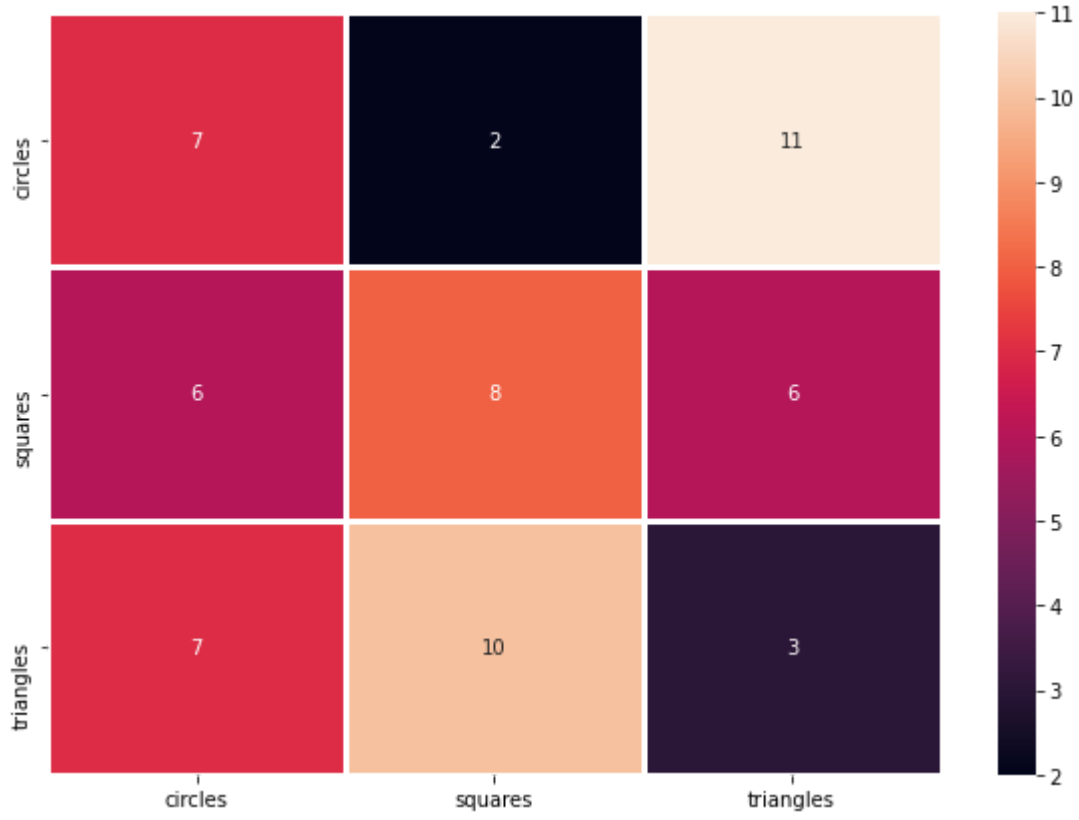
```
----------------CLASSIFICATION--------------
              precision    recall  f1-score   support

     circles       0.35      0.35      0.35        20
     squares       0.40      0.40      0.40        20
   triangles       0.15      0.15      0.15        20

    accuracy                           0.30        60
   macro avg       0.30      0.30      0.30        60
weighted avg       0.30      0.30      0.30        60

----------------MATRIX--------------
```

Out[48]: <AxesSubplot:>



Referências bibliográficas:

Mini Curso CNN Transfer Learning, William Sdayle Marins Silva. Disponível em: https://colab.research.google.com/drive/11akI_B5M0Y2cuO1Y5Qq7W36YuehkzvKh?usp=sharing. Acesso em 20 de Abril de 2021.

A Gentle Introduction to the Rectified Linear Unit (ReLU), Jason Brownlee. Disponível em: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/. Acesso em 20 de Abril de 2021.

Arquiteturas de Redes Neurais Convolucionais para reconhecimento de imagens, Alexandre Luiz Bianchi. Disponível em: https://www.viceri.com.br/insights/arquiteturas-de-redes-neurais-convolucionais-para-reconhecimento-de-imagens/. Acesso em 20 de Abril de 2021.

Overfitting e underfitting em Machine Learning, Henrique Branco. Disponível em: https://abracd.org/overfitting-e-underfitting-em-machine-learning/. Acesso em 20 de Abril de 2021.