T1_RNA_convolucional

November 22, 2020

Trabalho #1 - RNA convolucional

Nesse trabalho você vai desenvolver uma rede neural convolucional deep-learning usando a plataforma TensorFlow-Keras, para realizar uma tarefa de classificação de múltiplas classes, que consiste na identificação de sinais de mão a partir de imagens.

Coloque o seu nome aqui

Aluno: Bruno Rodrigues Silva

Em primeiro lugar é necessário importar alguns pacotes do Python que serão usados nesse trabalho: - numpy pacote de cálculo científico com Python - matplotlib biblioteca para gerar gráficos em Python - utils.py função para ler banco de dados

[ ]:
```
import numpy as np
import h5py
import matplotlib.pyplot as plt

%matplotlib inline
```

1 Visão geral do problema

Definição do problema:

O objetivo desse problema é desenvolver uma RNA que recebe como entrada uma imagem de sinais de mão, avalia a probabilidade dos dedos da mão mostrarem um número de 0 a 5 e determina qual o número mais provável entre os seis possíveis.

O banco de dados usado nesse trabalho é SIGNS, que consiste de imagens de sinais de mão desenvolvido por Andre Ng. Esse banco de dados pode ser obtido no link: https://github.com/cs230-stanford/cs230-code-examples/tree/master/tensorflow/vision

O banco de dados possui 1080 exemplos de treinamento e 120 exemplos de teste. Cada exemplo consiste de uma imagem colorida associada a um rótulo de 6 classes. A Figura abaixo mostra alguns exemplos dessas imagens.

1.1 Dados de treinamento

Os dados que iremos utilizar nesse trabalho estão nos arquivos train_signs.h5 e test_signs.h5.

Características dos dados:

- As imagens são coloridas e estão no padrão RGB;
- Cada imagem tem dimensão de 64x64x3;
- O valor da intensidade luminosa de cada plano de cor é um número inteiro entre 0 e 255;
- As saídas representam o rótulo do sinal de mão mostrado na imagem, send um número inteiro de 0 a 5.

1.2 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 dos arquivos de dados

train_dataset = h5py.File('train_signs.h5', "r")
X_train_orig = np.array(train_dataset["train_set_x"][:]) # your train set␣
 ↪features
Y_train_orig = np.array(train_dataset["train_set_y"][:]) # your train set labels

test_dataset = h5py.File('test_signs.h5', "r")
X_test_orig = np.array(test_dataset["test_set_x"][:]) # your test set features
Y_test_orig = np.array(test_dataset["test_set_y"][:]) # your test set labels

classes = np.array(test_dataset["list_classes"][:]) # the list of classes

Y_train_orig = Y_train_orig.reshape((Y_train_orig.shape[0], 1))
Y_test_orig = Y_test_orig.reshape((Y_test_orig.shape[0], 1))

print("X_train shape:", X_train_orig.shape, "y_train shape:", Y_train_orig.shape)
print("X_test shape:", X_test_orig.shape, "y_test shape:", Y_test_orig.shape)
```

```
X_train shape: (1080, 64, 64, 3) y_train shape: (1080, 1)
X_test shape: (120, 64, 64, 3) y_test shape: (120, 1)
```

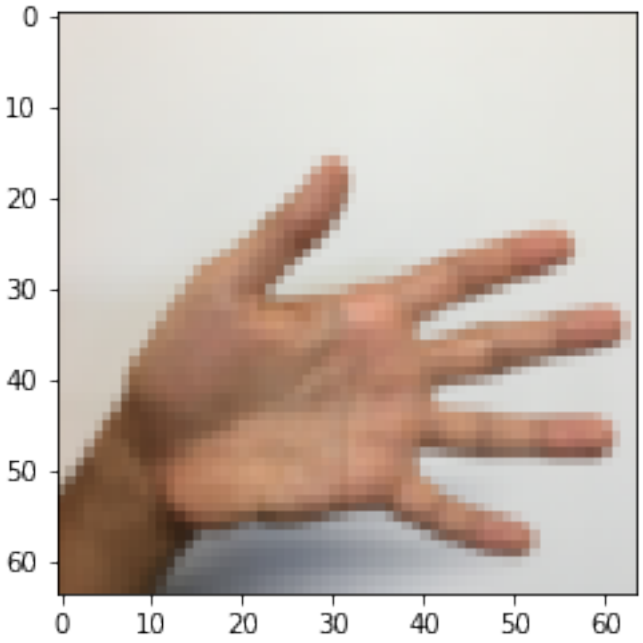
Pela dimensão dos tensores com os dados de treinamento e teste temos:

- 1080 imagens de treinamento com dimensão de 64x64x3 pixels;
- 120 imagens de teste com dimensão de 64x64x3 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 = 5
```



1.3 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.

Divisão do conjunto de dados de treinamento

Execute a célula abaixo para para dividir o conjunto de dados de treinamento nos conjuntos de treinamento e validação e redimensionar as saídas para que o primeiro eixo seja o dos exemplos e o segundo eixo o das classes.

[ ]:
```
# Dados de entrada
X_train_int = X_train_orig[:960,:]
X_val_int = X_train_orig[960:,:]

# Dados de saída
Y_train = Y_train_orig[:960]
Y_val = Y_train_orig[960:]
Y_test = Y_test_orig
```

```
print("Dimensão do tensor de dados de entrada de treinamento =", X_train_int.
 ↪shape)
print("Dimensão do tensor de dados de entrada de validação =", X_val_int.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 test =", Y_test.shape)
```

```
Dimensão do tensor de dados de entrada de treinamento = (960, 64, 64, 3)
Dimensão do tensor de dados de entrada de validação = (120, 64, 64, 3)
Dimensão do tensor de dados de saída de treinamento = (960, 1)
Dimensão do tensor de dados de saída de validação = (120, 1)
Dimensão do tensor de dados de saída de test = (120, 1)
```

Normalização dos dados de entrada

Execute a célula abaixo para normalizar e transformar as imagens em números reais dividindo por 255.

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

# Transformação dos dados em números reais
X_train = X_train_int.astype('float32') / 255
X_val = X_val_int.astype('float32')/255
X_test = X_test_orig.astype('float32') / 255

# Para verificar se os resultados estão corretos
print("Primeiros elementos da primeira linha da primeira imagem de treinamento =␣
 ↪", X_train[0,0,0:4,1])
print("Primeiros elementos da primeira linha da primeira imagem de validação =␣
 ↪", X_val[0,0,0:4,1])
print("Primeiros elementos da primeira linha da primeira imagem de teste = ",␣
 ↪X_test[0,0,0:4,1])
```

```
Dimensão das imagens de entrada= (64, 64, 3)
Primeiros elementos da primeira linha da primeira imagem de treinamento =
[0.8627451  0.8666667  0.87058824 0.8666667 ]
Primeiros elementos da primeira linha da primeira imagem de validação =
[0.85882354 0.85882354 0.85882354 0.85882354]
Primeiros elementos da primeira linha da primeira imagem de teste =  [0.8784314
0.8784314  0.88235295 0.88235295]
```

Codificação das classes

As classes dos sinais são identificadas por um número inteiro que varia de 0 a 5. 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 6 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 6 elementos, com todos os elementos iguais a zero a menos do correspondente ao da classe do sinal, que deve ser igual a um. A função que realiza essa transformação é conhecida na literatura de "one-hot-encoding", que no Keras é chamada de "to_categorical". Execute a célula abaixo para transformar os dados de saída usando a função "to_categorical" do keras.

[ ]:
```
# Importa classe de utilidades do Keras
from tensorflow.keras.utils import to_categorical

# Transformação das classes de números reais para vetores
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:  (960, 6)
Dimensão dos dados de saída do conjunto de validação:  (120, 6)
Dimensão dos dados de saída do conjunto de teste:  (120, 6)
```

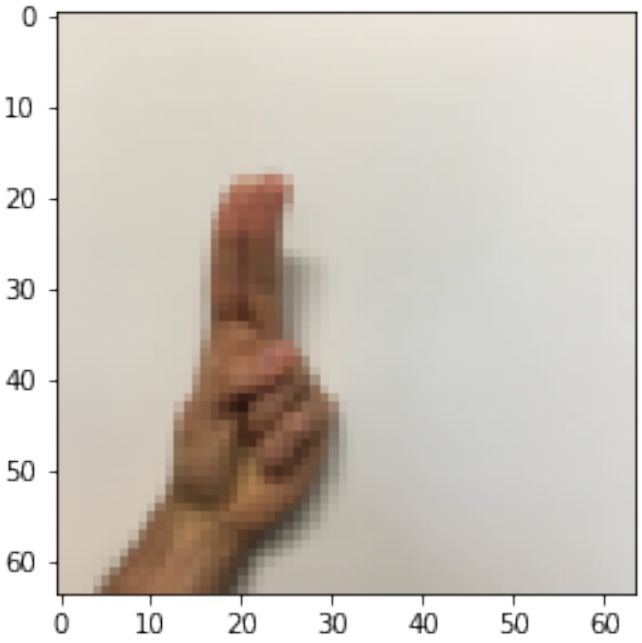
Visualização da entrada e saída correspondente

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 = 10
print("Classe numérica: ", Y_train[index], ", Vetor de saída correspondentes: ",␣
 ↪Y_train_hot[index])
plt.imshow(X_train_orig[index], cmap='gray', vmin=0, vmax=255)
```

```
Classe numérica:  [2] , Vetor de saída correspondentes:  [0. 0. 1. 0. 0. 0.]
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f38e5b64320>
```



2 RNA convolucional

Nesse trabalho você irá usar uma RNA convolucional e, assim, poderá verificar que uma RNA convolucional é mais eficiente para processar imagens do que uma RNA com camadas somente densas, como foi feito no Trabalho #5.

2.1 Exercício #1: criação da RNA

Você vai usar uma RNA com 3 camadas convolucionais, seguidas de camadas "max-pooling", e 3 camadas densas, com as seguintes características:

- Primeira camada convolucional: número de filtros n1, dimensão do filtro 3, "padding valid", "stride" 1, função de ativação ReLu;
- Segunda camada convolucional: número de filtros n2, dimensão do filtro 3, "padding valid", "stride" 1, função de ativação ReLu;
- Terceira camada convolucional: número de filtros n3, dimensão do filtro 3, "padding valid", "stride" 1, função de ativação ReLu;
- Camadas de max-pooling: dimensão da janela 2, "stride" 2;
- Primeira camada densa: número de neurônios n4, função de ativação ReLu;
- Segunda camada densa: número de neurônios n5, função de ativação ReLu;
- Camada de saída: número de neurônio n6, função de ativação softmax.

Ressalta-se que após cada camada convolucional tem uma camada de max-pooling.

Na célula abaixo crie uma função que recebe a dimensão dos dados de entrada e os números de neurônios das camadas e configura a RNA de acordo com as características acima. Não se esqueça

de incluir a camada de "flattening" entre a última camada de max-pooling e a primeira camada densa.

[ ]:
```
# PARA VOCÊ FAZER: função para configuração da RNA
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPool2D, Flatten

def build_model(n1, n2, n3, n4, n5, n6):
    model = Sequential([
                    Conv2D(n1, (3,3), strides=(1, 1), padding='valid',␣
 ↪activation='relu', input_shape=(64, 64, 3)),
                    MaxPool2D((2,2), 2),
                    Conv2D(n2, (3,3), strides=(1, 1),␣
 ↪padding='valid',activation='relu'),
                    MaxPool2D((2,2), 2),
                    Conv2D(n3, (3,3), strides=(1, 1), padding='valid',␣
 ↪activation='relu'),
                    MaxPool2D((2,2), 2),
                    Flatten(),
                    Dense(n4, 'relu'),
                    Dense(n5, 'relu'),
                    Dense(n6, 'softmax')
    ])
    return model
```

Defina os números de neurônios das camadas convolucionais, das camadas densas e da camada de saída e crie a RNA usando a função `build_model` criada na célula anterior. Utilize n1 = 8, n2 = 16, n3 = 32, n4 = 64, n5 = 32, n6 = 6. Após criar a RNA utilize o método summary para visualizar a sua rede.

[ ]:
```
# PARA VOCÊ FAZER: criação da RNA
(n1, n2, n3, n4, n5, n6) = (8, 16, 32, 64, 32, 6)
rna = build_model(n1, n2, n3, n4, n5, n6)
rna.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_9 (Conv2D)            (None, 62, 62, 8)         224
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 31, 31, 8)         0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 29, 29, 16)        1168
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 14, 14, 16)        0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 12, 12, 32)        4640
```

```
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 6, 6, 32)          0
_________________________________________________________________
flatten_3 (Flatten)          (None, 1152)              0
_________________________________________________________________
dense_9 (Dense)              (None, 64)                73792
_________________________________________________________________
dense_10 (Dense)             (None, 32)                2080
_________________________________________________________________
dense_11 (Dense)             (None, 6)                 198
=================================================================
Total params: 82,102
Trainable params: 82,102
Non-trainable params: 0
_________________________________________________________________
```

Saída esperada:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 62, 62, 8)         224
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 31, 31, 8)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 29, 29, 16)        1168
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 12, 12, 32)        4640
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 32)          0
_________________________________________________________________
flatten (Flatten)            (None, 1152)              0
_________________________________________________________________
dense (Dense)                (None, 64)                73792
_________________________________________________________________
dense_1 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_2 (Dense)              (None, 6)                 198
=================================================================
Total params: 82,102
Trainable params: 82,102
Non-trainable params: 0
_________________________________________________________________
```

2.2 Exercício #2: Número de parâmetros da RNA

Calcule o número de parâmetros da sua RNA. Escreva as contas realizadas e os seus resultados a seguir:

- Número de parâmetros da camada convulucional 1 = $Kernel^2 * Channels * Filters + Filters = 3^2 * 3 * 8 + 8 = 224$
- Número de parâmetros da camada convulucional 2 = $3^2 * 8 * 16 + 16 = 1168$
- Número de parâmetros da camada convulucional 3 = $3^2 * 16 * 32 + 32 = 4640$
- Número de parâmetros da camada densa 1 = $(1152 + 1) * 64 = 73792$
- Número de parâmetros da camada densa 2 = $(64 + 1) * 32 = 2080$
- Número de parâmetros da camada de saída = $(32 + 1) * 6 = 198$

2.3 Exercício #3: 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;
- beta1 = 0.9;
- beta2 = 0.999;
- decay = 0;
- número de épocas = 40.

[ ]:
```
# PARA VOCÊ FAZER: compilação e treinamento da RNA usando o método do gradiente␣
 ↪descendente com momento
from tensorflow.keras.optimizers import Adam
opt = Adam(lr=0.001, decay=0)
rna.compile(opt, loss='categorical_crossentropy', metrics=['accuracy'])
history = rna.fit(X_train, Y_train_hot, batch_size=1, epochs=40,␣
 ↪validation_data=(X_val, Y_val_hot))
```

```
Epoch 1/40
960/960 [==============================] - 5s 5ms/step - loss: 1.7835 -
accuracy: 0.2146 - val_loss: 1.7209 - val_accuracy: 0.2750
Epoch 2/40
960/960 [==============================] - 5s 5ms/step - loss: 1.2491 -
accuracy: 0.4896 - val_loss: 1.0835 - val_accuracy: 0.6167
Epoch 3/40
960/960 [==============================] - 5s 5ms/step - loss: 0.8487 -
accuracy: 0.6792 - val_loss: 0.7887 - val_accuracy: 0.6750
Epoch 4/40
960/960 [==============================] - 4s 5ms/step - loss: 0.5788 -
accuracy: 0.7844 - val_loss: 0.6205 - val_accuracy: 0.8083
Epoch 5/40
960/960 [==============================] - 5s 5ms/step - loss: 0.3925 -
accuracy: 0.8594 - val_loss: 0.4867 - val_accuracy: 0.8500
Epoch 6/40
```

```
960/960 [==============================] - 5s 5ms/step - loss: 0.2792 -
accuracy: 0.9073 - val_loss: 0.4933 - val_accuracy: 0.8417
Epoch 7/40
960/960 [==============================] - 5s 5ms/step - loss: 0.1734 -
accuracy: 0.9365 - val_loss: 0.8845 - val_accuracy: 0.7250
Epoch 8/40
960/960 [==============================] - 5s 5ms/step - loss: 0.2116 -
accuracy: 0.9250 - val_loss: 0.4396 - val_accuracy: 0.8333
Epoch 9/40
960/960 [==============================] - 5s 5ms/step - loss: 0.1217 -
accuracy: 0.9667 - val_loss: 0.5175 - val_accuracy: 0.8333
Epoch 10/40
960/960 [==============================] - 5s 5ms/step - loss: 0.1047 -
accuracy: 0.9635 - val_loss: 0.4182 - val_accuracy: 0.8583
Epoch 11/40
960/960 [==============================] - 5s 5ms/step - loss: 0.1030 -
accuracy: 0.9688 - val_loss: 0.4400 - val_accuracy: 0.8583
Epoch 12/40
960/960 [==============================] - 5s 5ms/step - loss: 0.0674 -
accuracy: 0.9760 - val_loss: 0.6066 - val_accuracy: 0.8500
Epoch 13/40
960/960 [==============================] - 5s 5ms/step - loss: 0.0386 -
accuracy: 0.9906 - val_loss: 0.8326 - val_accuracy: 0.8333
Epoch 14/40
960/960 [==============================] - 5s 5ms/step - loss: 0.0958 -
accuracy: 0.9667 - val_loss: 1.5472 - val_accuracy: 0.7250
Epoch 15/40
960/960 [==============================] - 5s 5ms/step - loss: 0.0677 -
accuracy: 0.9792 - val_loss: 0.5040 - val_accuracy: 0.8917
Epoch 16/40
960/960 [==============================] - 5s 5ms/step - loss: 0.0067 -
accuracy: 0.9990 - val_loss: 0.7624 - val_accuracy: 0.8333
Epoch 17/40
960/960 [==============================] - 5s 5ms/step - loss: 0.1058 -
accuracy: 0.9667 - val_loss: 0.8654 - val_accuracy: 0.8083
Epoch 18/40
960/960 [==============================] - 5s 5ms/step - loss: 0.0548 -
accuracy: 0.9844 - val_loss: 0.8439 - val_accuracy: 0.8417
Epoch 19/40
960/960 [==============================] - 5s 5ms/step - loss: 0.0280 -
accuracy: 0.9906 - val_loss: 0.6246 - val_accuracy: 0.8667
Epoch 20/40
960/960 [==============================] - 5s 5ms/step - loss: 0.0129 -
accuracy: 0.9948 - val_loss: 0.6130 - val_accuracy: 0.8583
Epoch 21/40
960/960 [==============================] - 5s 5ms/step - loss: 0.0987 -
accuracy: 0.9760 - val_loss: 0.6394 - val_accuracy: 0.8583
Epoch 22/40
```

```
960/960 [==============================] - 5s 5ms/step - loss: 0.0130 -
accuracy: 0.9979 - val_loss: 1.0891 - val_accuracy: 0.8167
Epoch 23/40
960/960 [==============================] - 5s 5ms/step - loss: 0.0446 -
accuracy: 0.9885 - val_loss: 1.0986 - val_accuracy: 0.7917
Epoch 24/40
960/960 [==============================] - 5s 5ms/step - loss: 0.0858 -
accuracy: 0.9708 - val_loss: 0.6140 - val_accuracy: 0.8333
Epoch 25/40
960/960 [==============================] - 5s 5ms/step - loss: 0.0030 -
accuracy: 1.0000 - val_loss: 0.5286 - val_accuracy: 0.8667
Epoch 26/40
960/960 [==============================] - 5s 5ms/step - loss: 0.0426 -
accuracy: 0.9906 - val_loss: 0.6638 - val_accuracy: 0.8167
Epoch 27/40
960/960 [==============================] - 4s 5ms/step - loss: 0.0477 -
accuracy: 0.9823 - val_loss: 0.8359 - val_accuracy: 0.8583
Epoch 28/40
960/960 [==============================] - 5s 5ms/step - loss: 0.0813 -
accuracy: 0.9781 - val_loss: 0.9281 - val_accuracy: 0.7917
Epoch 29/40
960/960 [==============================] - 5s 5ms/step - loss: 0.0081 -
accuracy: 0.9969 - val_loss: 0.8028 - val_accuracy: 0.8500
Epoch 30/40
960/960 [==============================] - 5s 5ms/step - loss: 5.6769e-04 -
accuracy: 1.0000 - val_loss: 0.7953 - val_accuracy: 0.8500
Epoch 31/40
960/960 [==============================] - 5s 5ms/step - loss: 2.6667e-04 -
accuracy: 1.0000 - val_loss: 0.8304 - val_accuracy: 0.8583
Epoch 32/40
960/960 [==============================] - 5s 5ms/step - loss: 1.6078e-04 -
accuracy: 1.0000 - val_loss: 0.8274 - val_accuracy: 0.8667
Epoch 33/40
960/960 [==============================] - 5s 5ms/step - loss: 8.5282e-05 -
accuracy: 1.0000 - val_loss: 0.8168 - val_accuracy: 0.8583
Epoch 34/40
960/960 [==============================] - 5s 5ms/step - loss: 4.5305e-05 -
accuracy: 1.0000 - val_loss: 0.8532 - val_accuracy: 0.8583
Epoch 35/40
960/960 [==============================] - 5s 5ms/step - loss: 2.1277e-05 -
accuracy: 1.0000 - val_loss: 0.8874 - val_accuracy: 0.8583
Epoch 36/40
960/960 [==============================] - 5s 5ms/step - loss: 1.1158e-05 -
accuracy: 1.0000 - val_loss: 0.9063 - val_accuracy: 0.8500
Epoch 37/40
960/960 [==============================] - 5s 5ms/step - loss: 5.8305e-06 -
accuracy: 1.0000 - val_loss: 0.9126 - val_accuracy: 0.8667
Epoch 38/40
```

```
960/960 [==============================] - 5s 5ms/step - loss: 3.5678e-06 -
accuracy: 1.0000 - val_loss: 0.9290 - val_accuracy: 0.8750
Epoch 39/40
960/960 [==============================] - 5s 5ms/step - loss: 2.0130e-06 -
accuracy: 1.0000 - val_loss: 0.9452 - val_accuracy: 0.8750
Epoch 40/40
960/960 [==============================] - 5s 5ms/step - loss: 1.2941e-06 -
accuracy: 1.0000 - val_loss: 0.9742 - val_accuracy: 0.8750
```

Saída esperada:

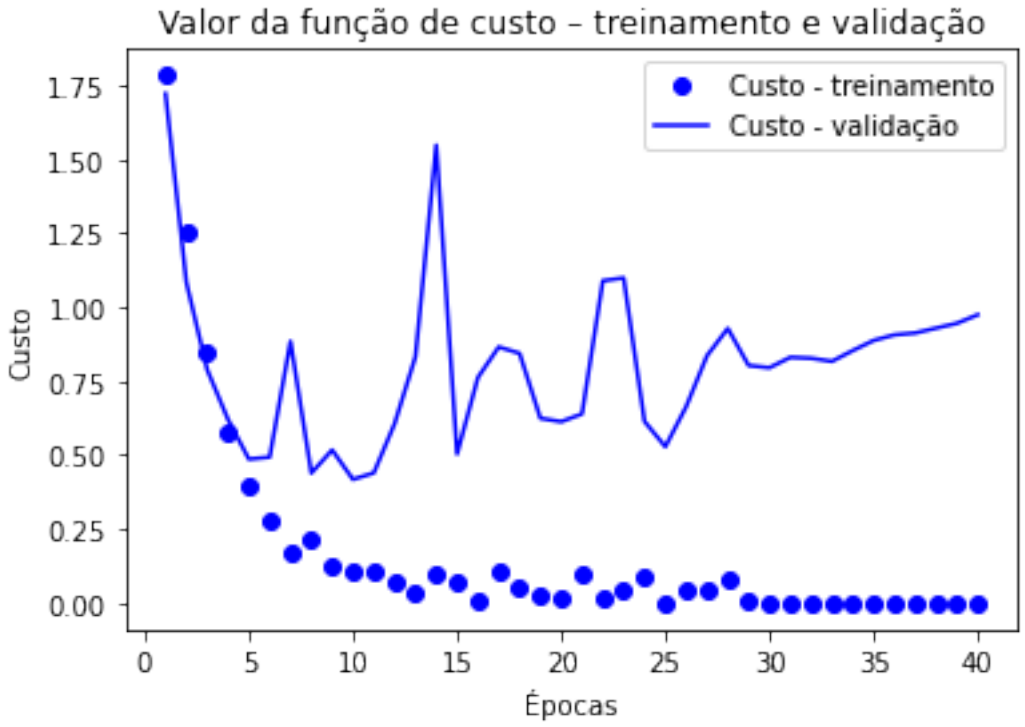
```
Train on 960 samples, validate on 120 samples
Epoch 1/40
960/960 [==============================] - 2s 2ms/sample - loss: 1.7913 - accuracy: 0.1594 - val
.
.
.
Epoch 40/40
960/960 [==============================] - 0s 479us/sample - loss: 0.0028 - accuracy: 1.0000 - v
```

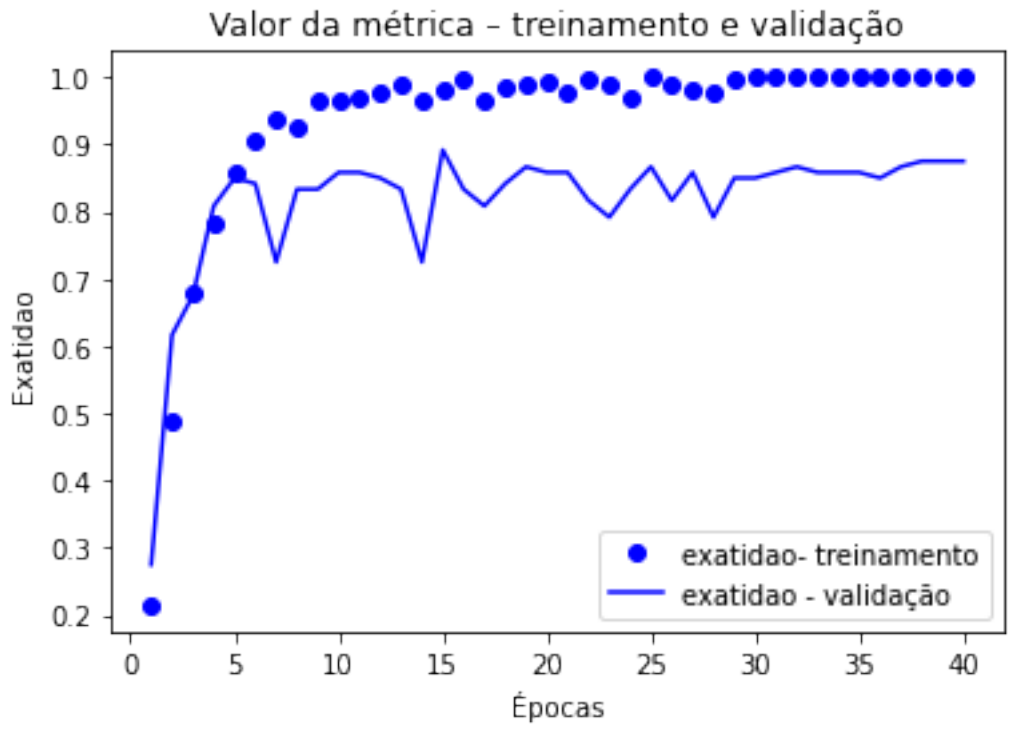
2.4 Visualização dos resultados

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

[ ]:
```
# 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()
```





2.5 Análise dos resultados

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

- O treinamento é bem rápido, sendo que em somente 40 épocas obtém-se uma extaidão de 100% para os dados de treinamento.
- O valor do custo para os dados de treinamento diminui constantemente ao longo do treinamento e a exatidão aumenta constantemente.
- O valor do custo para os dados de validação diminuem até a época 20 e depois estabiliza.
- A exatidão para os dados de validação aumenta constantemente ao longo do treinamento, mas menos do que para os dados de treinamento.
- A exatidão obtida para os dados de validação é de cerca de 92,5%, o que pode ser considerado um resultado muito bom.

2.6 Exercício #4: Avaliação do desempenho da RNA

Na célula abaixo, usando o método `evaluate`, verifique o desempenho da RNA calculando o valor do custo e da métrica para os dados de treinamento, validação e teste.

[ ]:
```
# PARA VOCÊ FAZER: cálculo do custo e exatidão para os dados de treinamento,␣
↪validação e teste
train_eval = rna.evaluate(X_train, Y_train_hot, 1)
val_eval = rna.evaluate(X_val, Y_val_hot, 1)
test_eval = rna.evaluate(X_test, Y_test_hot, 1)
```

```
print(train_eval)
print(val_eval)
print(test_eval)
```

```
960/960 [==============================] - 2s 2ms/step - loss: 7.9012e-07 -
accuracy: 1.0000
120/120 [==============================] - 0s 2ms/step - loss: 0.9742 -
accuracy: 0.8750
120/120 [==============================] - 0s 2ms/step - loss: 0.5450 -
accuracy: 0.9083
[7.901214758021524e-07, 1.0]
[0.9741779565811157, 0.875]
[0.5450438261032104, 0.9083333611488342]
```

Saída esperada:

```
960/960 [==============================] - 0s 215us/sample - loss: 0.0022 - accuracy: 1.0000
120/120 [==============================] - 0s 241us/sample - loss: 0.2762 - accuracy: 0.9250
120/120 [==============================] - 0s 249us/sample - loss: 0.2151 - accuracy: 0.9417
[0.0021954899944830685, 1.0]
[0.2761955052614212, 0.925]
[0.2150653511285782, 0.94166666]
```

2.7 Exercício #5: Verificação dos resultados

Na célula abaixo calcule a previsões da sua RNA para as imagens do conjunto de teste e depois verifique se algumas dessas previsões estão corretas.

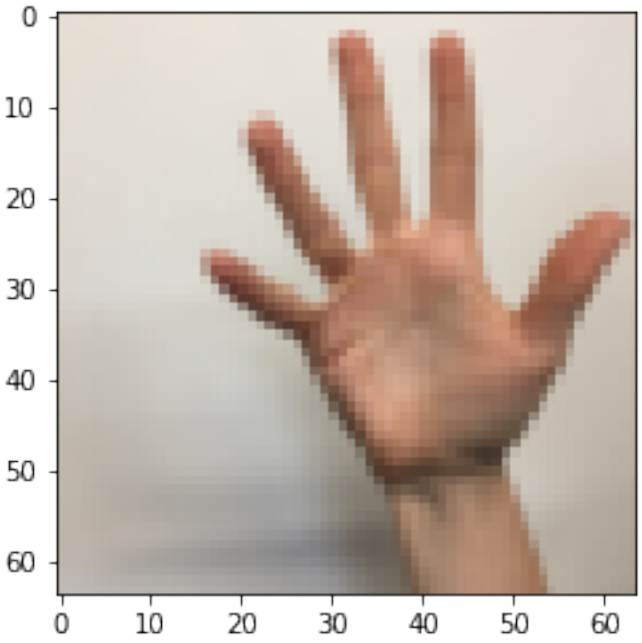
Note que a previsão da RNA é um vetor de 6 elementos com as probabilidades da imagem mostrar os seis sinais. Para detereminar a classe prevista deve-se transformar esse vetor em um número inteiro de 0 a 5, que representa o sinal sendo mostrado. Para fazer essa transformação use a função numpy `argmax(Y_test, axis=?)`, onde `Y_test` é o tensor com as saídas previstas pela RNA. Em qual eixo você deve calcular o índice da maior probabilidade?

Troque a variável index (variando entre 0 e 119) para verificar se a sua RNA consegue classificar corretamente o sinal de mão mostrado nas imagens.

[ ]:
```
# PARA VOCÊ FAZER: cálculo das classes previstas
index = 89
y_pred = np.argmax(rna.predict(X_test), axis=1)
print("Classe esperada:", Y_test[index][0], "\nClasse predita: ", y_pred[index])
plt.imshow(X_test_orig[index], cmap='gray', vmin=0, vmax=255)
```

```
Classe esperada: 5
Classe predita:  5
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f38d79de7b8>
```

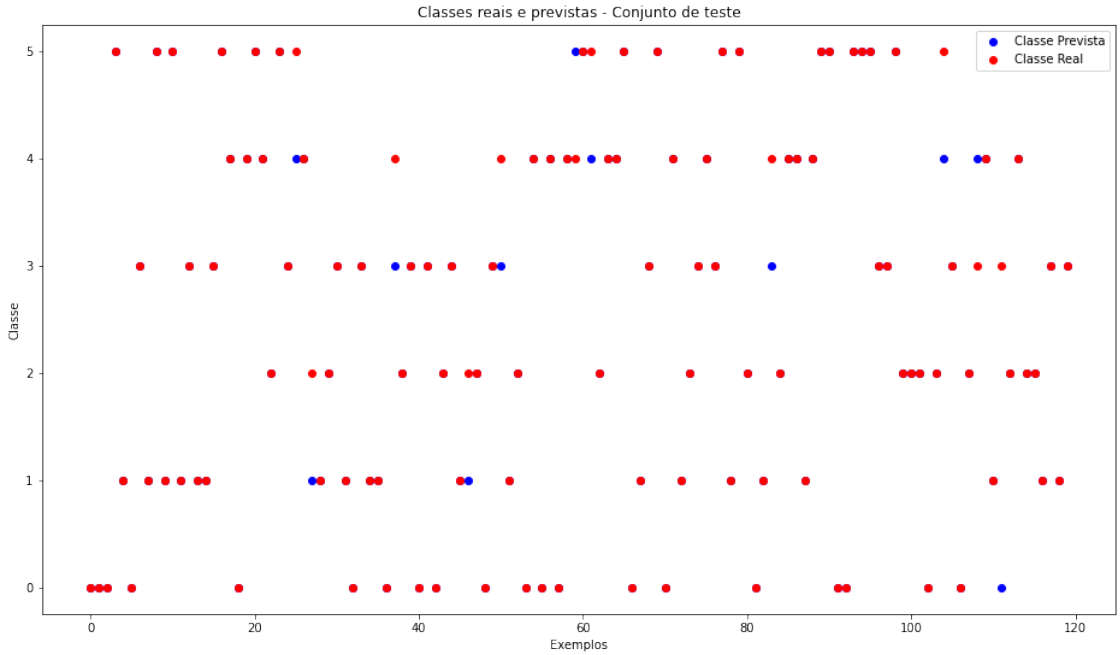


2.8 Exercício #6: Visualização dos resultados

Na célula abaixo crie um código para fazer um gráfico com as classes reais e as previstas pela sua RNA para todos os exemplos do connjunto de teste.

[ ]:
```
# PARA VOCÊ FAZER: visualização das classes previstas pela RNA de todas as␣
 ↪imagens do conjunto de test
fig = plt.figure(figsize=(16,9))
plt.scatter([i for i in range(Y_test.shape[0])], y_pred, label="Classe␣
 ↪Prevista", c="b")
plt.scatter([i for i in range(Y_test.shape[0])], Y_test, label="Classe Real",␣
 ↪c="r")
plt.xlabel("Exemplos")
plt.ylabel("Classe")
plt.legend()
plt.title("Classes reais e previstas - Conjunto de teste")
```

[ ]:
```
Text(0.5, 1.0, 'Classes reais e previstas - Conjunto de teste')
```



Saída prevista:

```
Dimensão vetor classes reais=  (1, 120)
Dimensão vetor classes previstas=  (120,)
```

3 Exercício #7: Criação do modelo para visualização das saídas das camadas convolucionais

Para visualizar as saídas das camadas de uma RNA deve-se criar um modelo que recebe uma imagem como entrada e gera como saída as ativações das camadas que se deseja visualizar. O Keras possui a classe de modelos "Keras Class Model"para fazer isso.

Na célula abaixo crie esse modelo usando dois argumentos: (1) tensores de entrada; (2) lista de tensores de saída, que são as saídas das 6 primeiras camadas da sua RNA (3 camadas convolucionais e 3 camadas max-pooling).

[ ]:
```
# PARA VOCÊ FAZER: criação do modelo para visualização das saídas das camadas␣
↪convolucionais
from tensorflow.keras import models
camadas_saidas = [layer.output for layer in rna.layers[:6]]
rna_ativacoes = models.Model(inputs=rna.input, outputs=camadas_saidas)
rna_ativacoes.summary()
```

```
Model: "functional_3"
_________________________________________________________________
```

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_9_input (InputLayer)  [(None, 64, 64, 3)]       0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 62, 62, 8)         224
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 31, 31, 8)         0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 29, 29, 16)        1168
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 14, 14, 16)        0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 12, 12, 32)        4640
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 6, 6, 32)          0
=================================================================
Total params: 6,032
Trainable params: 6,032
Non-trainable params: 0
_________________________________________________________________
```

Saída esperada:

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_input (InputLayer)    [(None, 64, 64, 3)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 62, 62, 8)         224
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 31, 31, 8)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 29, 29, 16)        1168
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 12, 12, 32)        4640
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 32)          0
=================================================================
Total params: 6,032
Trainable params: 6,032
Non-trainable params: 0
_________________________________________________________________
```

3.1 Exercício #8: Redimensionamento da imagem para visualização das saídas

Quando esse modelo recebe uma imagem de entrada, ele retorna as ativações das camadas da RNA original escolhidas com saídas. No caso dessa RNA temos uma entrada e seis saídas, uma saída para cada conjunto de ativações das camadas convolucionais e max-pooling.

A imagem usada como entrada dessa nova rna deve ser um tensor de mesmo tamanho que o usado na RNA original. Uma imagem colorida tem 3 eixos (altura, largura, cor) e o tensor de entrada da RNA tem 4 eixos (exemplo, altura, largura, cor), portanto, deve-se incluir um quarto eixo na imagem antes dela ser usada como entrada desse modelo.

Na célula abaixo crie um código que inclui esse novo eixo em uma imagem colorida.

[ ]:
```
# PARA VOCÊ FAZER: inclusão do eixo de exemplo em uma imagem
img = np.expand_dims(X_test[index], axis=0)
print("Dimensao do tensor criado com a imagem escolhida =", img.shape)
```

```
Dimensao do tensor criado com a imagem escolhida = (1, 64, 64, 3)
```

Saída esperada:

```
Dimensão do tensor criado com a imagem escolhida =  (1, 64, 64, 3)
```

3.2 Exercício #9: Execução da nova RNA

O próximo passo para visualização das saídas das camadas convolucionais é executar o novo modelo em modo de predição. Crie na célula abaixo um código para obter a saída da primeira camada convolucional.

[ ]:
```
# PARA VOCÊ FAZER: cálculo das saídas das camadas convolucionais
ativacoes = rna_ativacoes.predict(img)
ativacao_primeira_conv = ativacoes[0]
print("Dimensão do tensor de saída da primeira camada convolucional␣
 ↪=",ativacao_primeira_conv.shape)
```

```
Dimensão do tensor de saída da primeira camada convolucional = (1, 62, 62, 8)
```

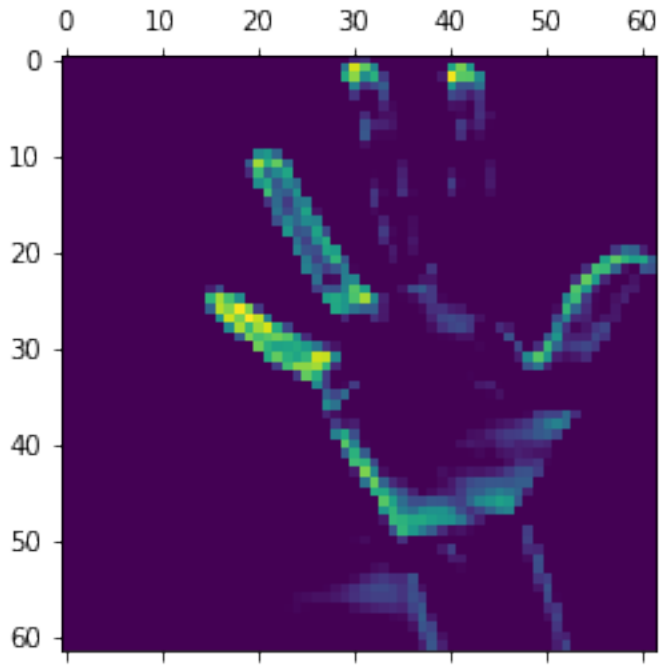
Saída esperada:

```
Dimensão do tensor de saída da primeira camada convolucional = (1, 62, 62, 8)
```

Observe que a saída dessa primeira camada convolucional é um mapa de características de dimensão 62x62 com 8 canais. Execute a célula abaixo para visualizar as saídas dos filtros dessa camada. Troque a variável index (use um valor entre 0 e 7) para visualizar os 8 canais.

[ ]:
```
index = 4
plt.matshow(ativacao_primeira_conv[0,:,:,index], cmap='viridis')
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f38d574e080>
```

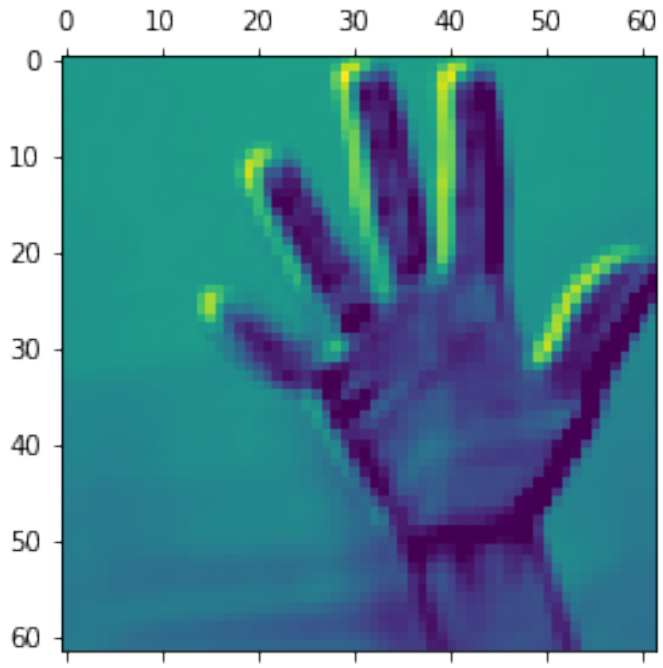


3.3 Exercíco #10:

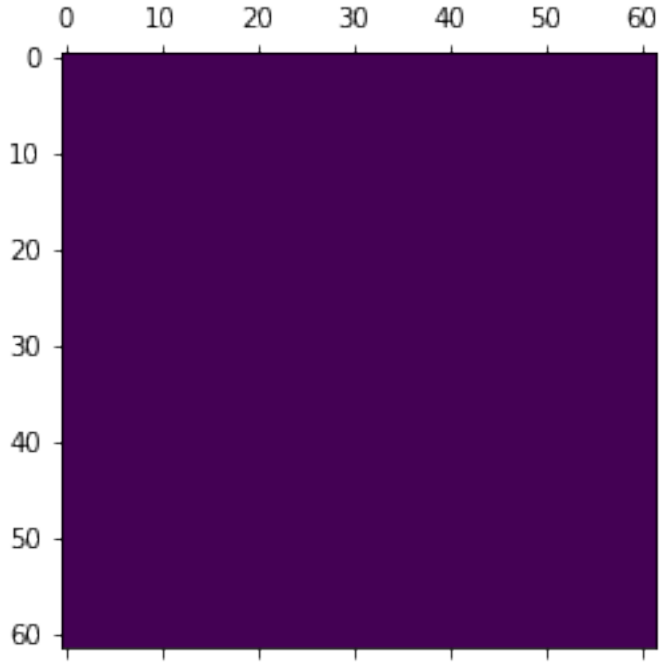
Na célula abaixo crie um código para visulizar as saídas de todos os filtros de todas as camadas convolucionais e max-polling para essa imagem de entrada.

[ ]:
```
# Para você fazer: visualização de todos os canais das saídas das camadas␣
 ↪selecionadas
for i in range(6):
    ativacao_tmp = ativacoes[i]
    print('Camada', i+1, '\n-----')
    for j in range(ativacao_tmp.shape[3]):
        print('Filtro', j+1, 'de', ativacao_tmp.shape[3])
        plt.matshow(ativacao_tmp[0,:,:,j], cmap='viridis')
        plt.show()
```

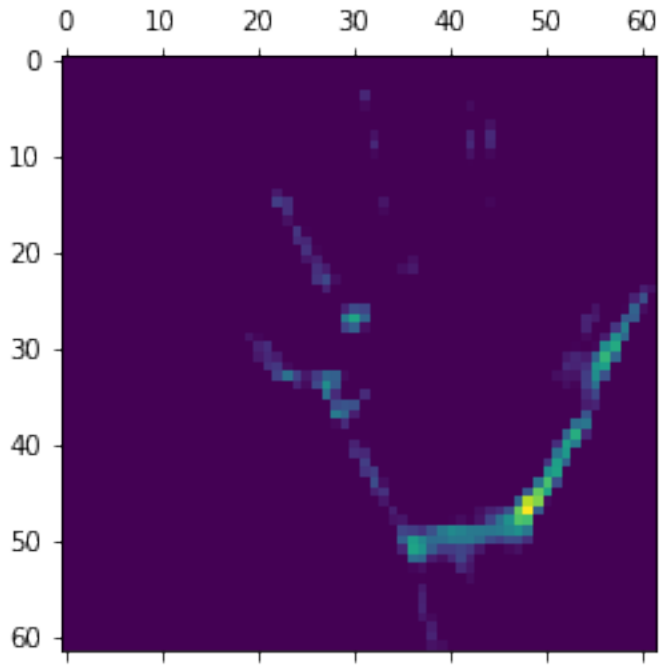
```
Camada 1
-----
Filtro 1 de 8
```



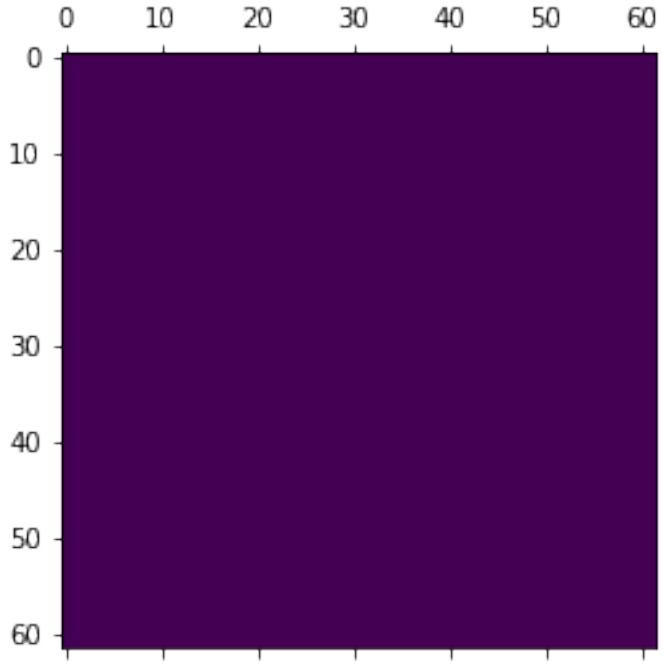
Filtro 2 de 8



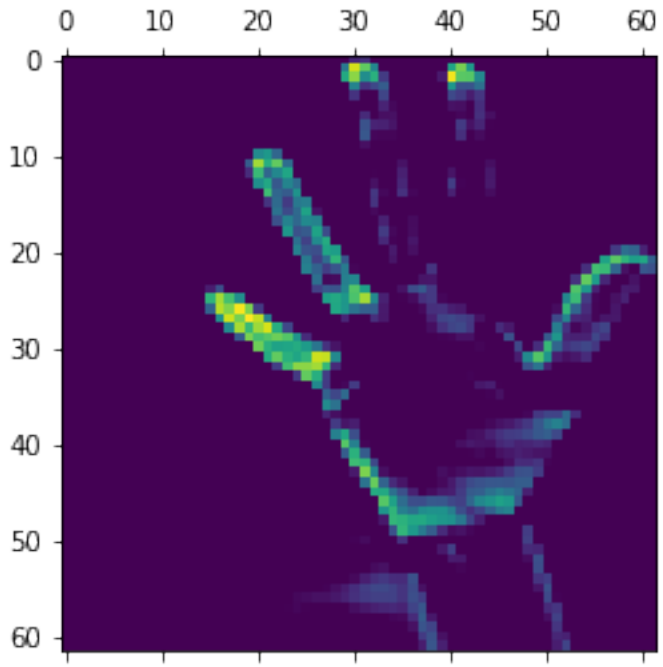
Filtro 3 de 8



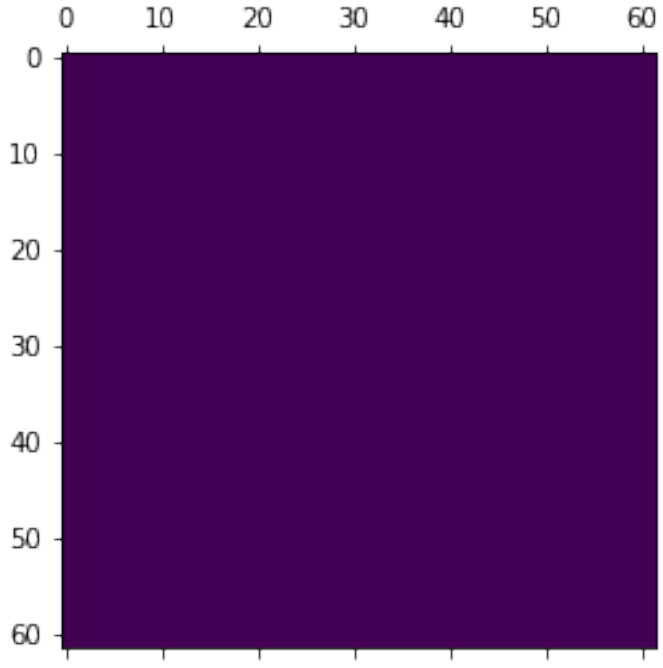
Filtro 4 de 8



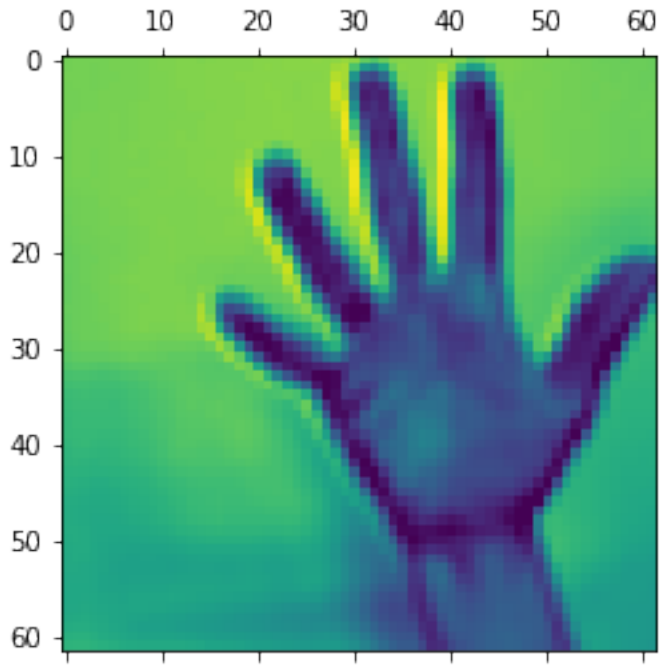
Filtro 5 de 8



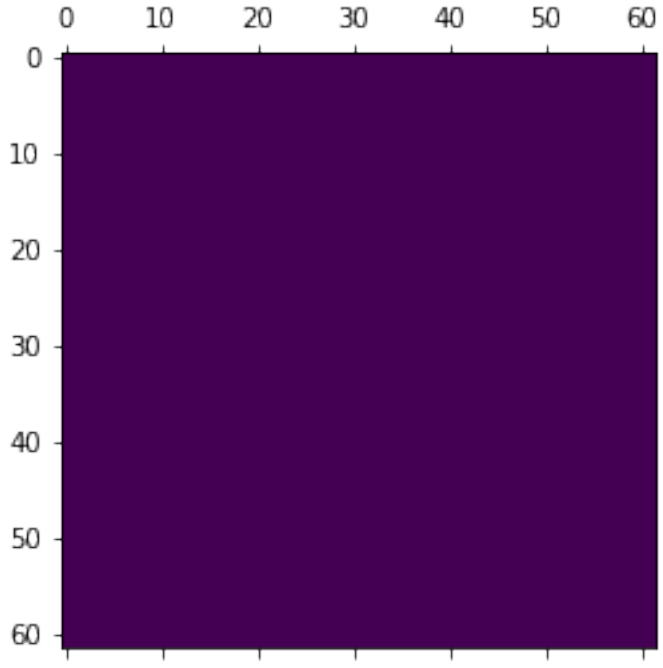
Filtro 6 de 8



Filtro 7 de 8

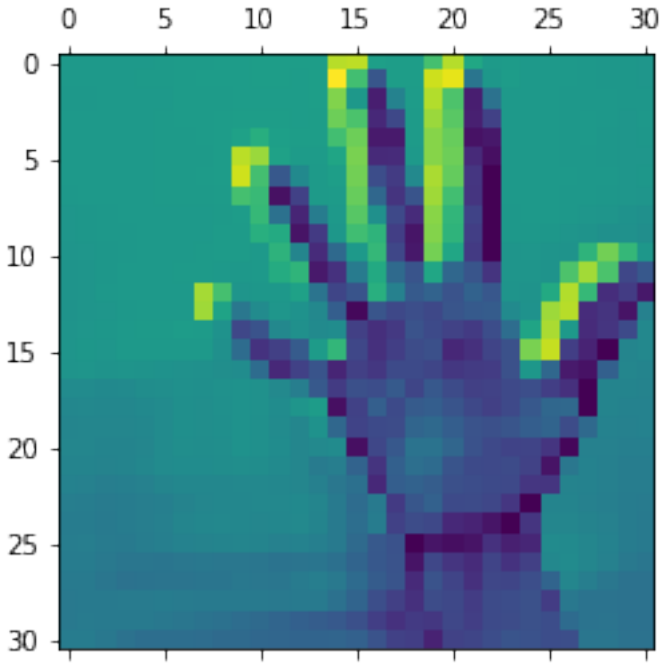


Filtro 8 de 8

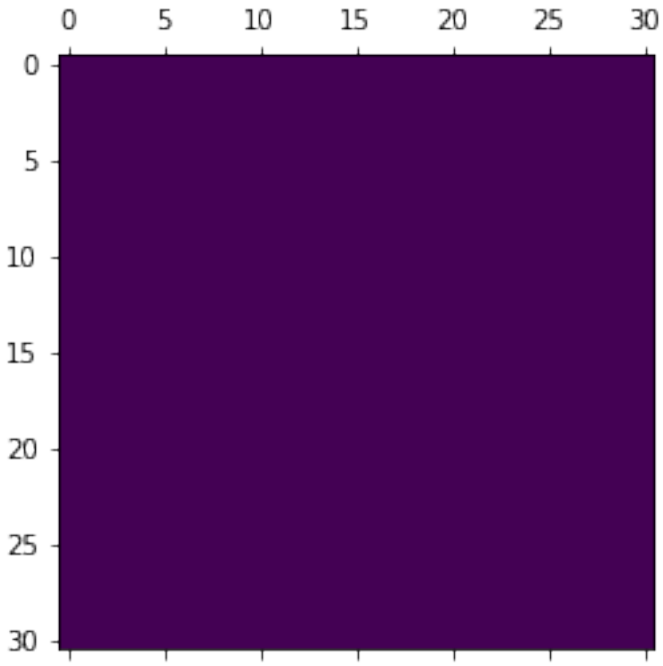


Camada 2

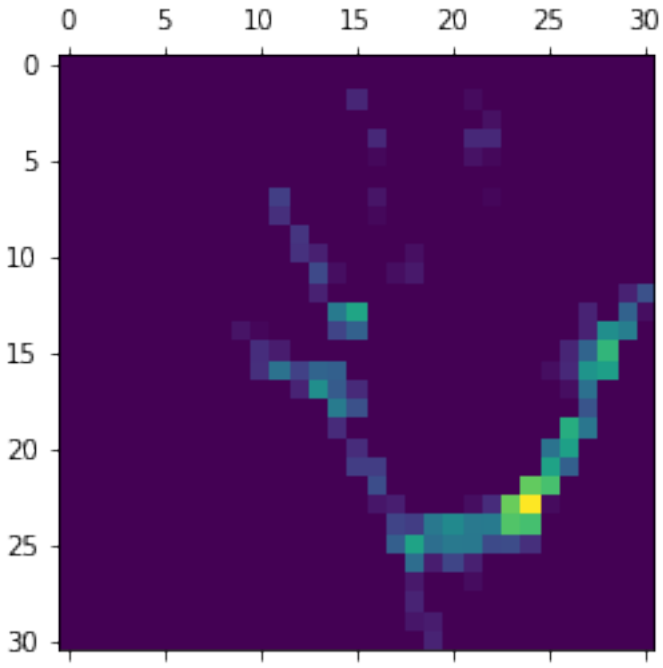
Filtro 1 de 8



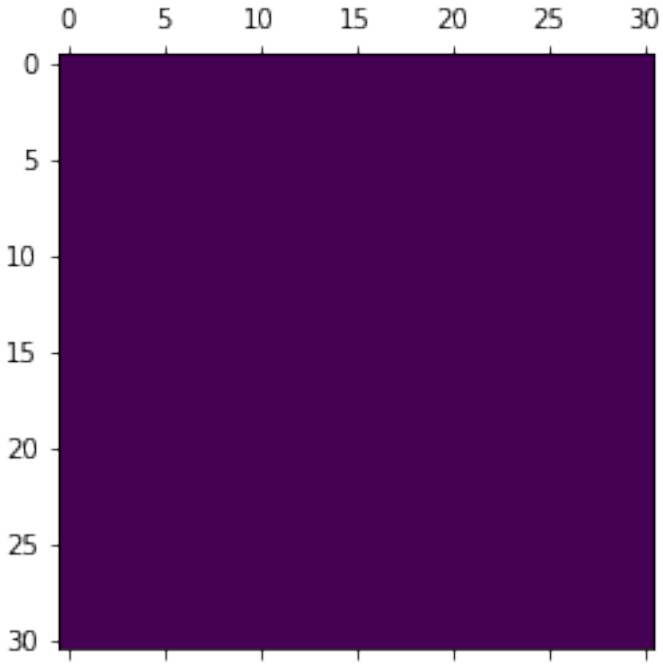
Filtro 2 de 8



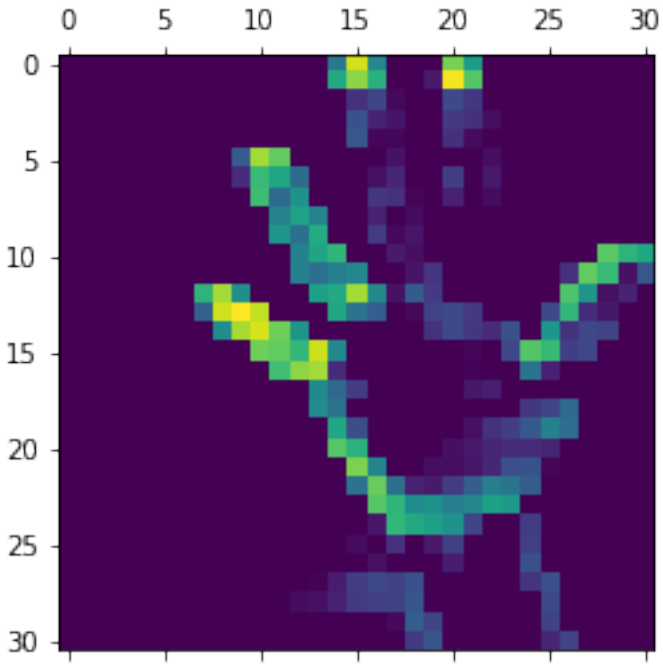
Filtro 3 de 8



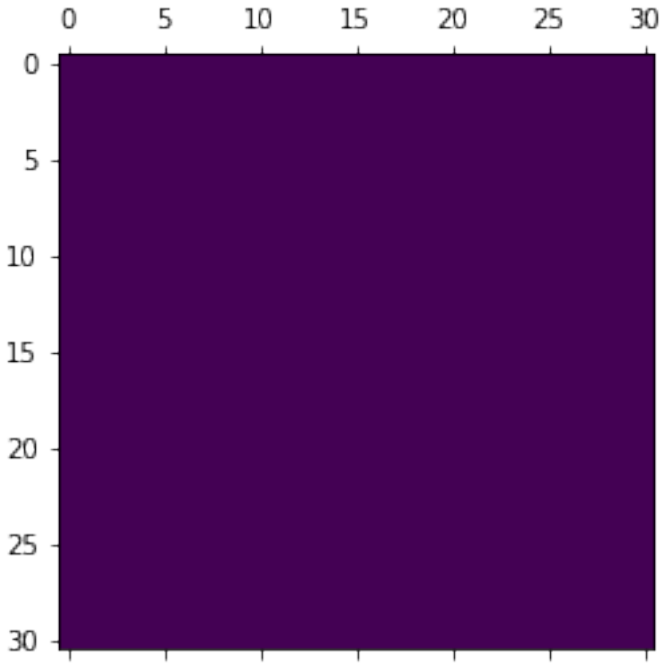
Filtro 4 de 8



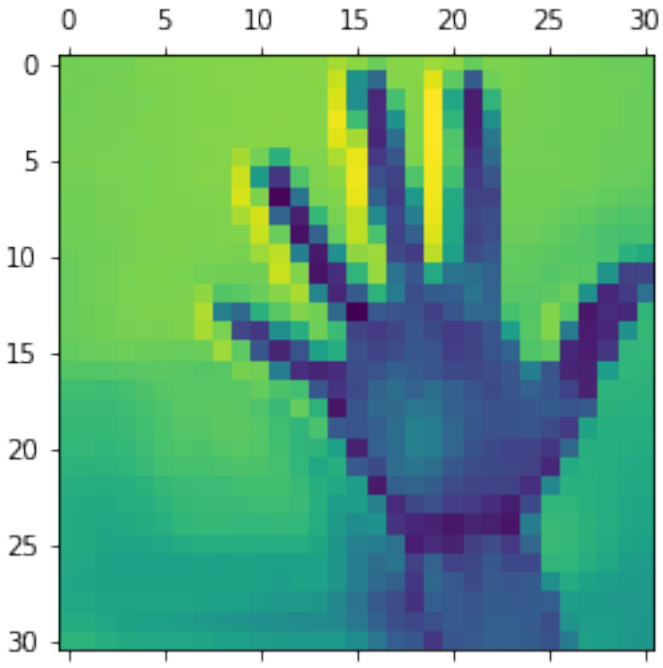
Filtro 5 de 8



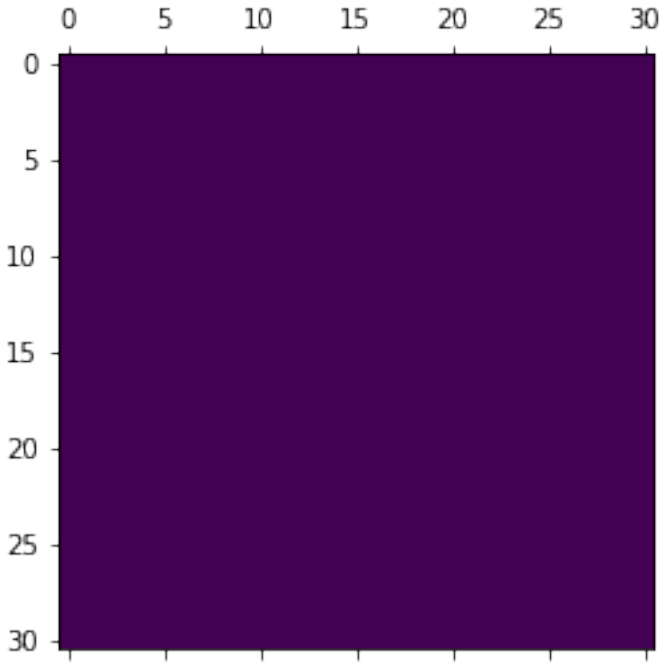
Filtro 6 de 8



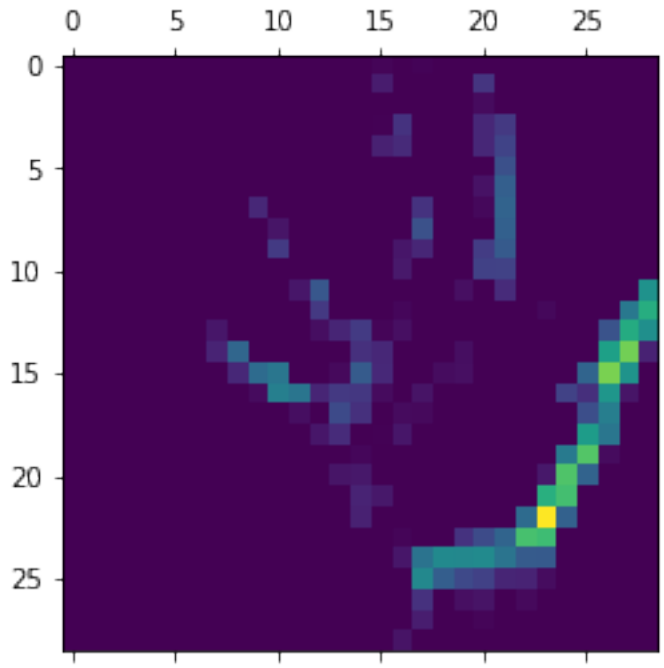
Filtro 7 de 8



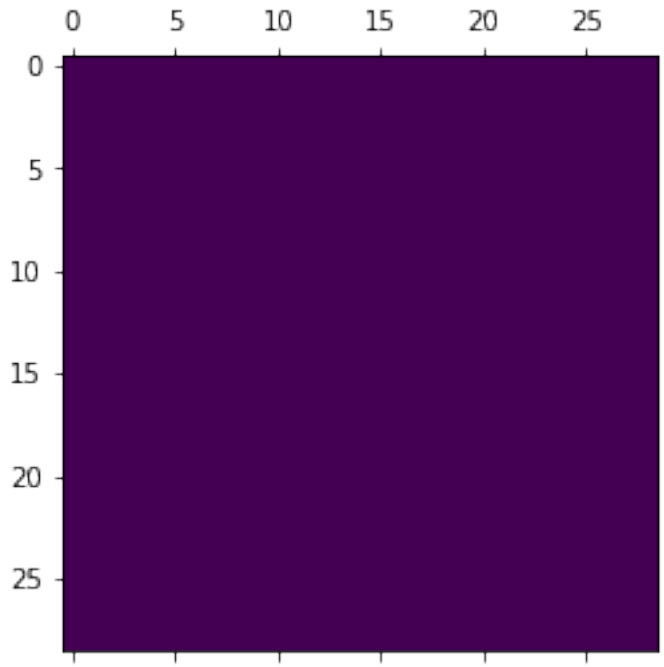
Filtro 8 de 8



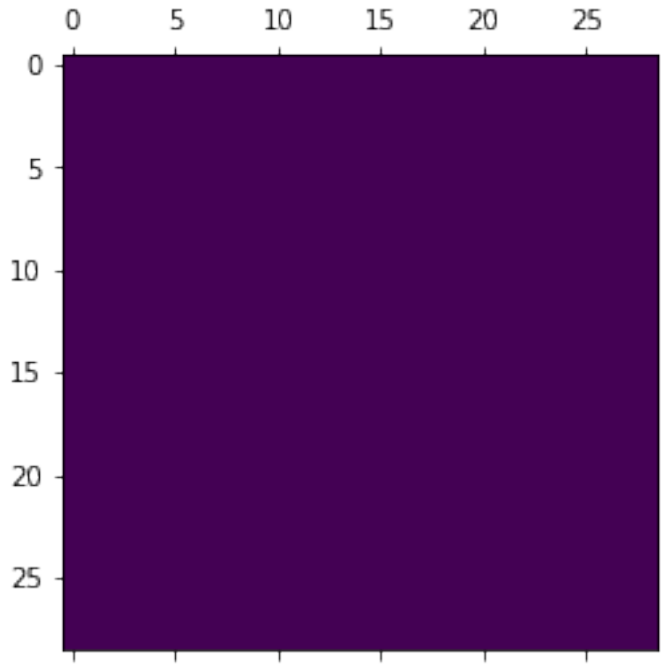
```
Camada 3
-----
Filtro 1 de 16
```



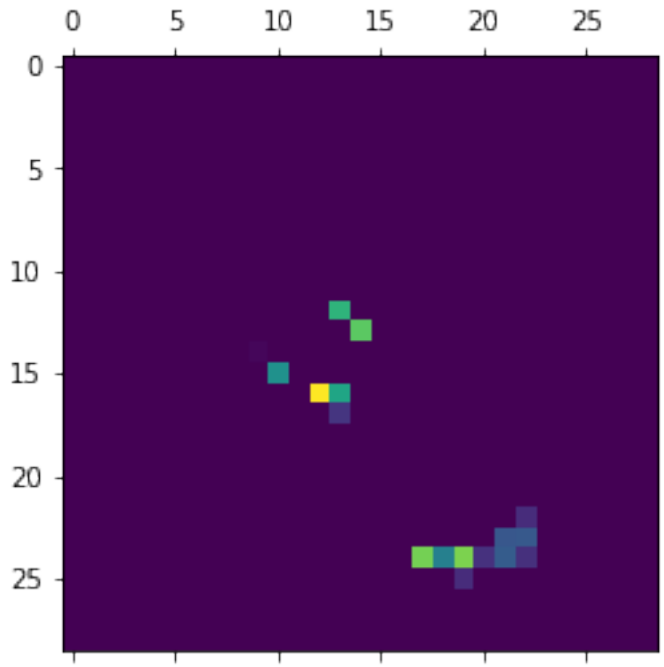

```
Filtro 2 de 16
```



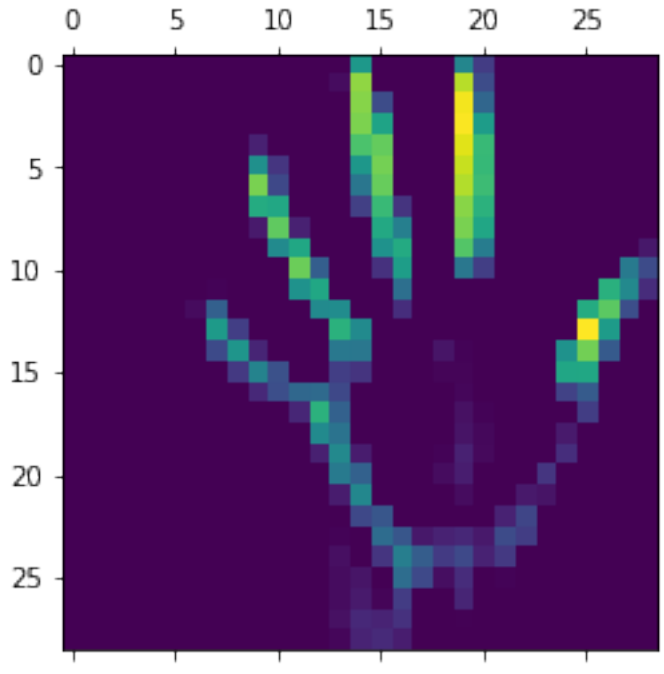
Filtro 3 de 16



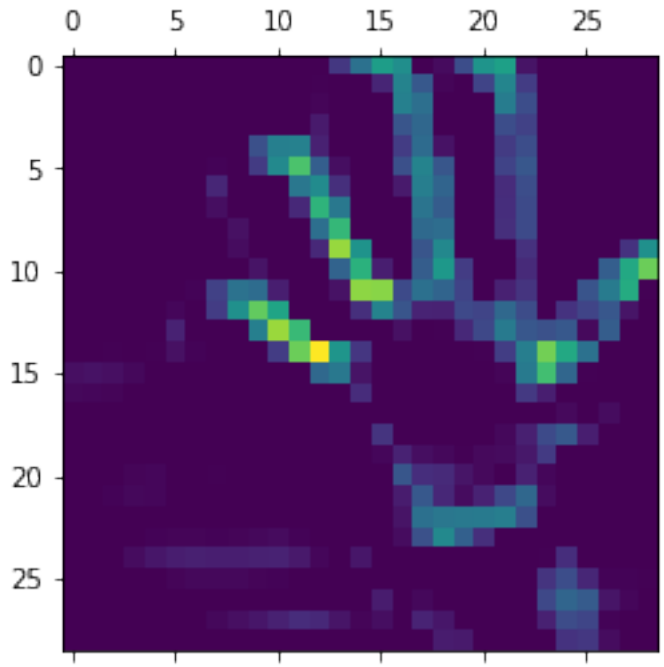
Filtro 4 de 16



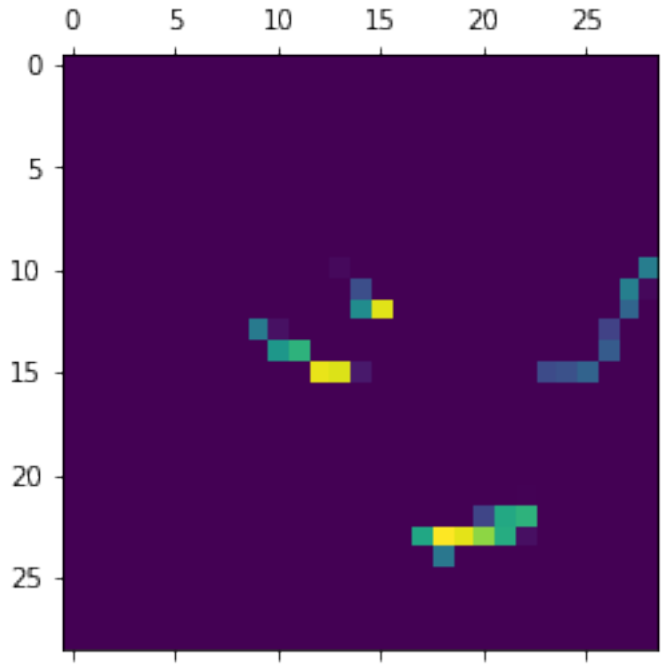
Filtro 5 de 16



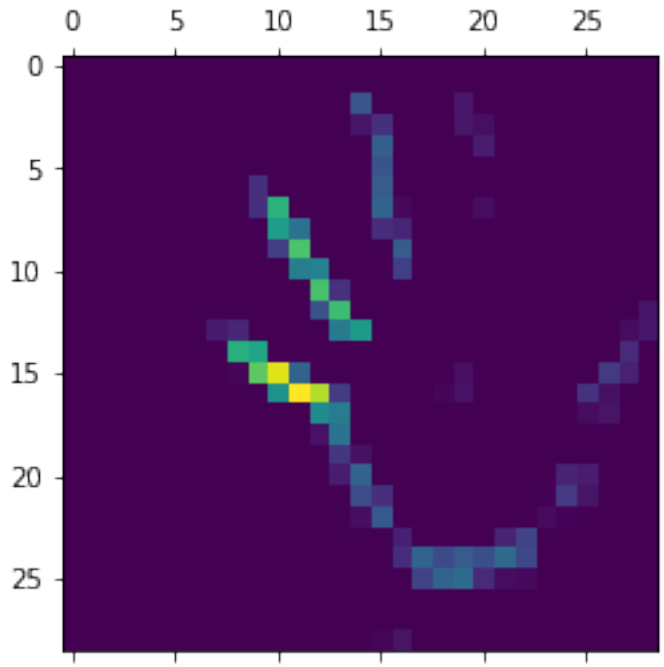
Filtro 6 de 16



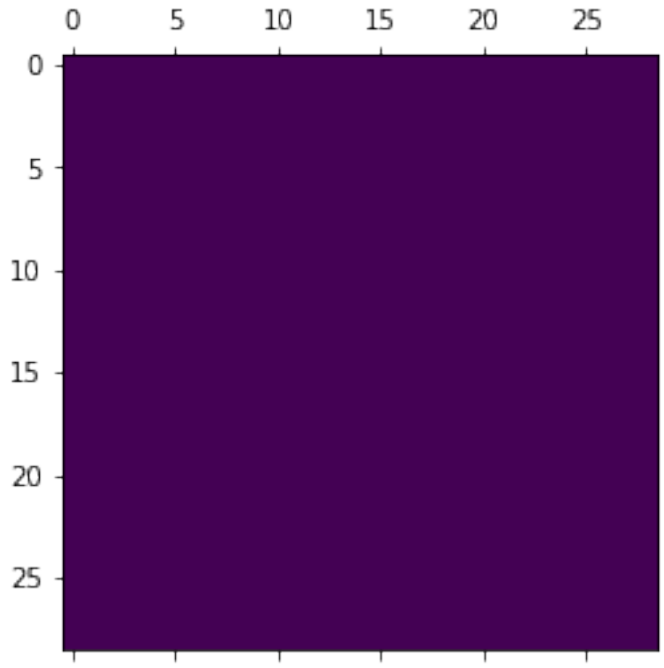
Filtro 7 de 16



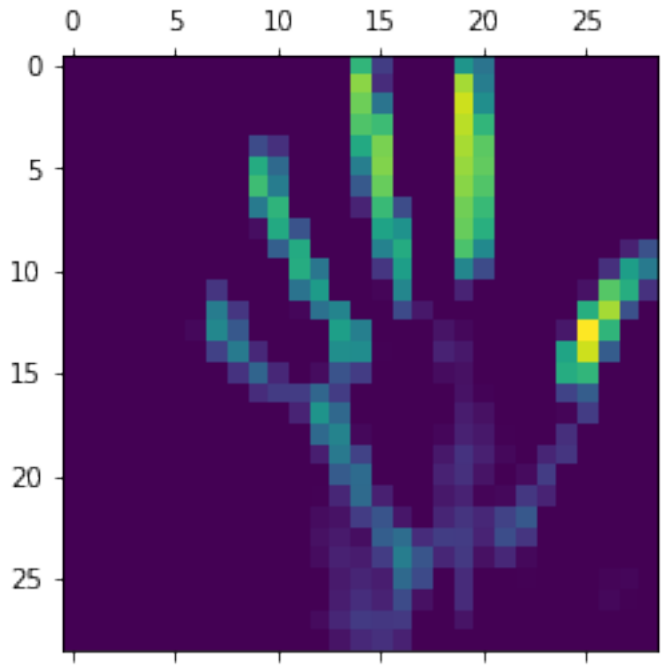
Filtro 8 de 16



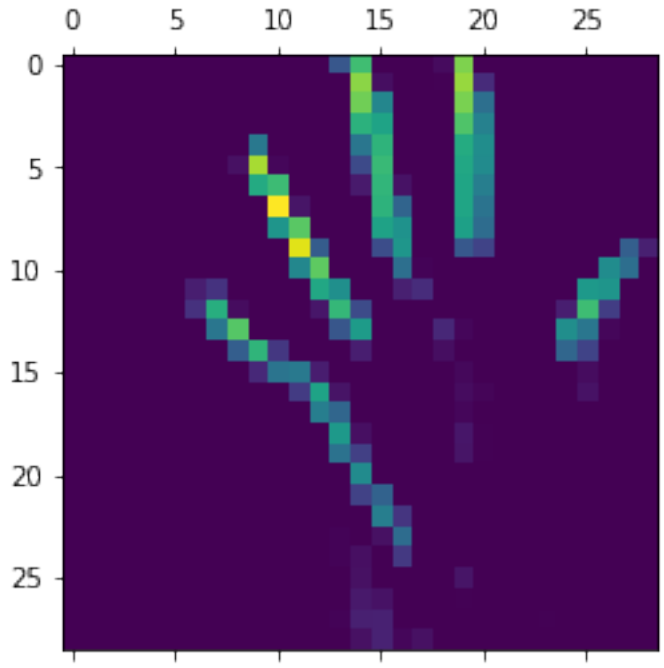
Filtro 9 de 16



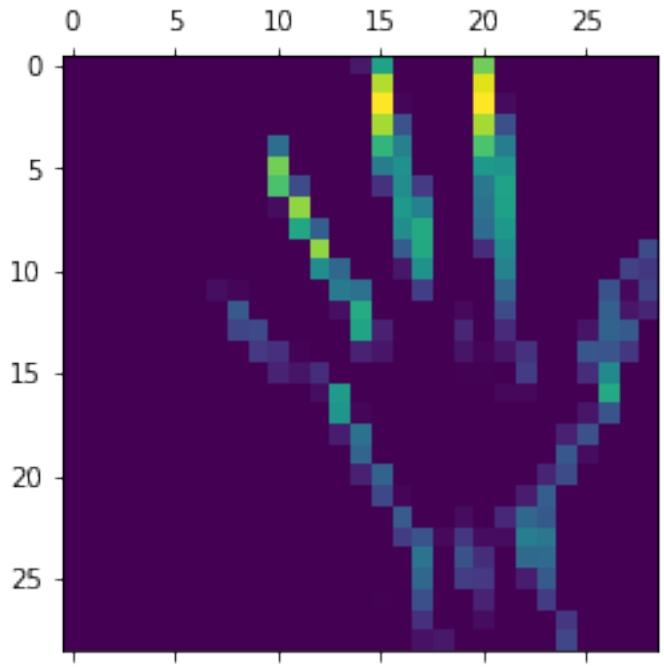
Filtro 10 de 16



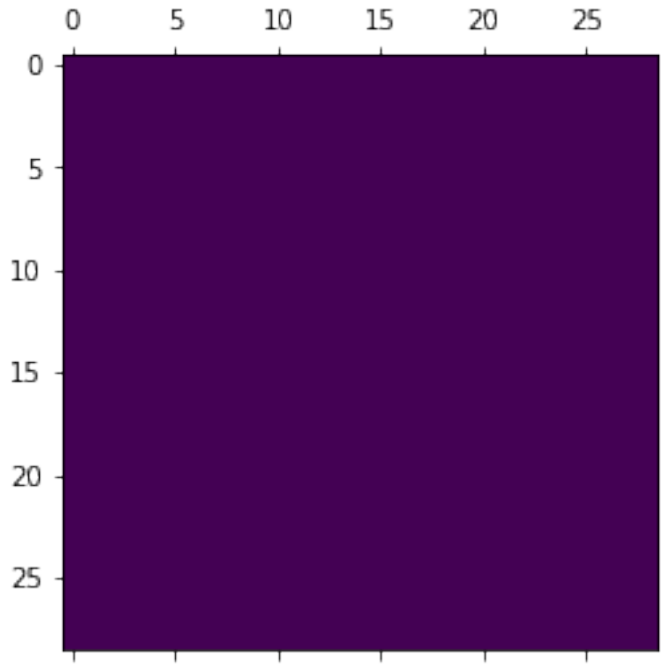
Filtro 11 de 16



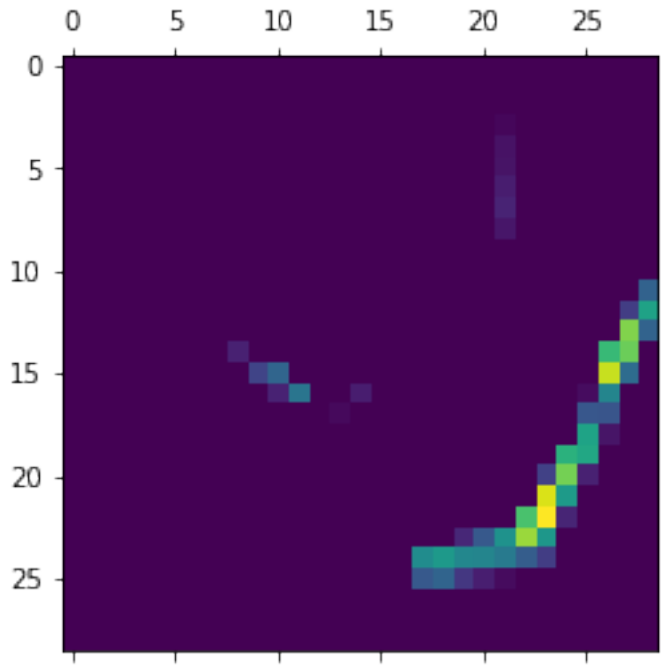
Filtro 12 de 16



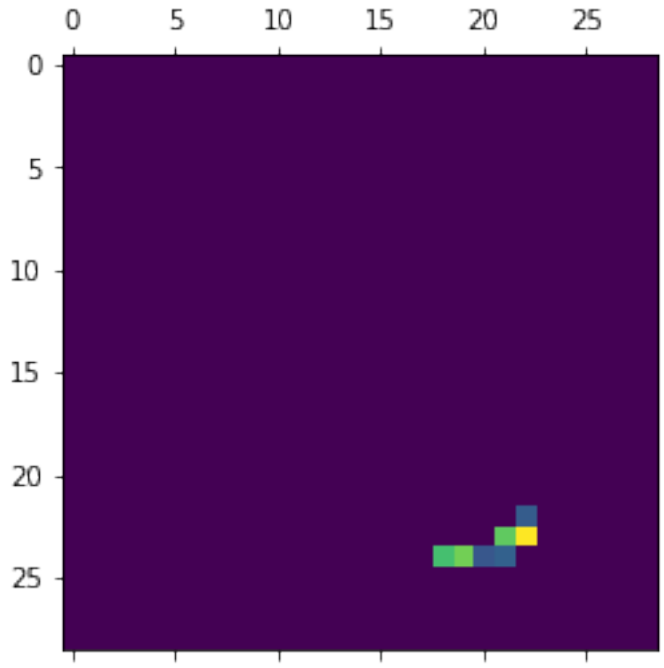
Filtro 13 de 16



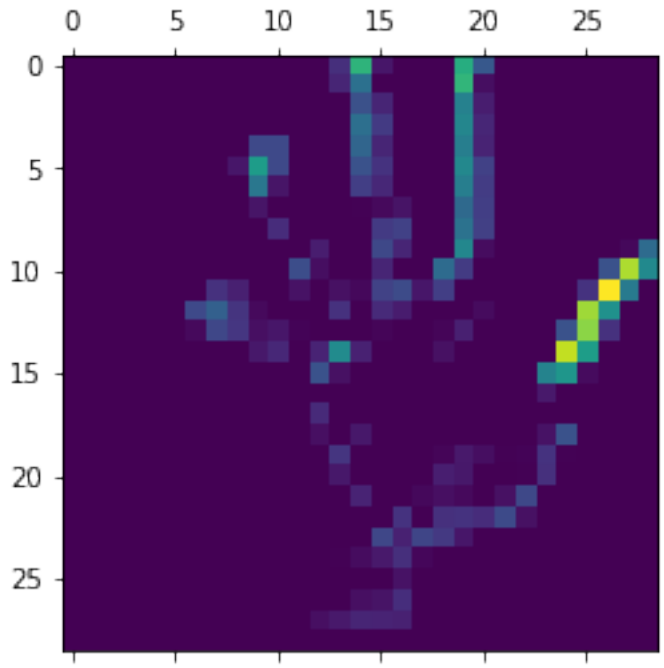
Filtro 14 de 16



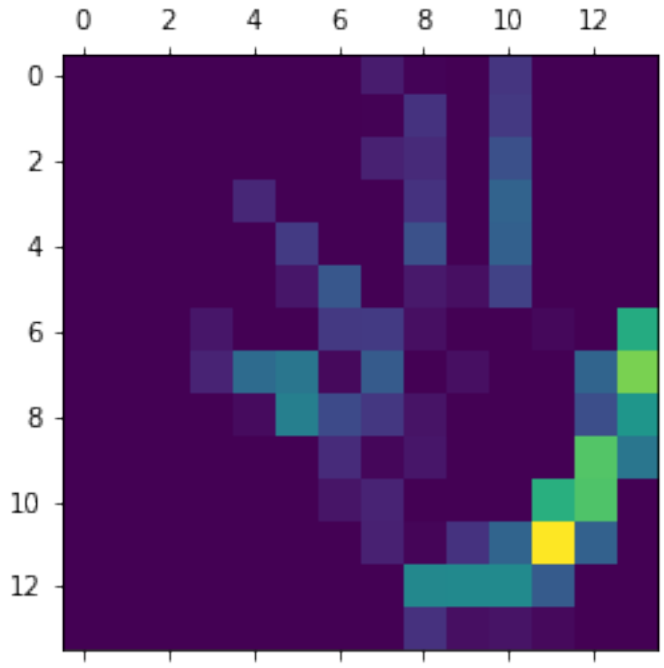
Filtro 15 de 16



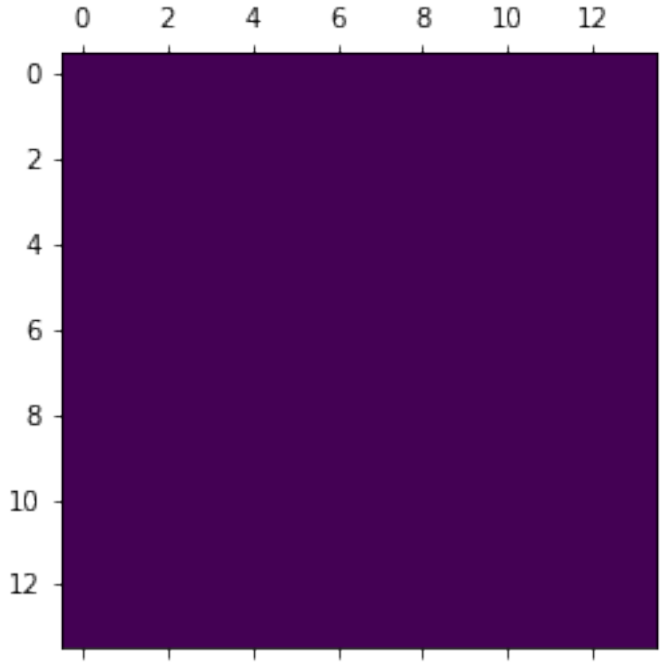
Filtro 16 de 16



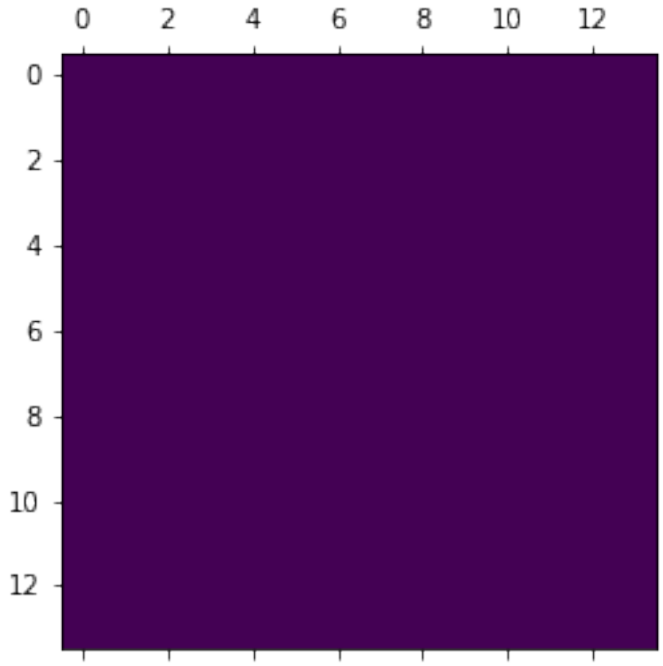

```
Camada 4
-----
Filtro 1 de 16
```



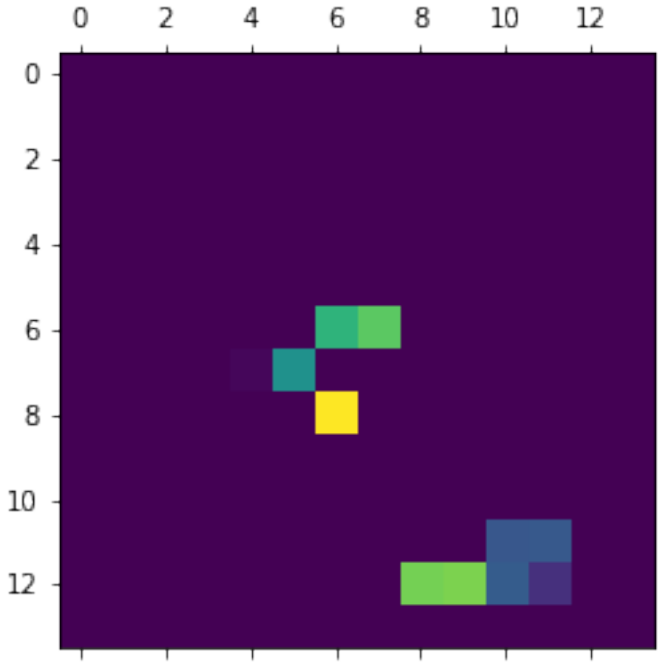
Filtro 2 de 16



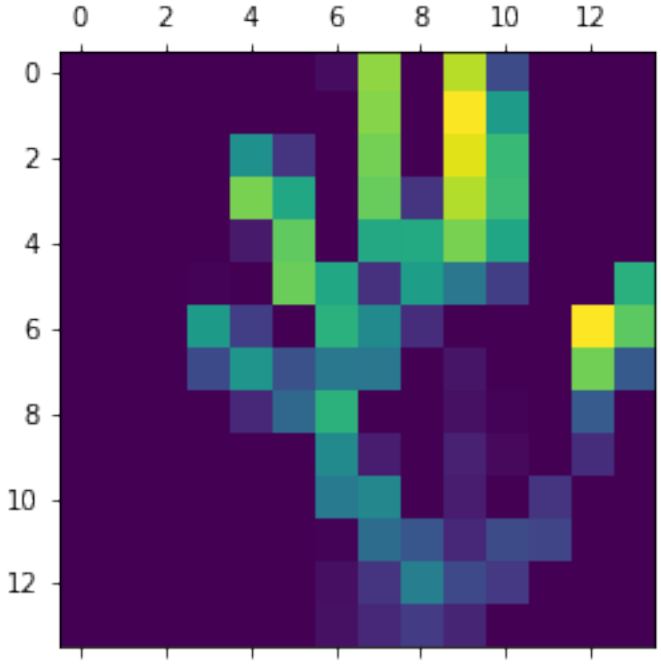
Filtro 3 de 16



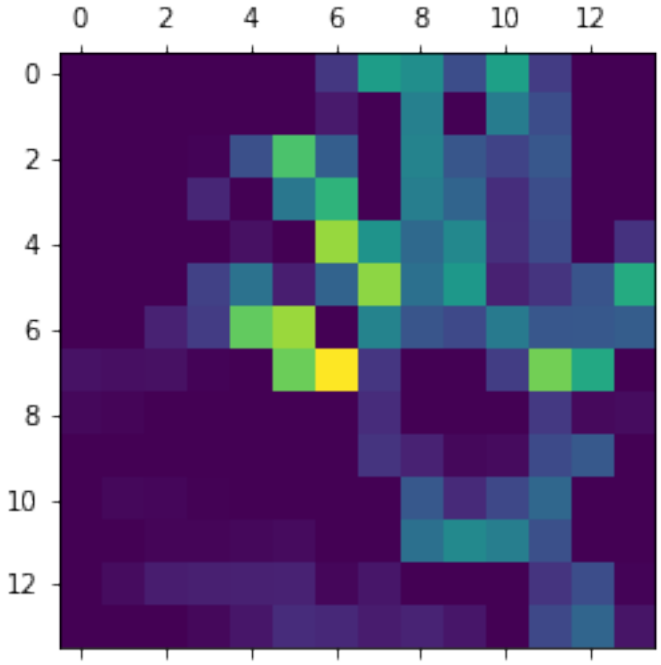
Filtro 4 de 16



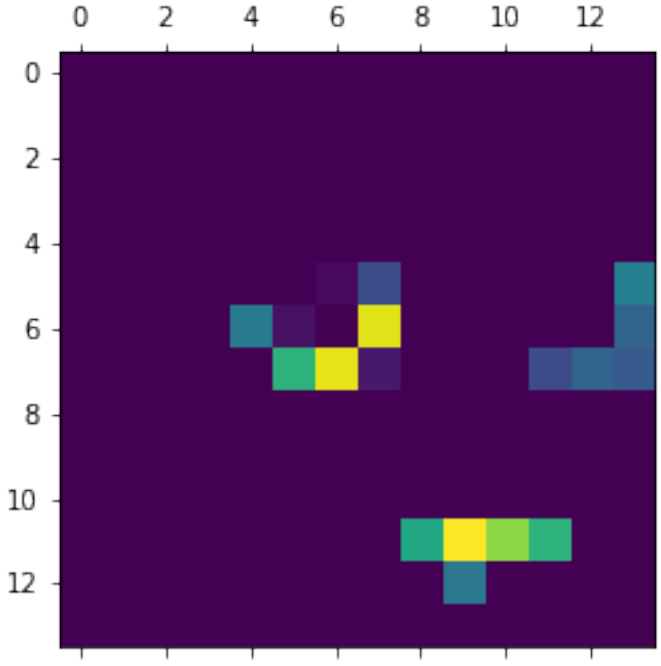
Filtro 5 de 16



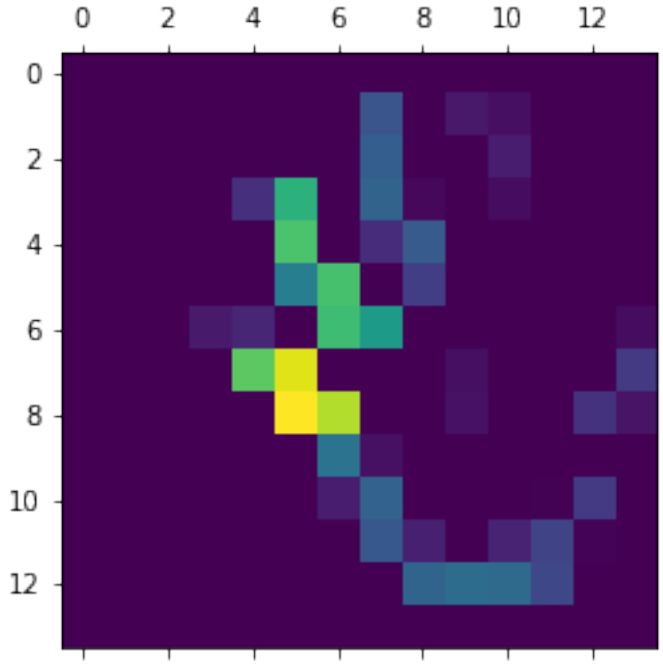
Filtro 6 de 16



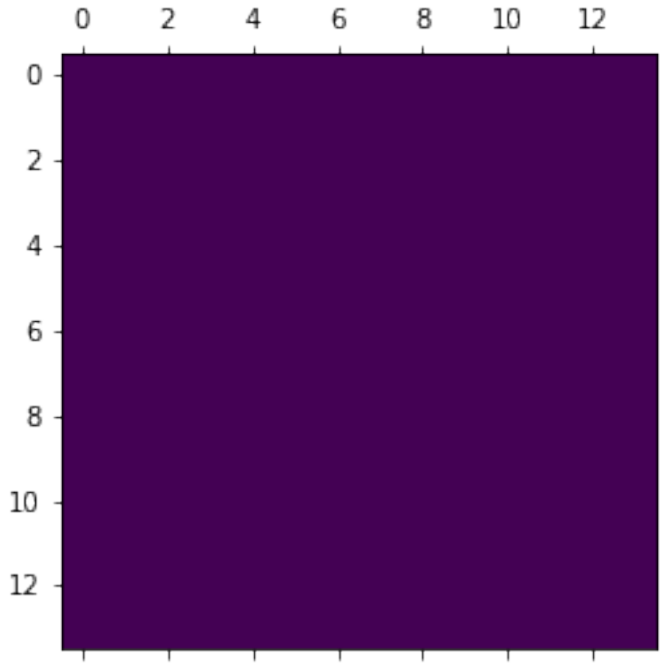
Filtro 7 de 16



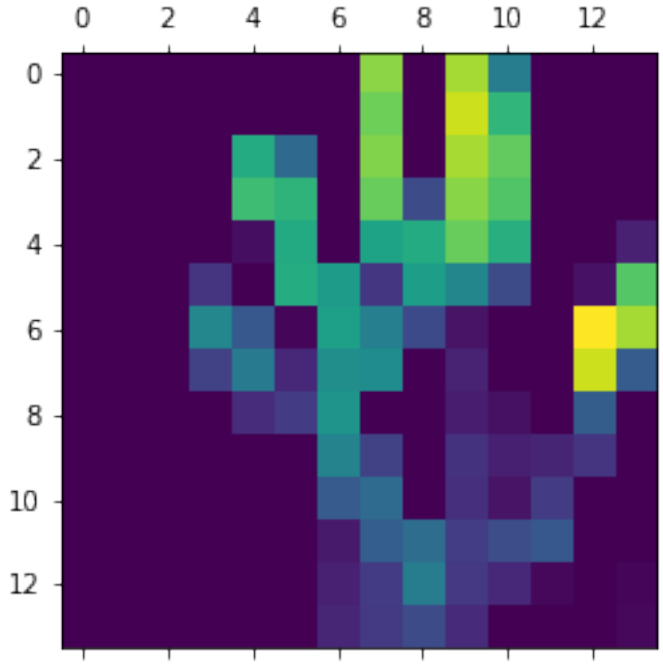
Filtro 8 de 16



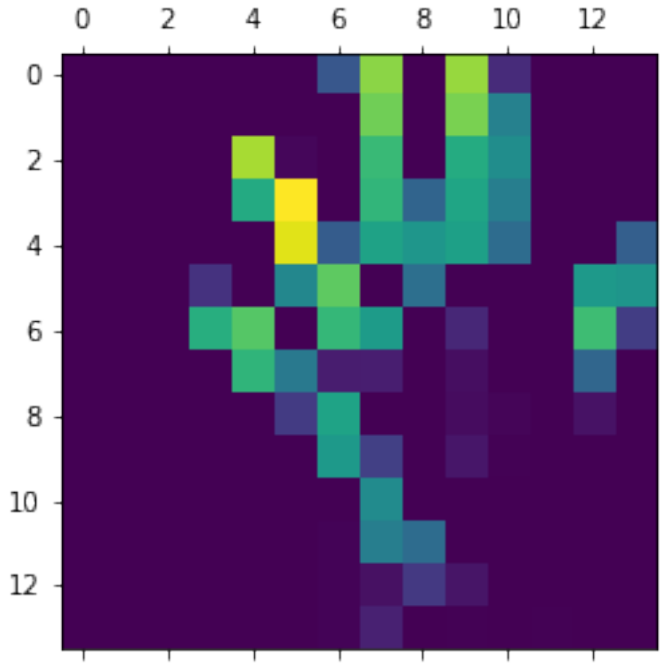
Filtro 9 de 16



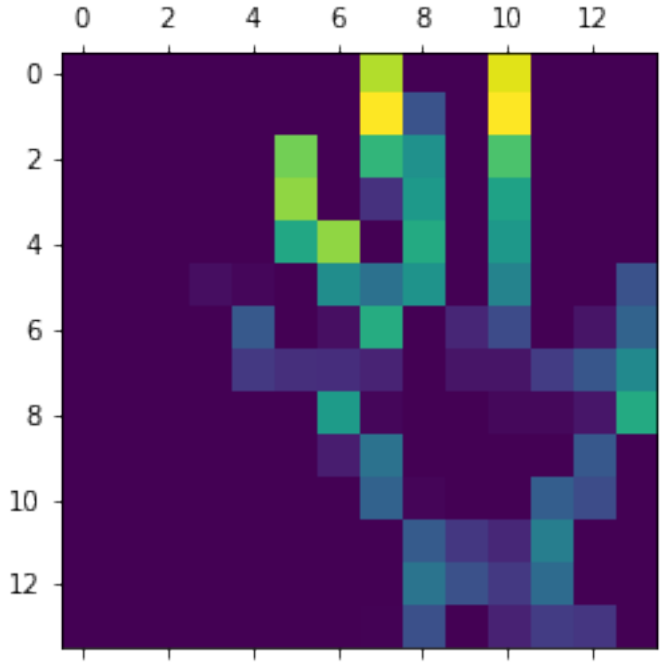
Filtro 10 de 16



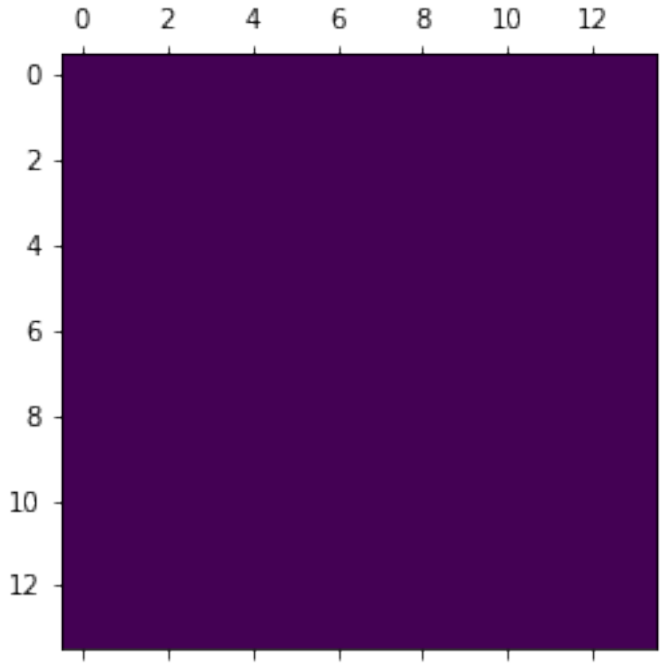
Filtro 11 de 16



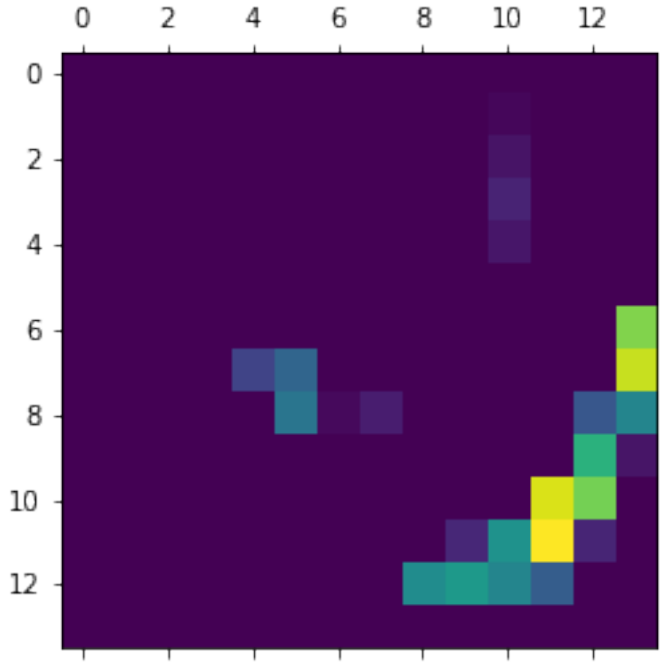
Filtro 12 de 16



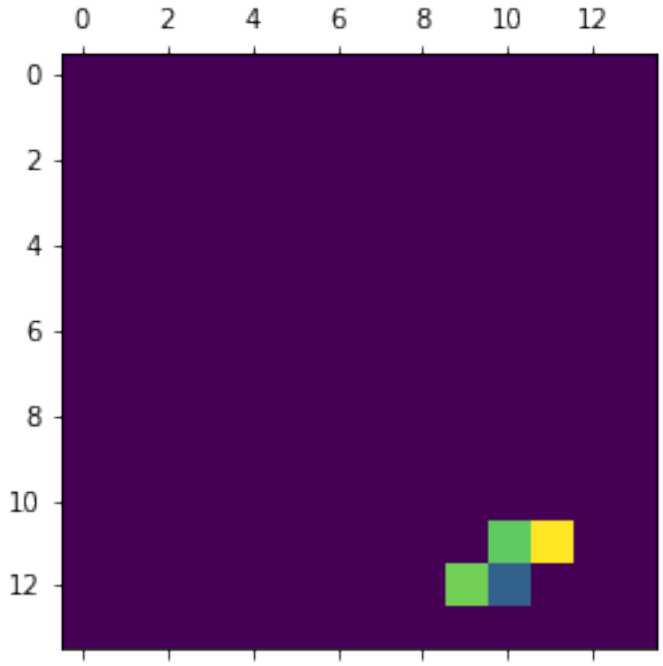
Filtro 13 de 16



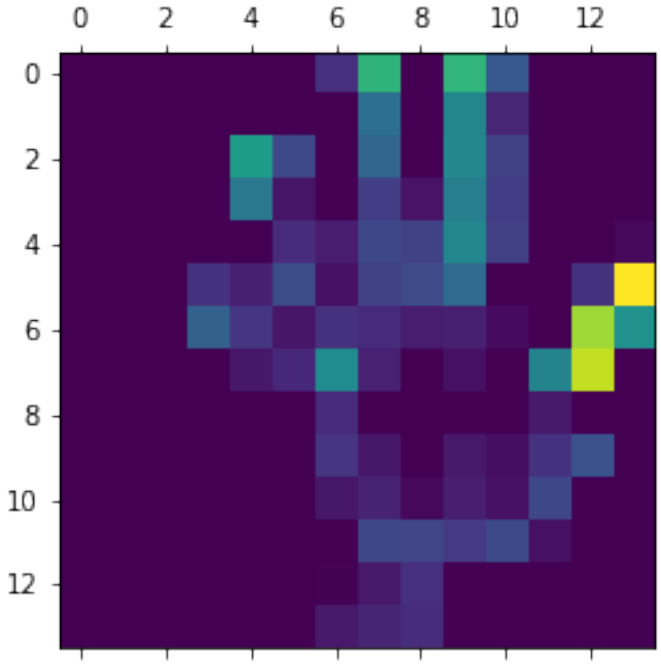
Filtro 14 de 16



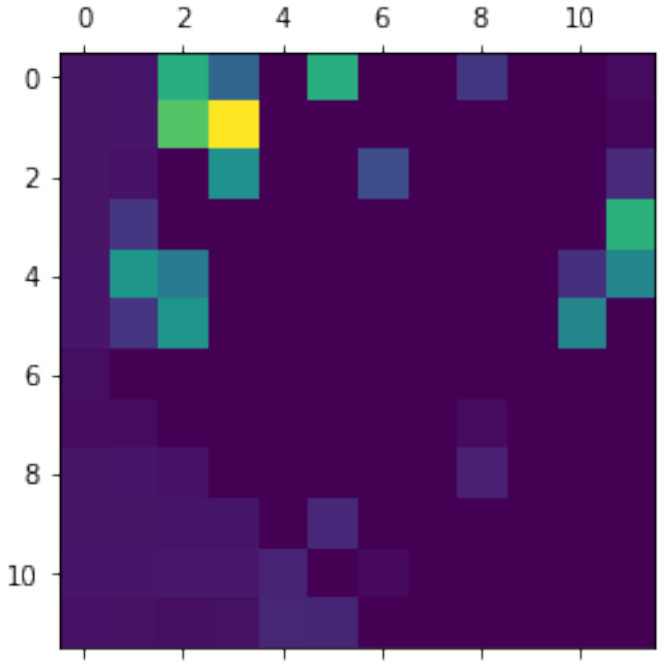
Filtro 15 de 16



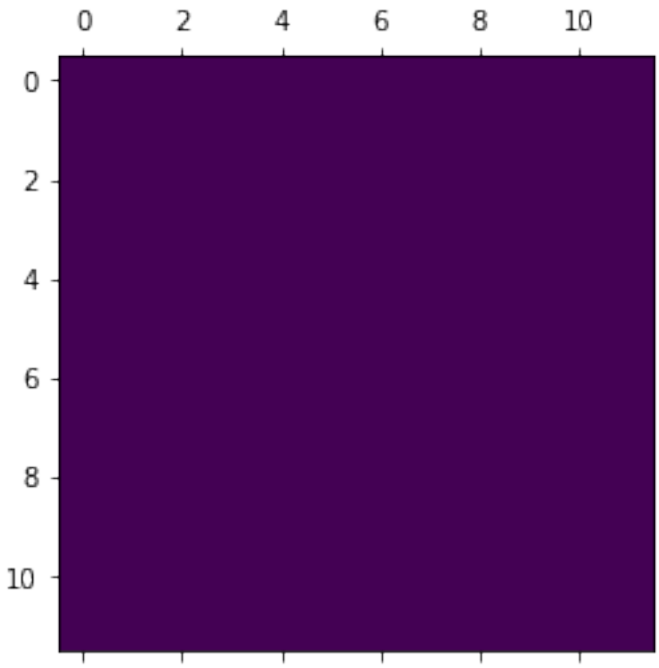
```
Filtro 16 de 16
```



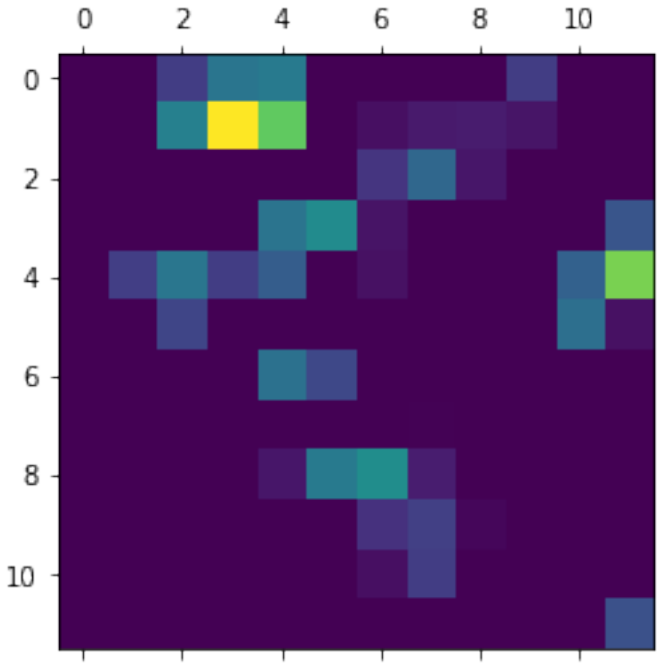

```
Camada 5
-----
Filtro 1 de 32
```



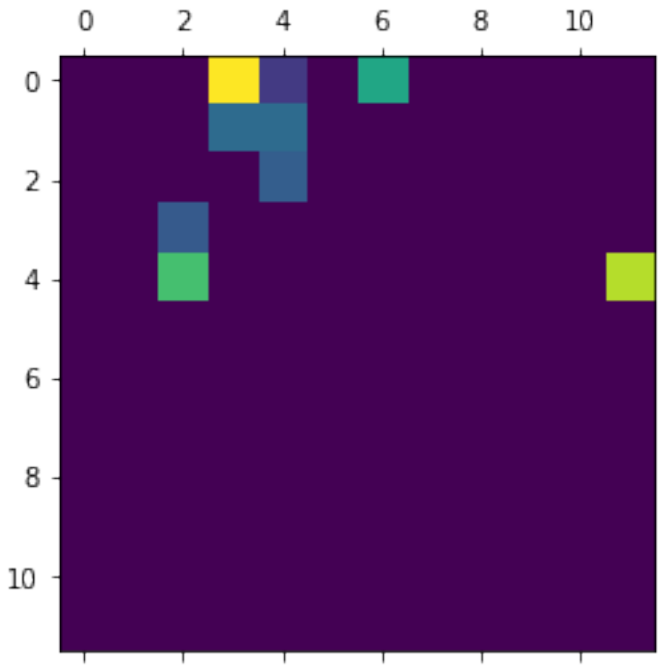
Filtro 2 de 32



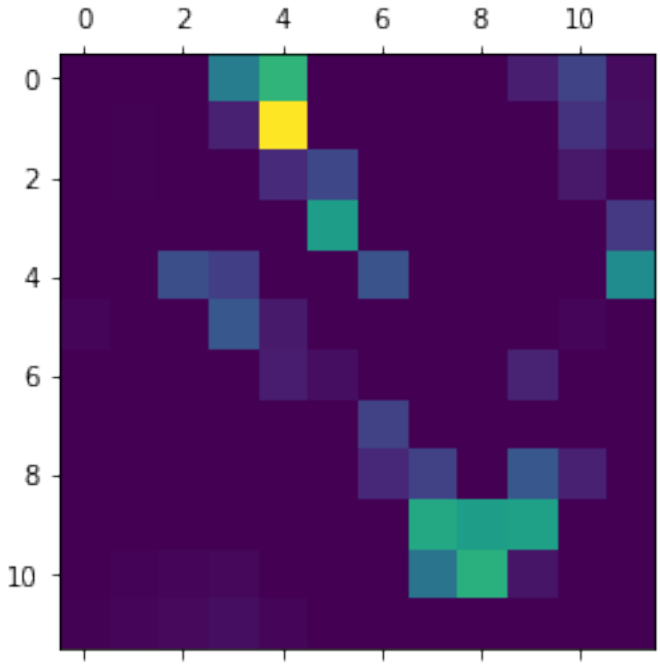
Filtro 3 de 32



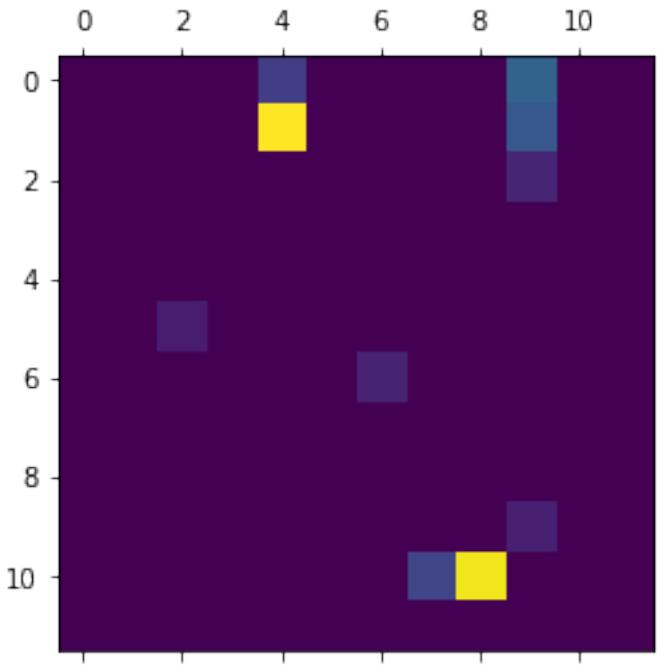
Filtro 4 de 32



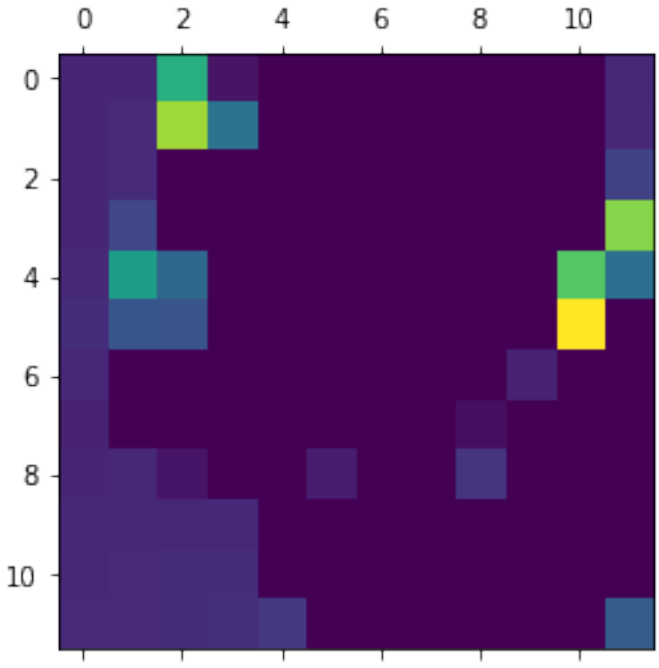
Filtro 5 de 32



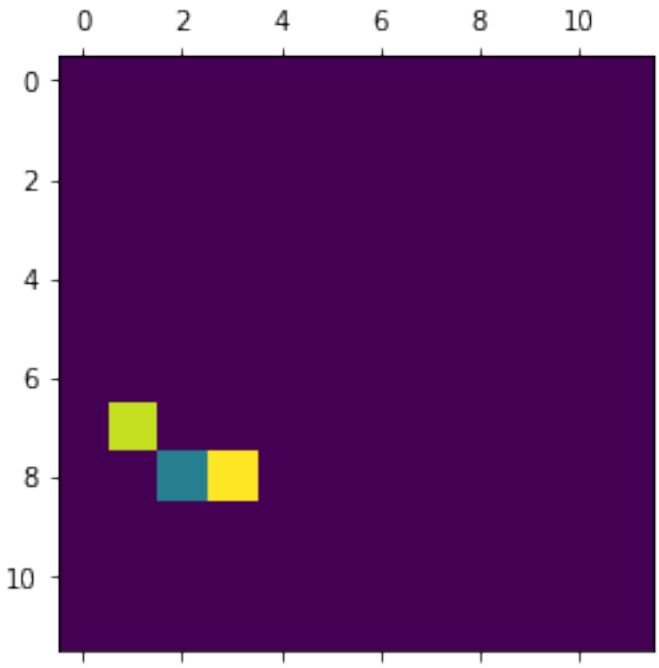
Filtro 6 de 32



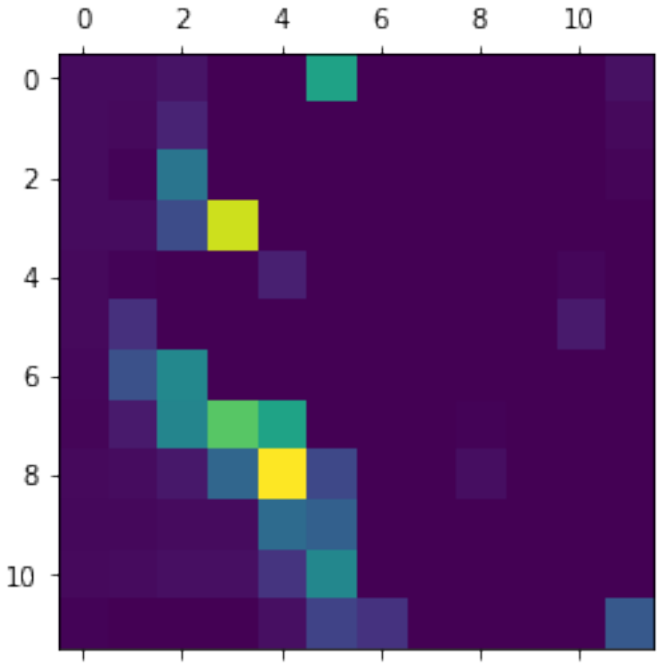
Filtro 7 de 32



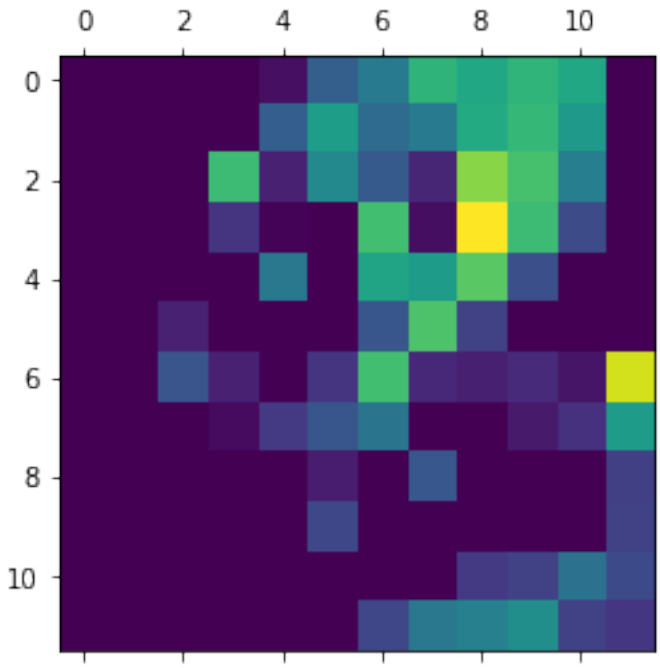
Filtro 8 de 32



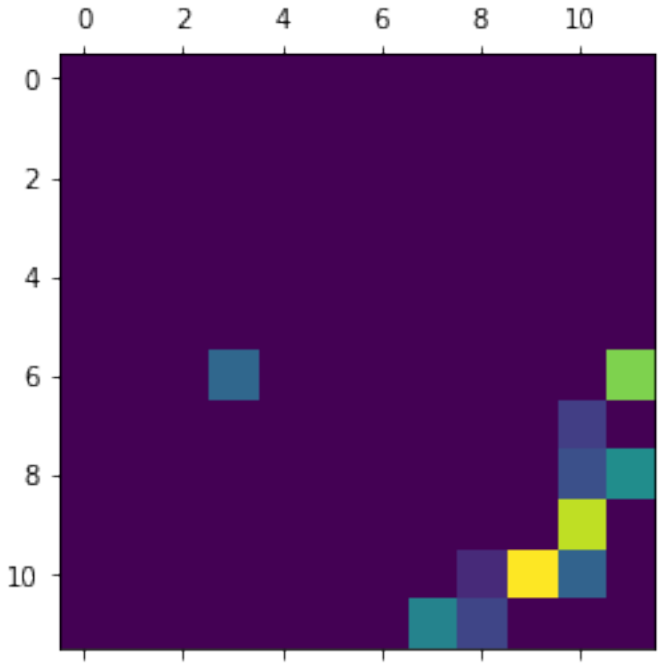
Filtro 9 de 32



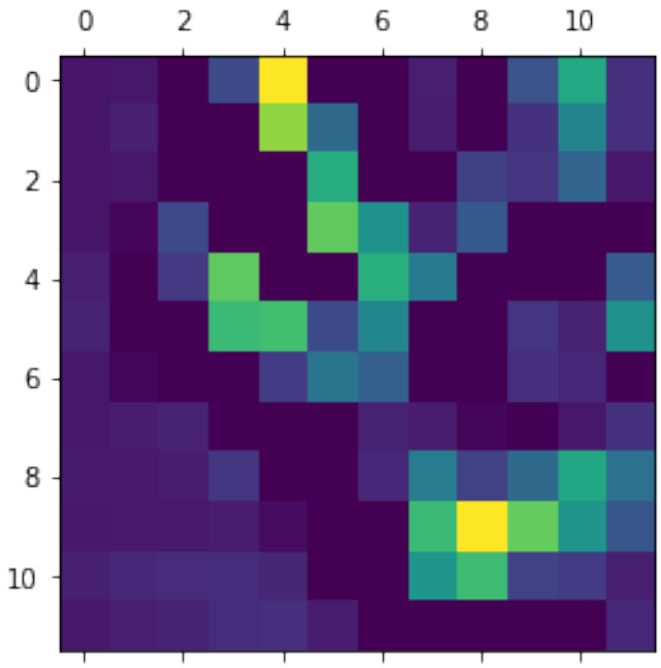
Filtro 10 de 32



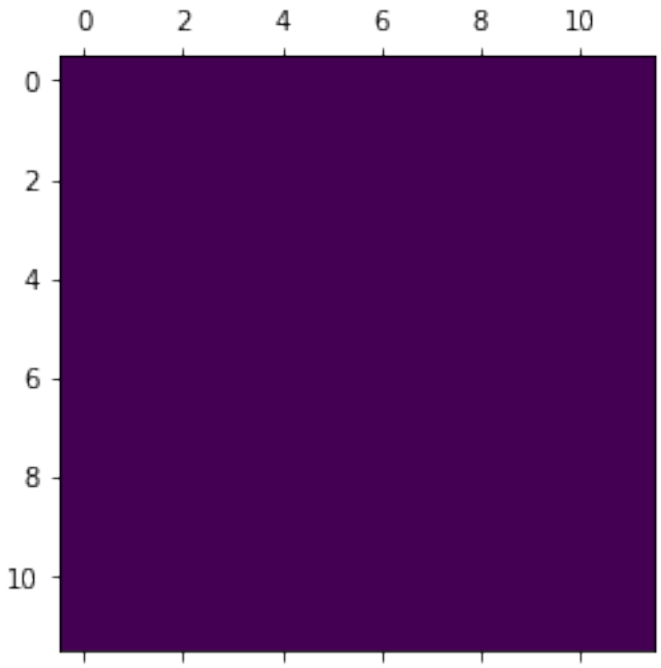
Filtro 11 de 32



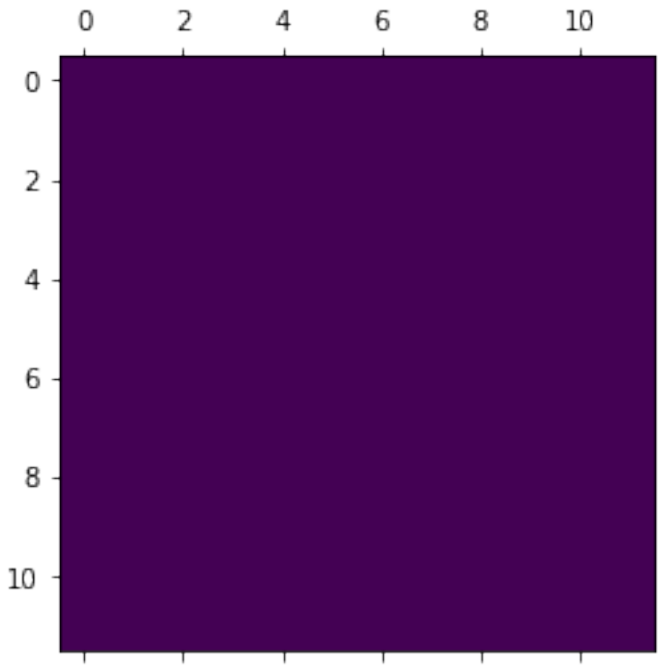
Filtro 12 de 32



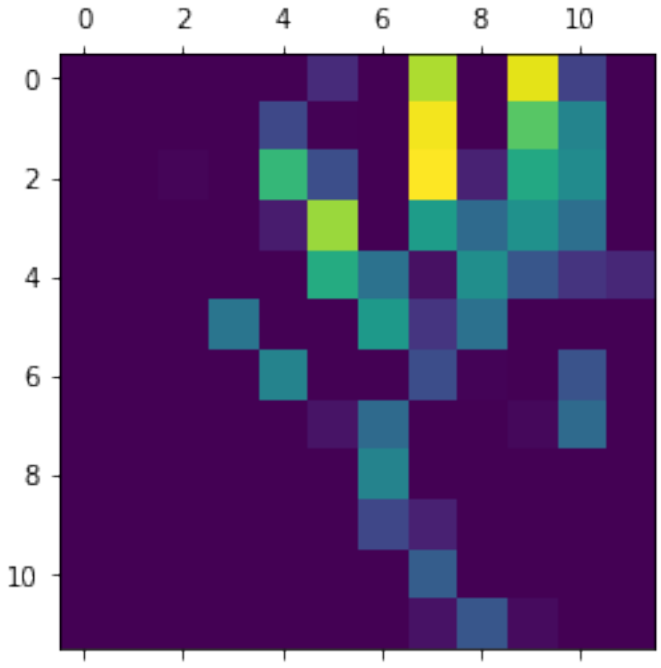
Filtro 13 de 32



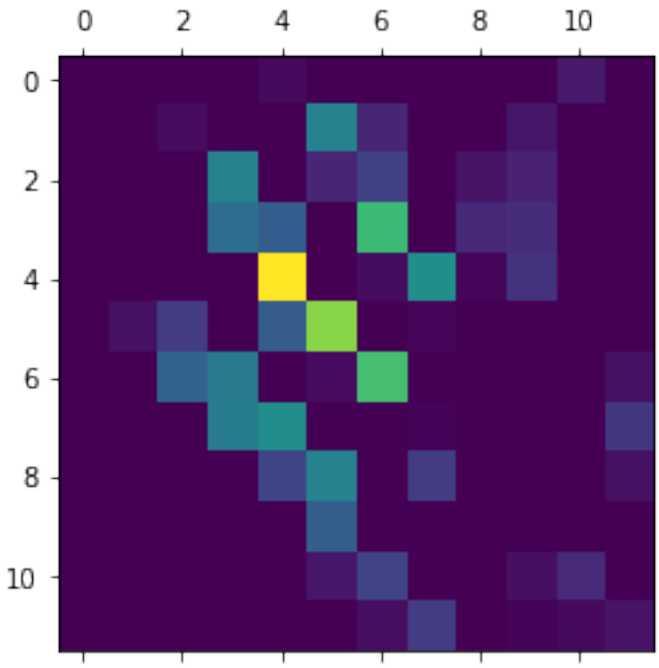
Filtro 14 de 32



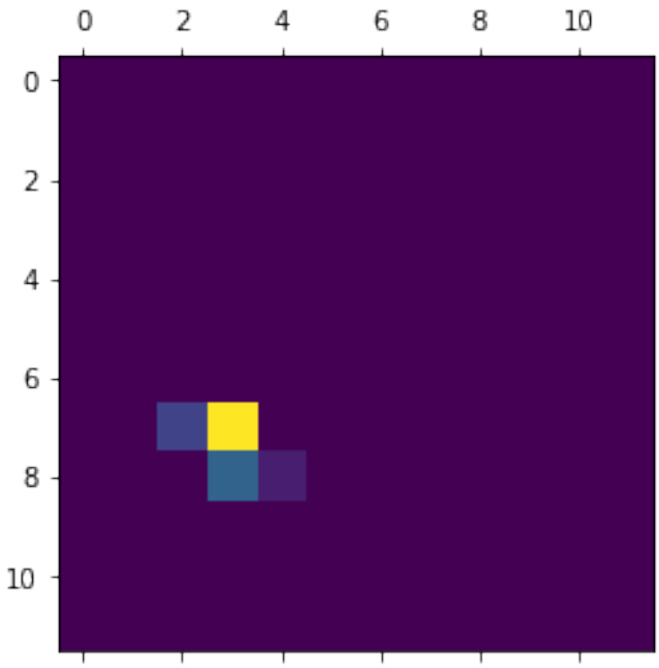
Filtro 15 de 32



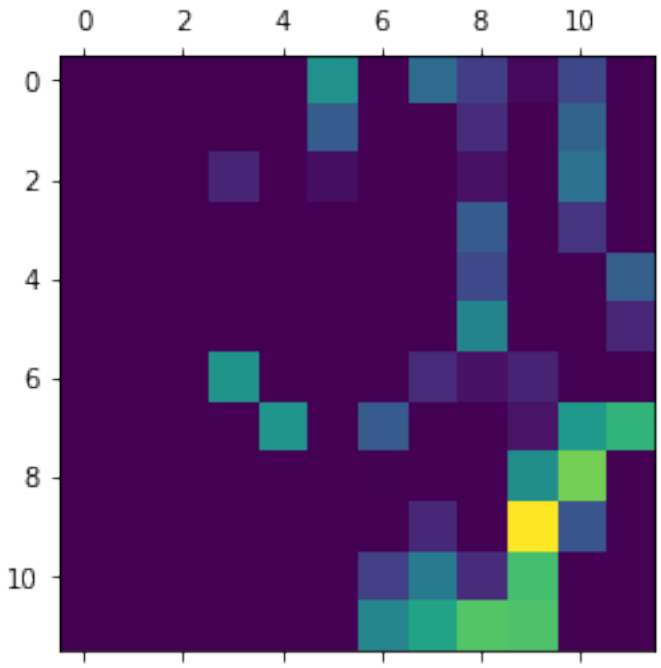
Filtro 16 de 32



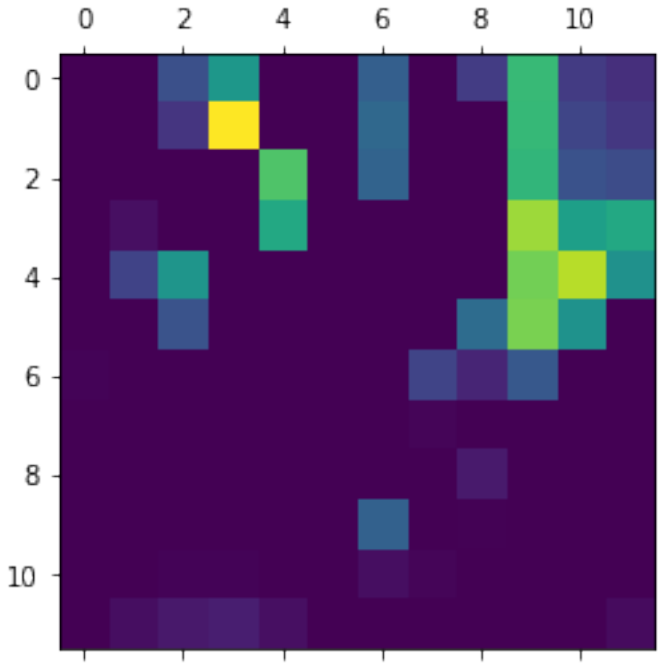
Filtro 17 de 32



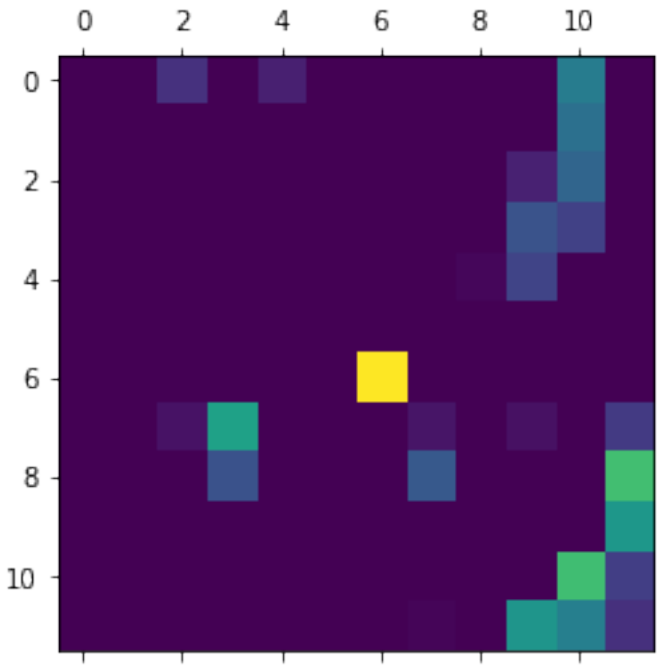
Filtro 18 de 32



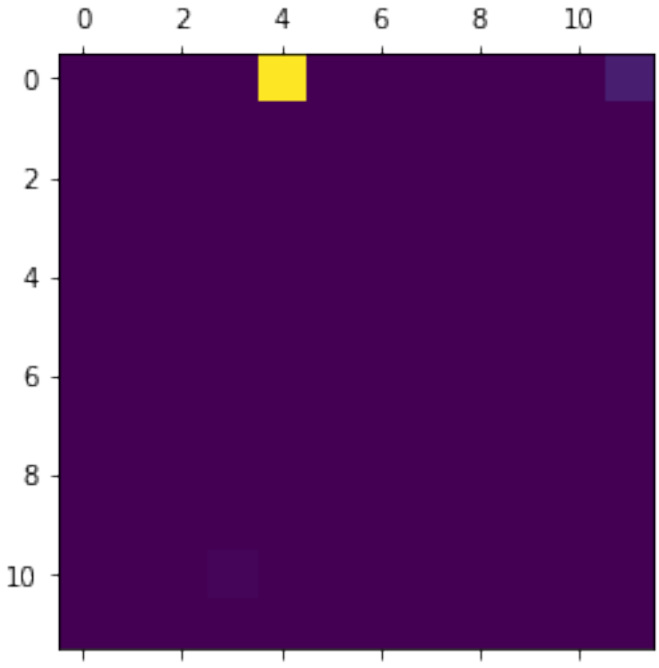
Filtro 19 de 32



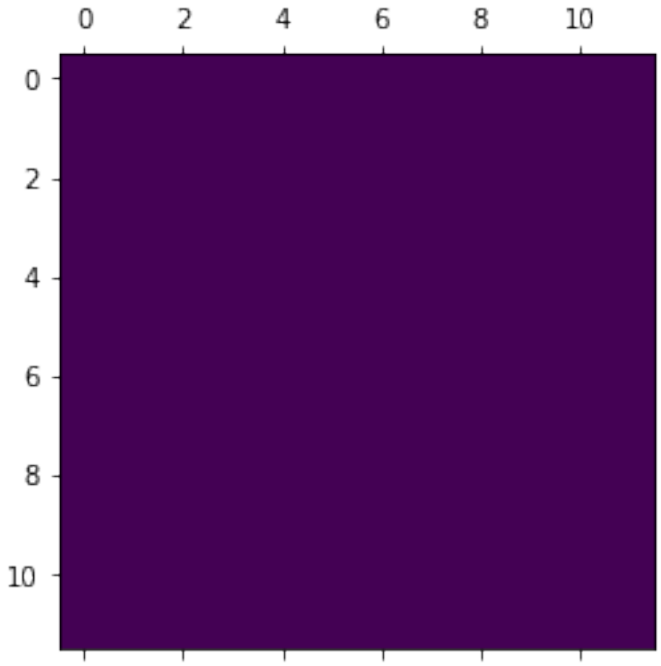
Filtro 20 de 32



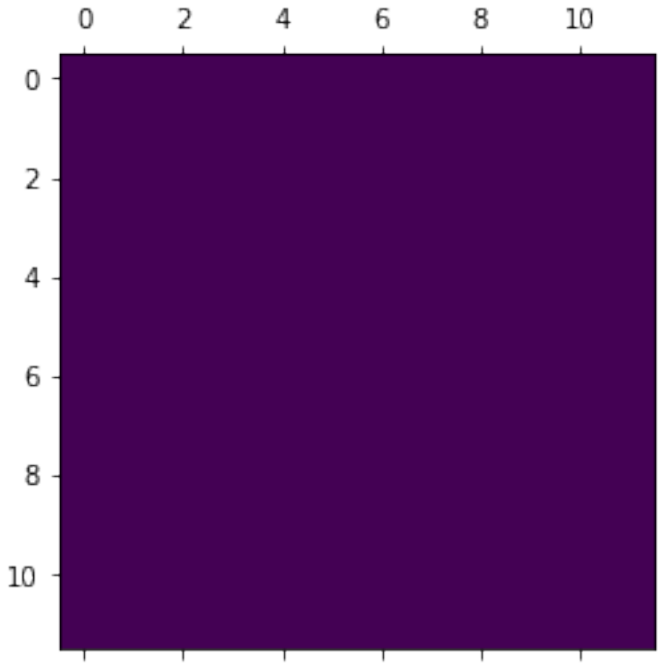
Filtro 21 de 32



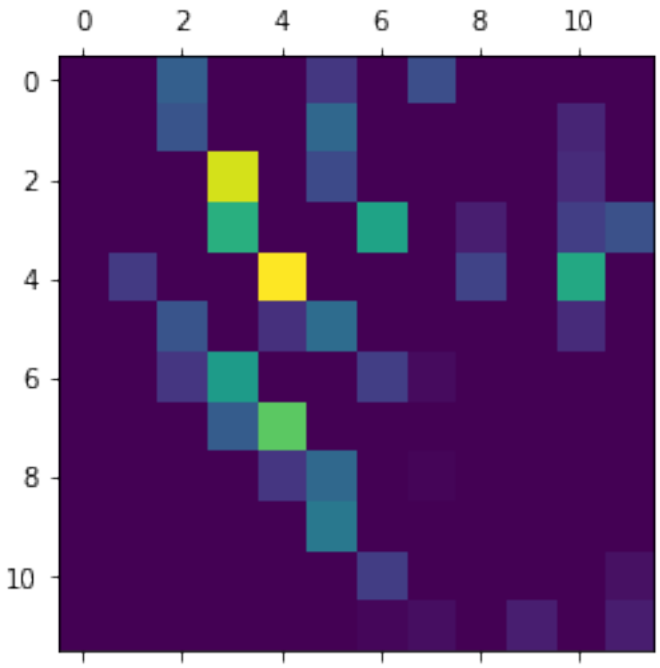
Filtro 22 de 32



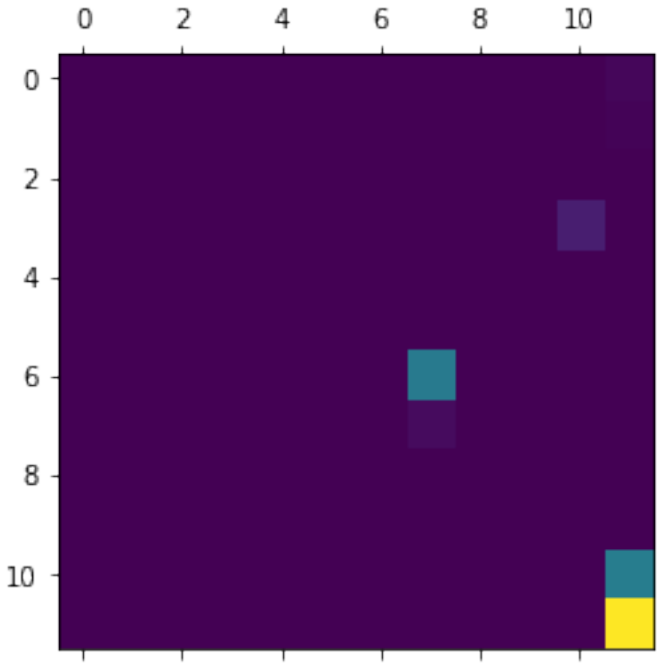
Filtro 23 de 32



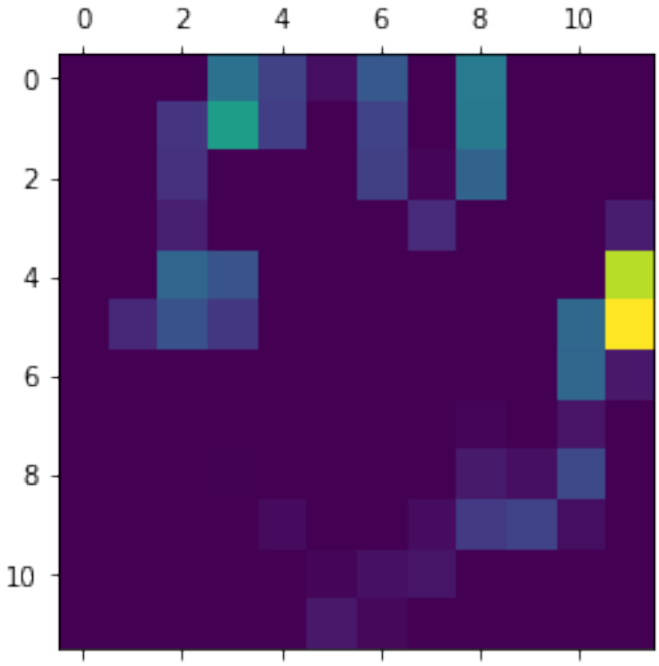
Filtro 24 de 32



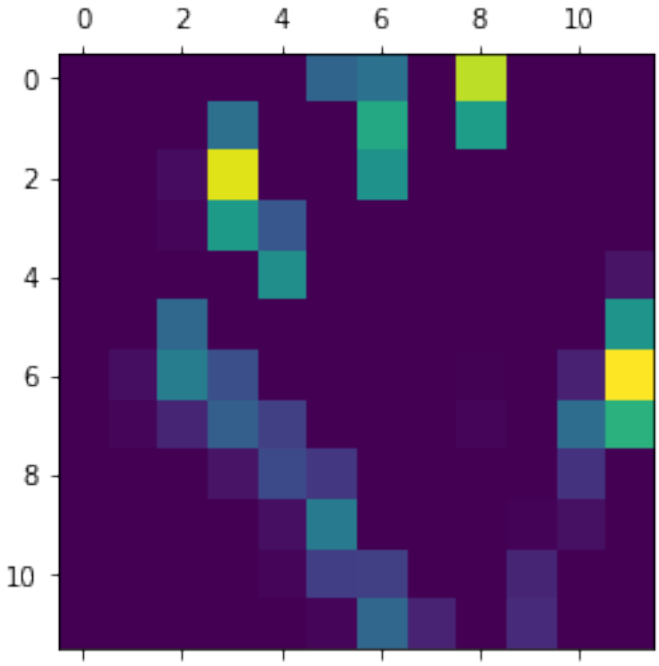
Filtro 25 de 32



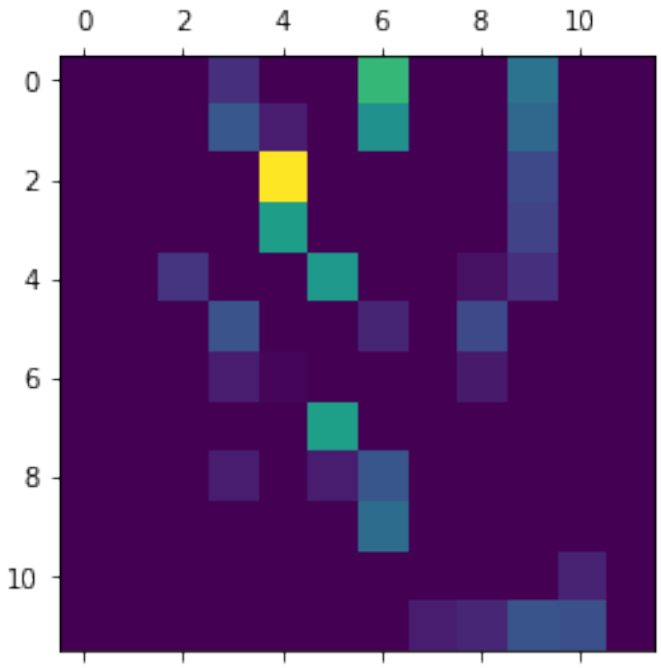
Filtro 26 de 32



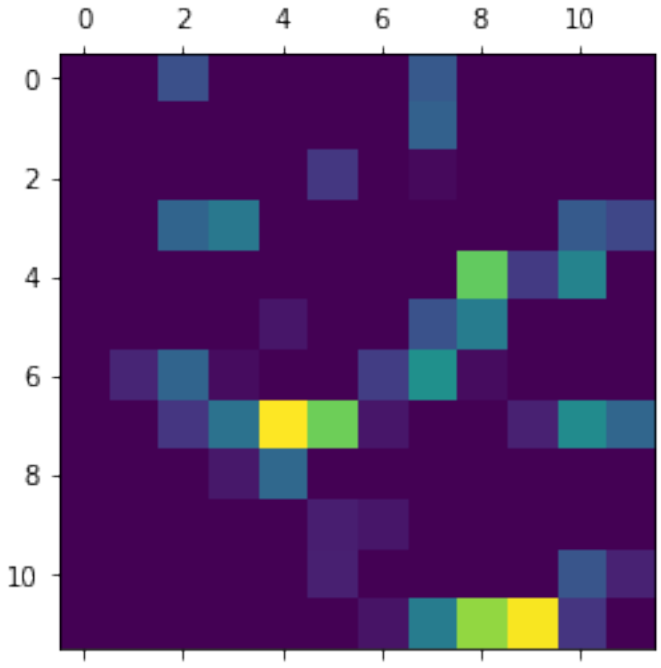
Filtro 27 de 32



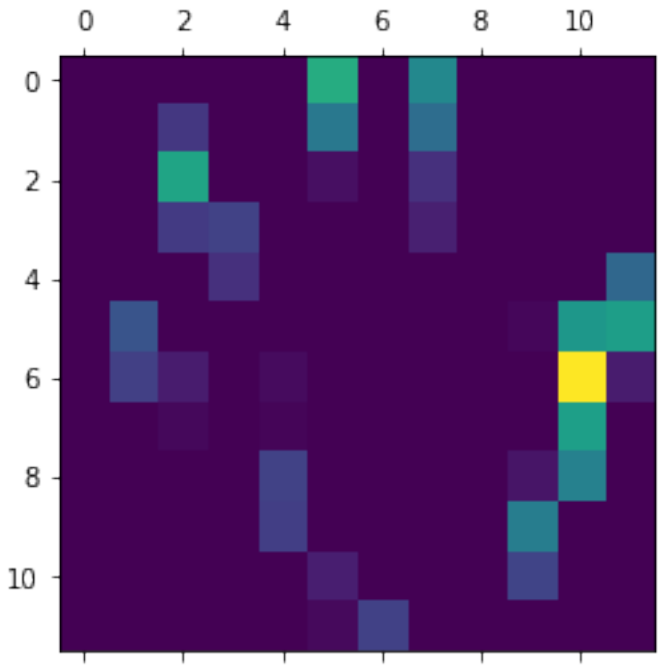
Filtro 28 de 32



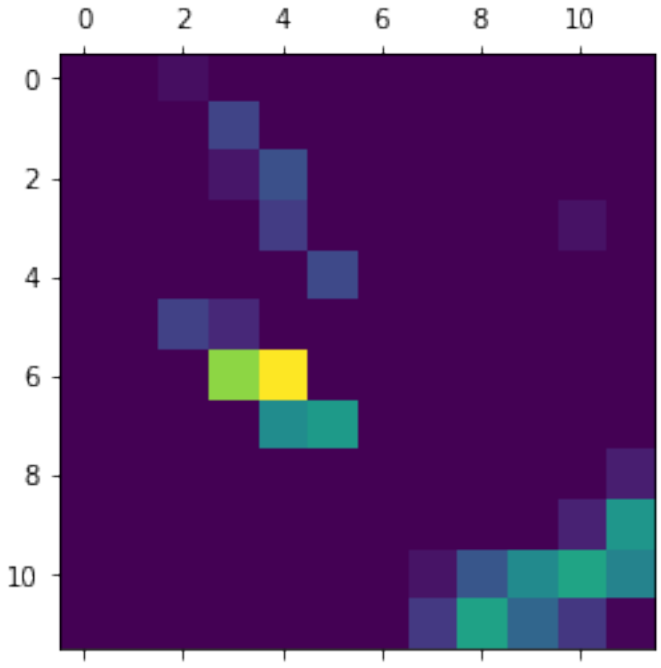
Filtro 29 de 32



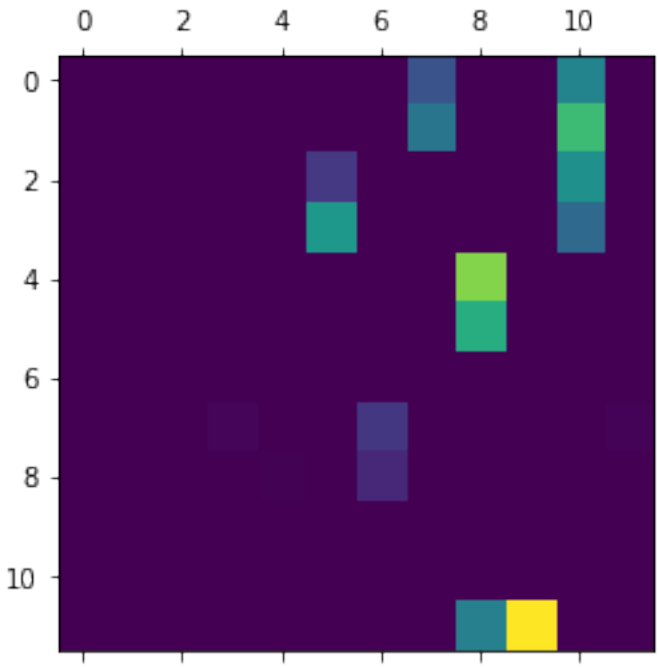
Filtro 30 de 32



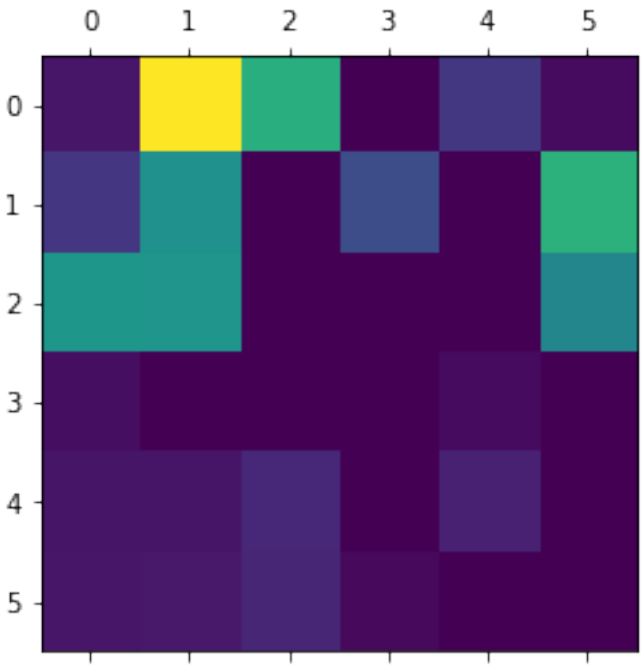
Filtro 31 de 32



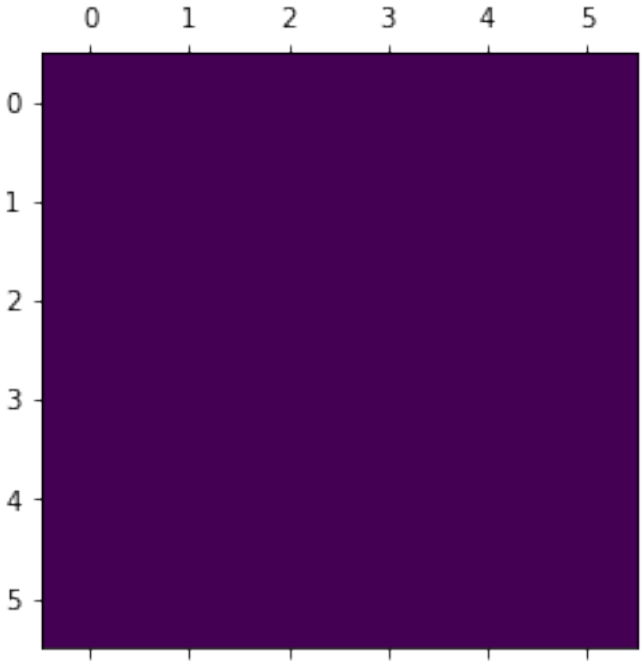
```
Filtro 32 de 32
```



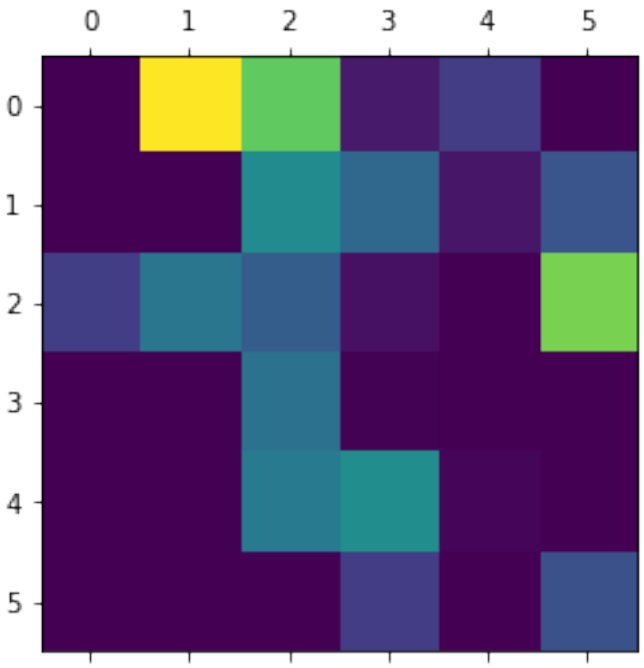

```
Camada 6
-----
Filtro 1 de 32
```



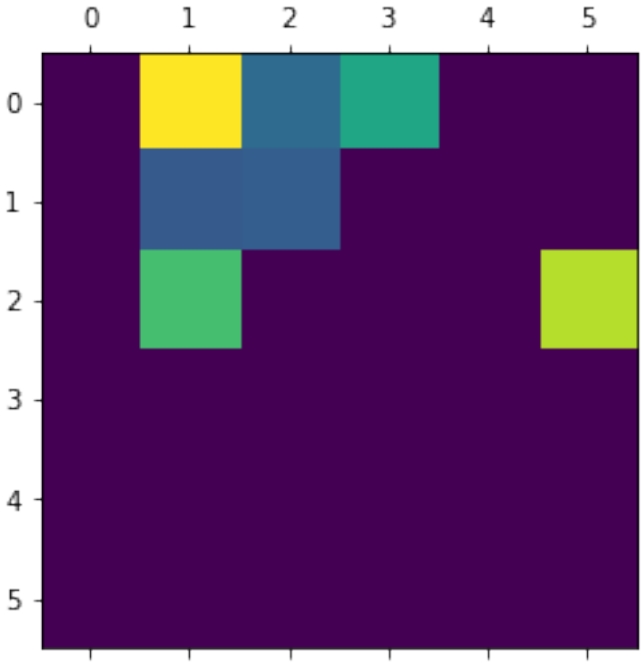
Filtro 2 de 32



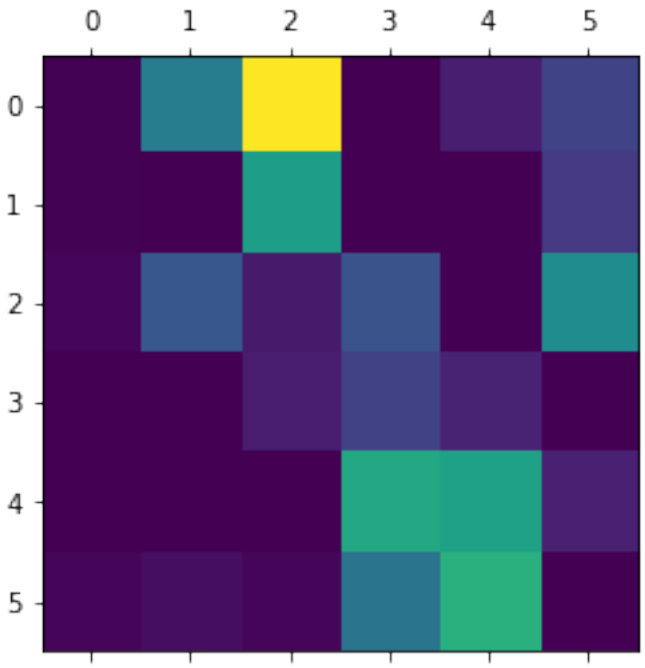
Filtro 3 de 32



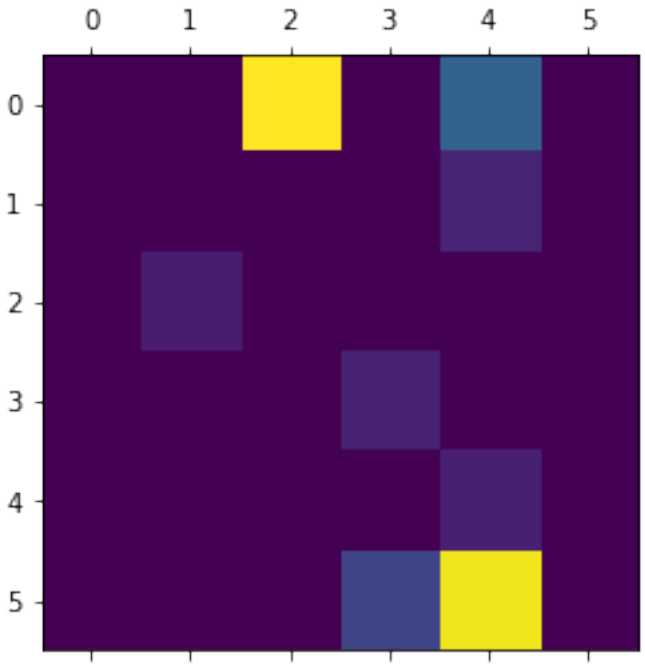
Filtro 4 de 32



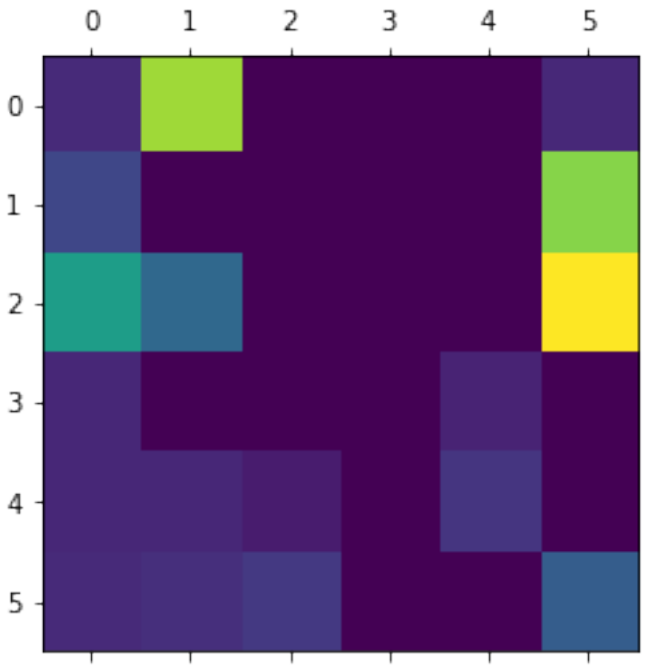
Filtro 5 de 32



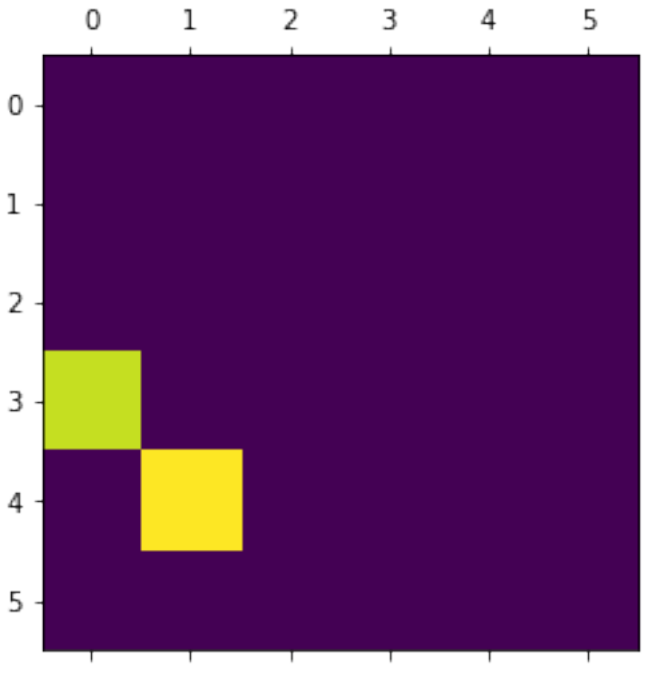
Filtro 6 de 32



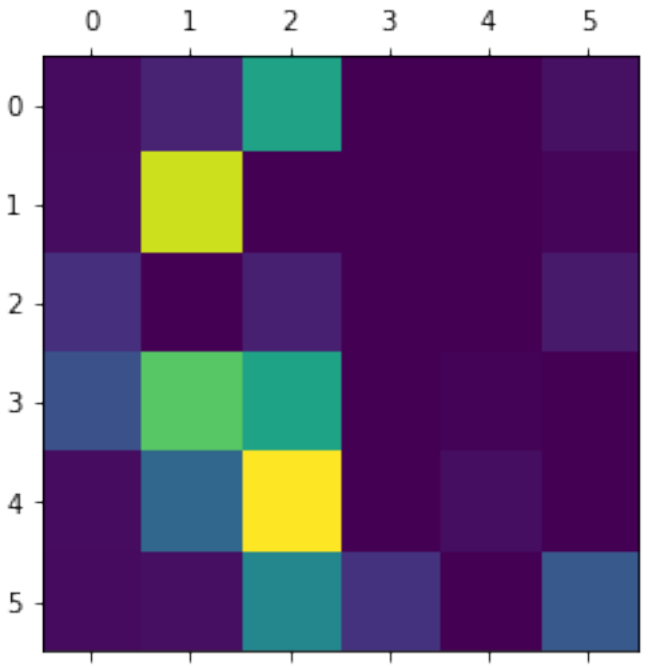
Filtro 7 de 32



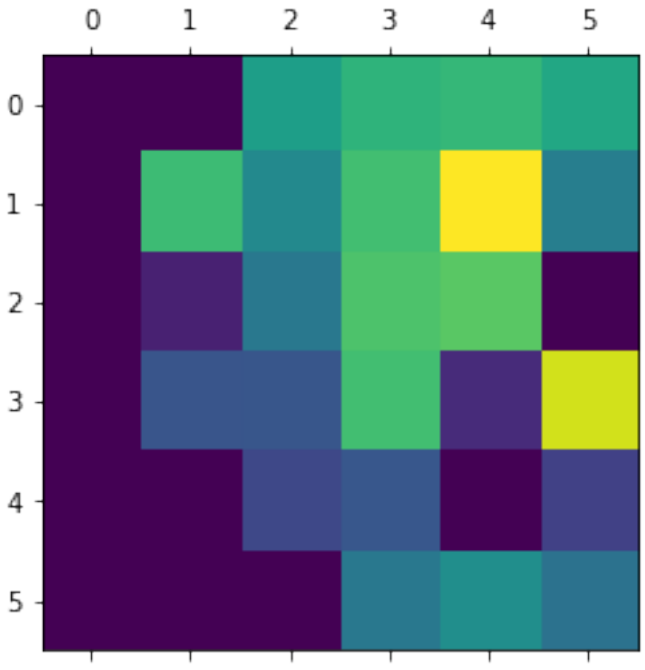
Filtro 8 de 32



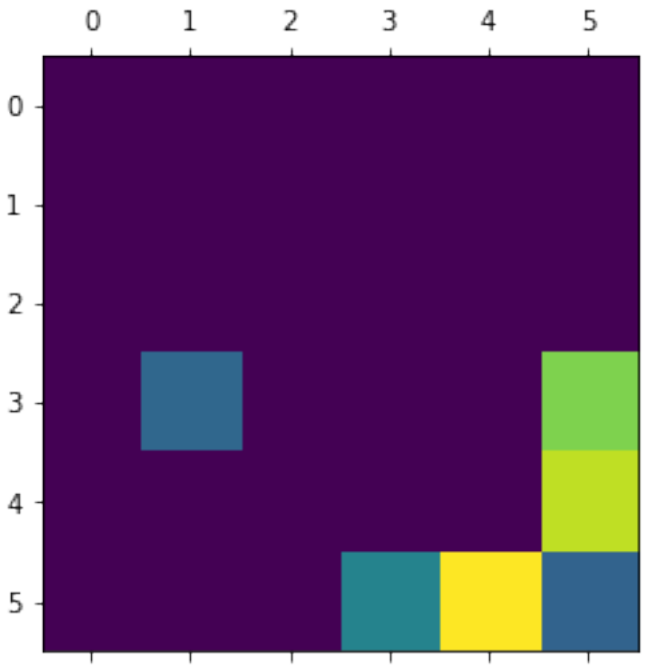
Filtro 9 de 32



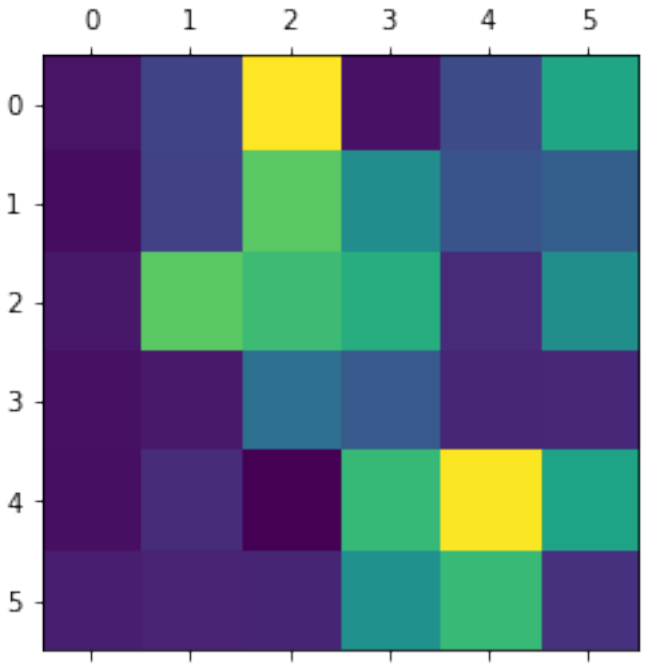
Filtro 10 de 32



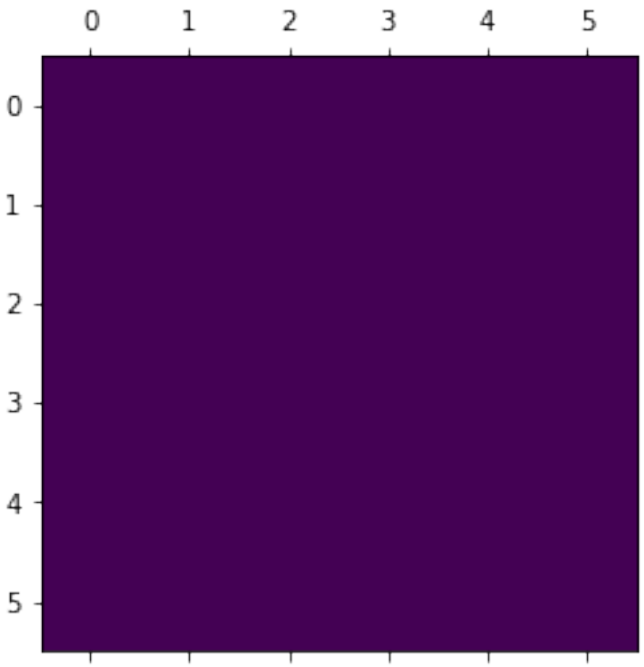
Filtro 11 de 32



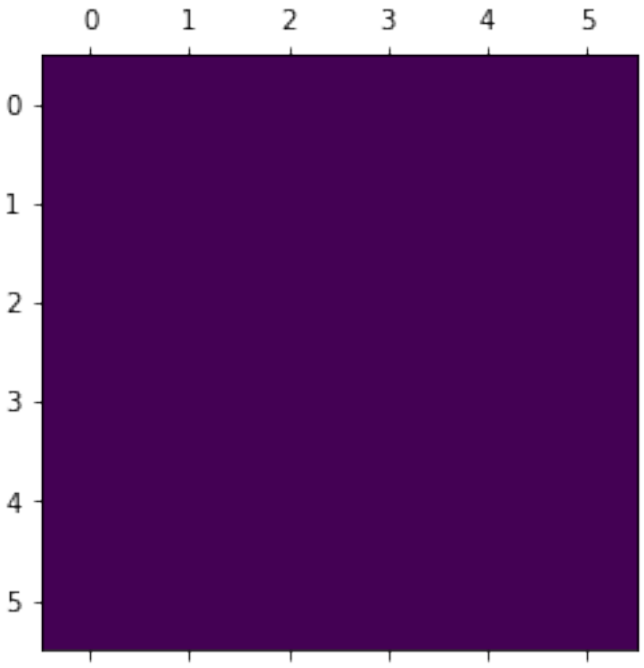
Filtro 12 de 32



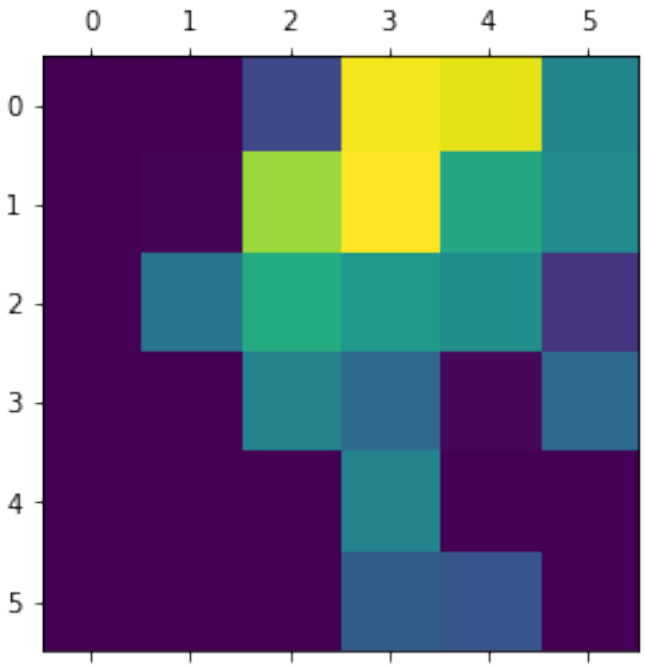
Filtro 13 de 32



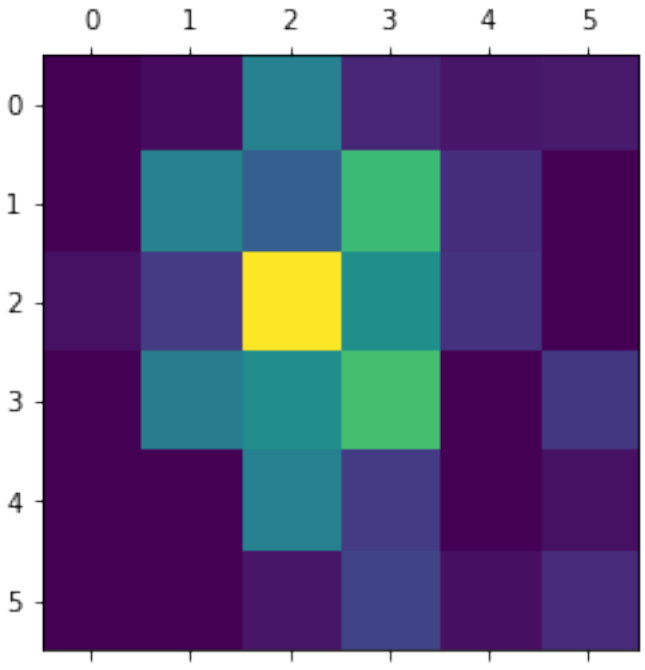
Filtro 14 de 32



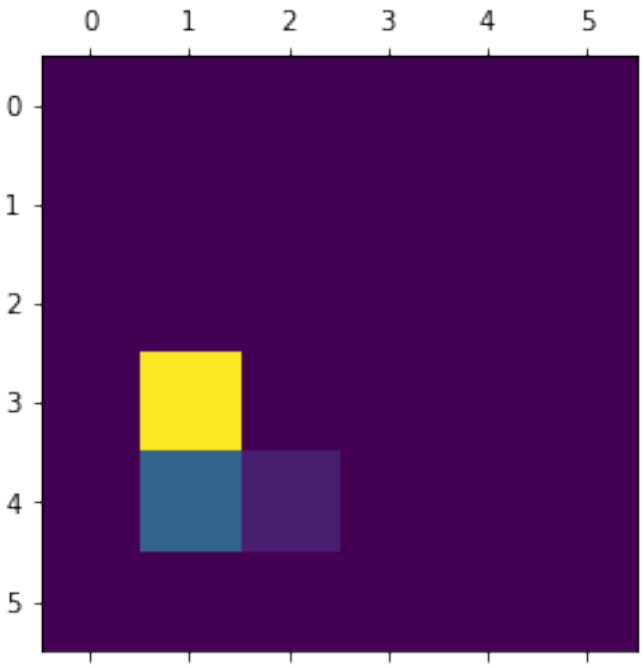
Filtro 15 de 32



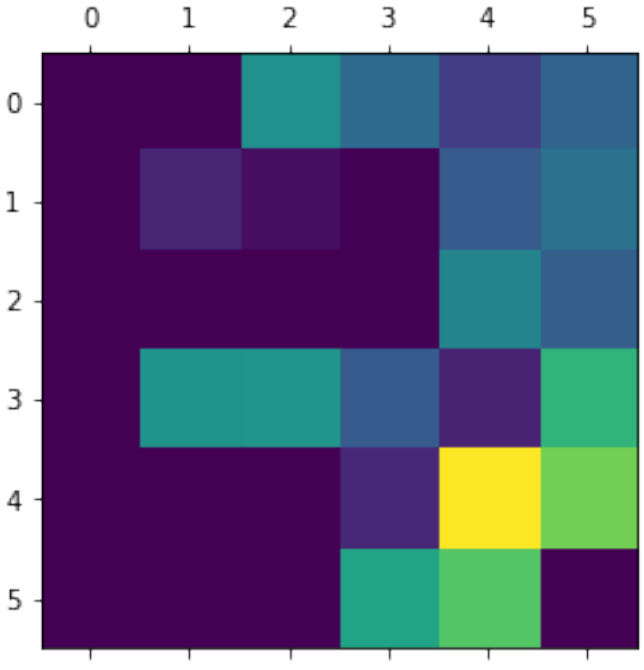
Filtro 16 de 32



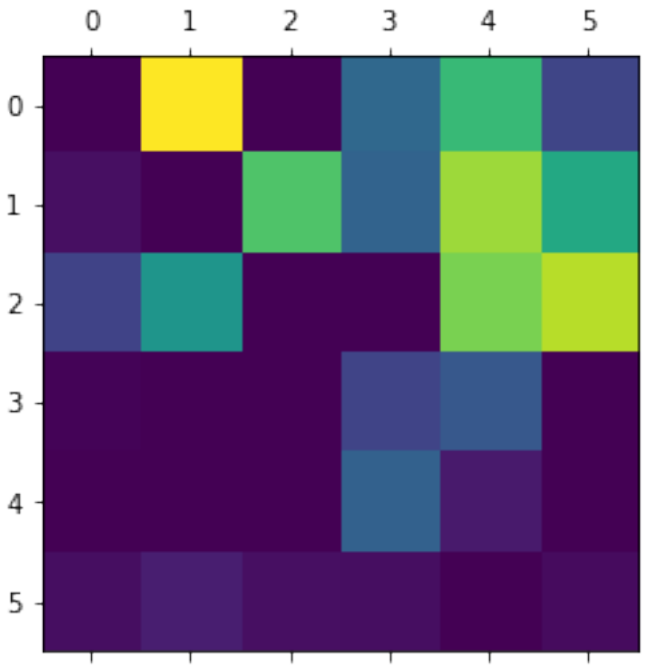
Filtro 17 de 32



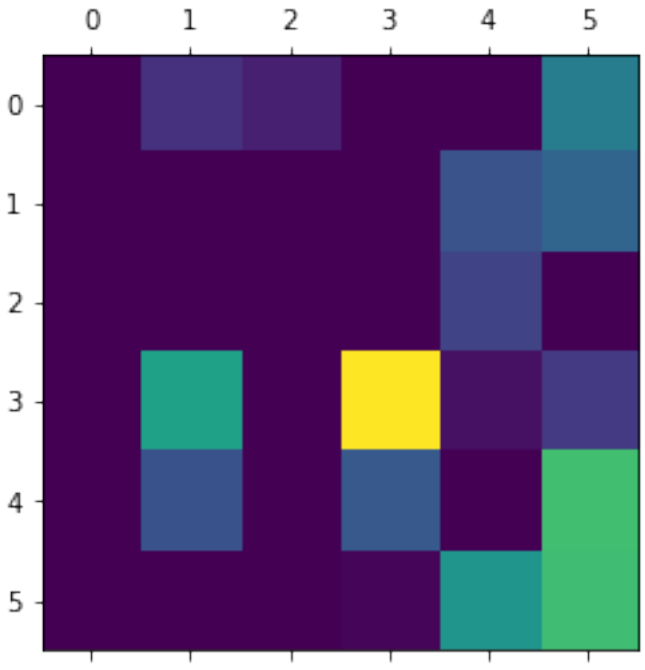
Filtro 18 de 32



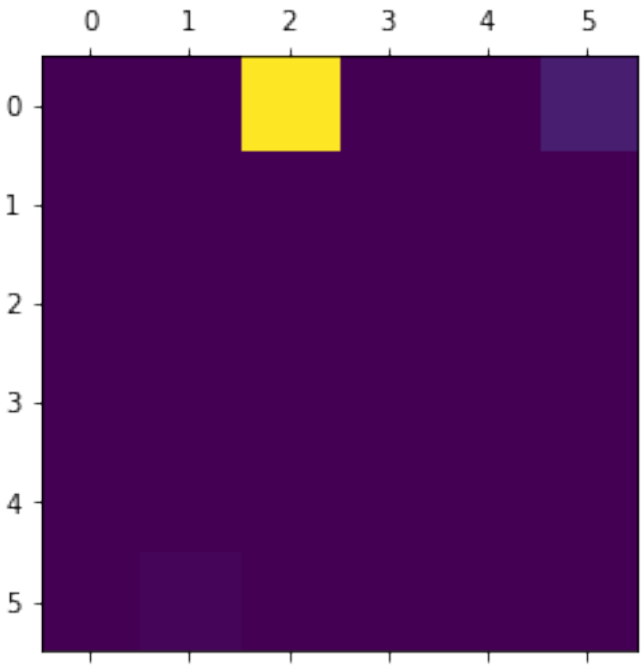
Filtro 19 de 32



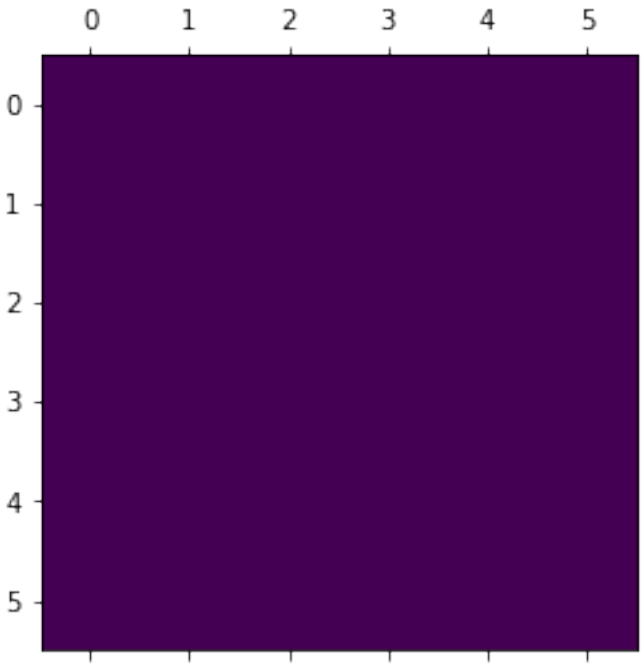
Filtro 20 de 32



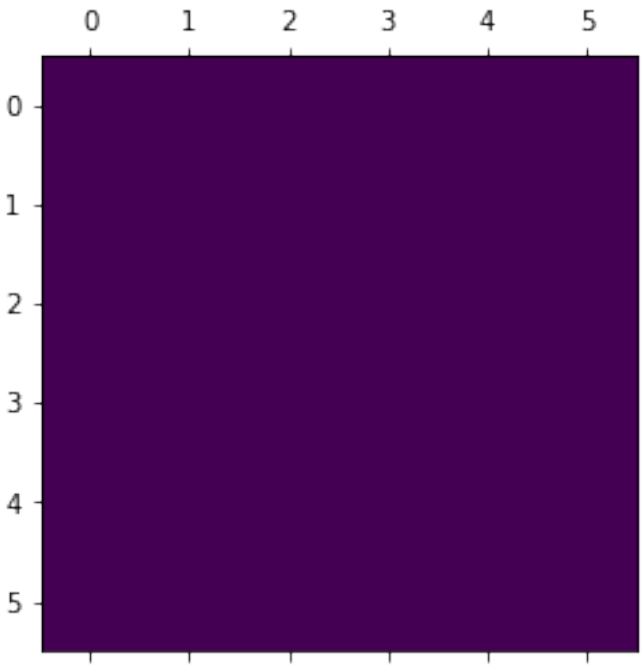
Filtro 21 de 32



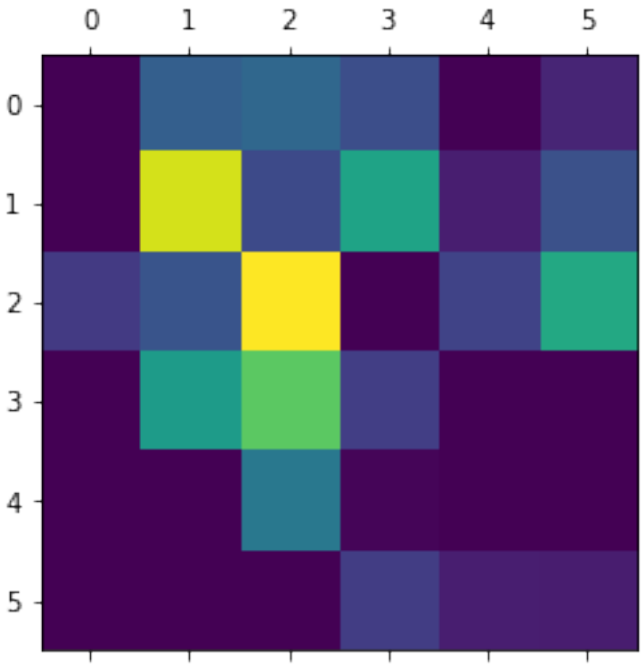
Filtro 22 de 32



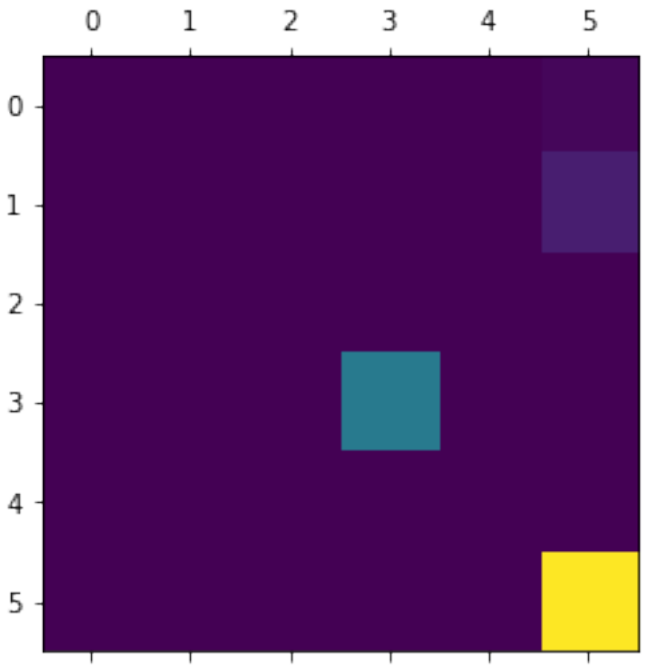
Filtro 23 de 32



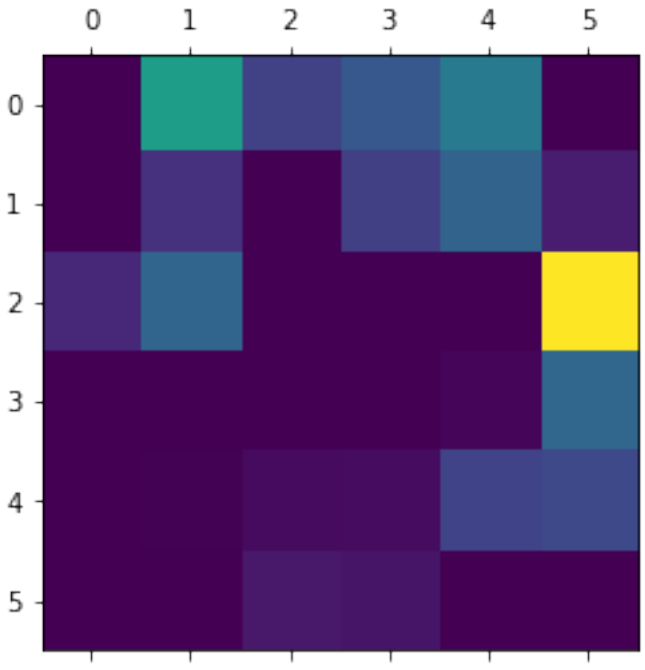
Filtro 24 de 32



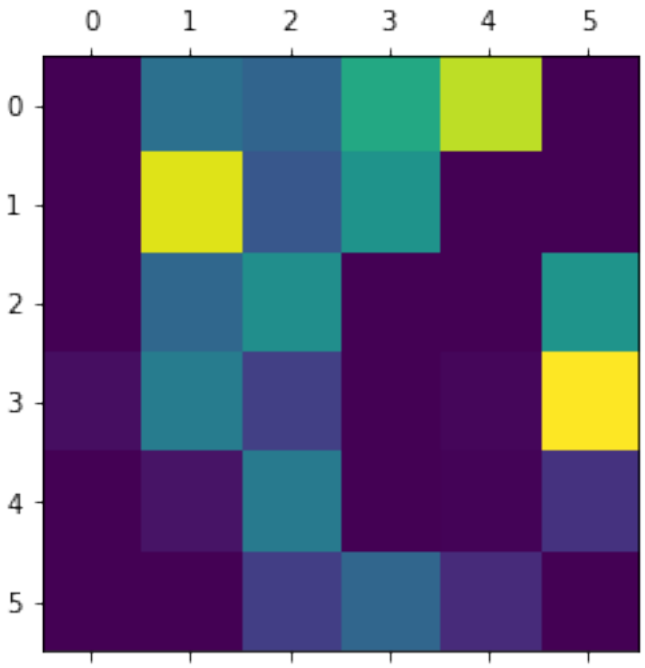
Filtro 25 de 32



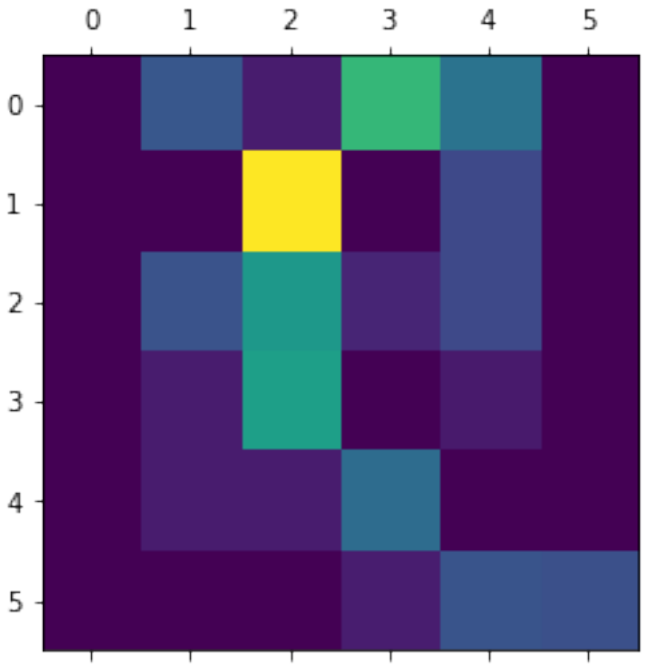
Filtro 26 de 32



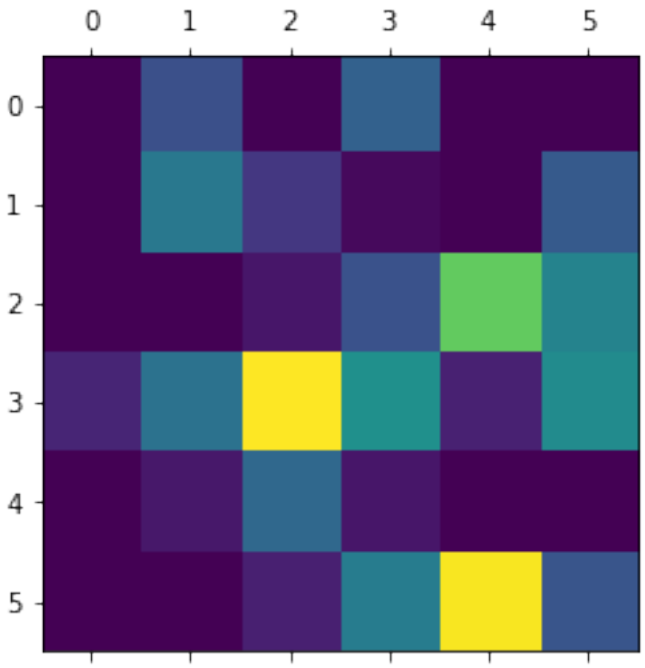
Filtro 27 de 32



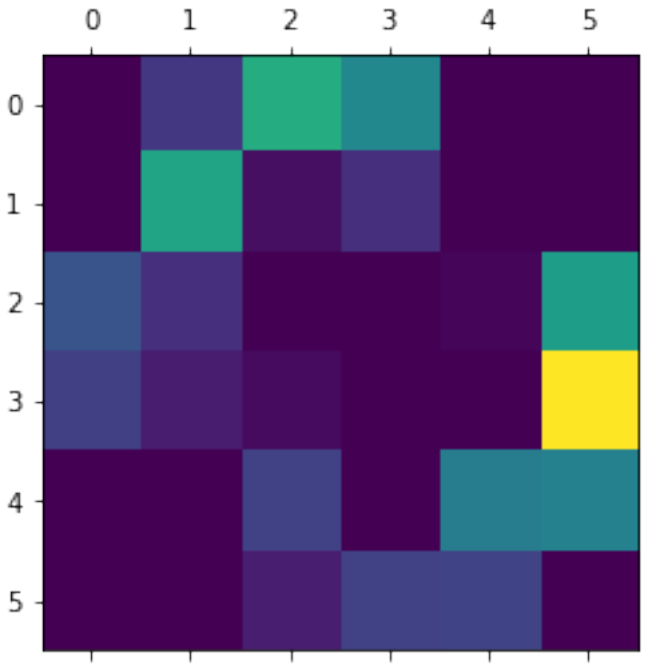
Filtro 28 de 32



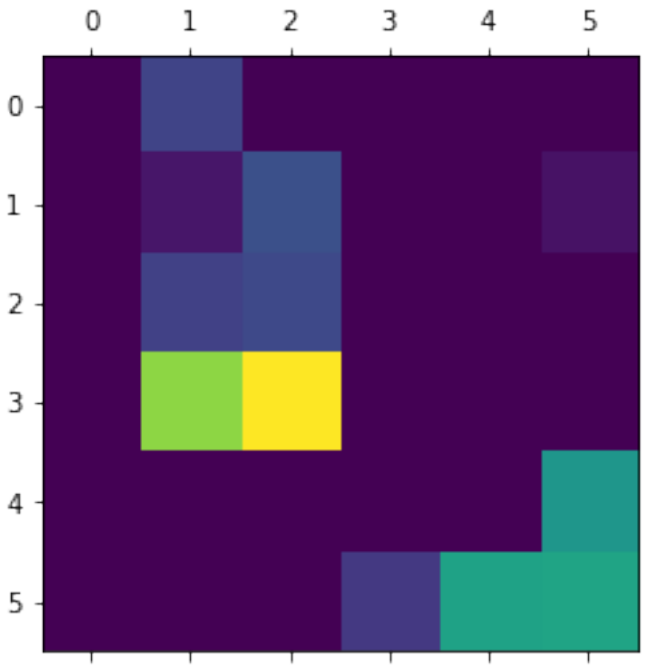
Filtro 29 de 32



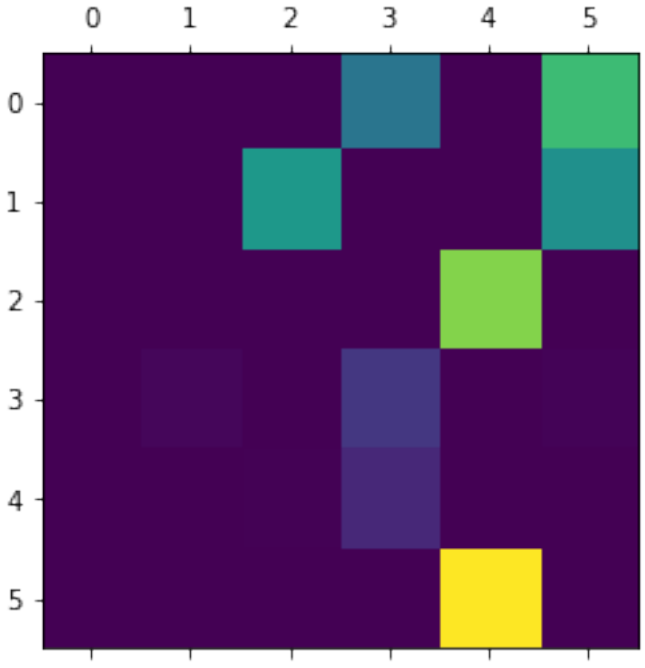
Filtro 30 de 32



Filtro 31 de 32



Filtro 32 de 32



4 Conclusões

A partir desses resultados você pode concluir o seguinte:

- Em geral as primeiras camadas de uma RNA convolucional agem como uma coleção de detectores de vários tipos de bordas.
- Nas primeiras camadas a ativações contém quase toda a informação presente na imagem original.
- Na medida em que avançamos para dentro da rede, as ativações se tornam mais abstratas e com menor significado visual e começam a codificar características de alto nível.
- Características de níveis mais alto contém menos informação visual e mais informações relacionadas com a tarefa a ser realizada.
- A não ativação de filtros para uma determindada imagem aumenta com a profundidade da camada: na 1ª camada praticamente todos os filtros são ativados, mas nas camadas mais profundas menos filtros ficam ativos.
- Quando um filtro não é ativado por uma imagem, significa que o padrão codificado por aquele filtro não está presente naquela imagem.
- Uma característica importante das RNAs convolucionais deep learning é que as características aprendidas pelas suas camadas se tornam cada vez mais abstratas com a profundidade da camada.
- Uma RNA deep learning age efetivamente como um destilador de informação, onde dados brutos são repetidamente transformados de forma que informações irrelevantes são descartadas e informações importantes são ressaltadas e refinadas.

Importante:

- Observe que essa RNA possui somente 82.102 parâmetros, enquanto que se fosse usada uma RNA densa seriam necessário um número muito maior de parâmetros.
- Mesmo com um número pequeno de parâmetros a RNA convolucional é capaz de obter resultados muito bons.