Jedi vs. Sith vs. Neither

July 7, 2022

1 Jedi vs. Sith vs. Neither

A ternary image classifier example

1.1 Goal

The goal of this notebook is to demonstrate the functionality of a ternary image classifier network.

1.2 Architecture

This neural network will use a Convolutional architecture on the Keras neural network framework with TensorFlow as the backend. After training, it will classify images as either 'Jedi' or 'Sith' or 'Neither'.

[1]:
```
# Set up TensorFlow
import tensorflow as tf

gpu_devices = tf.config.experimental.list_physical_devices('GPU')
for device in gpu_devices:
    tf.config.experimental.set_memory_growth(device, True)
```

[2]:
```
import numpy as np
import os
import PIL
import PIL.Image
import pathlib
import random
```

1.3 Global Parameters

The global parameters are batch size (32) and image size (400x400).

[3]:
```
batch_size = 32
img_height = 400
img_width = 400
```

1.4 Training Dataset

We have a training dataset of hand-curated images in three subdirectories called "jedi", "sith", and "neither". We'll load them here.

1.4.1 Data Quality

The dataset I'm using came from Kaggle, here: Star Wars Images | Kaggle.

Note that Obi-Wan Kenobi is, for some reason, split into "Kenobi" and "Obi-Wan" directories. Note also that there are non-Star Wars images in some of the directories. For example, there are pictures of Edgar Allan Poe in Poe Dameron's directory. Inexplicably, there's a picture of Hermione Granger in Orson Krennic's directory.

I'm leaving these in as neither Edger Allan Poe, nor Hermione Granger are Jedi or Sith, therefore they can reasonably be called "neither".

Rey has the unfortunate coincidence of sharing her name with a famous *luchador* so there are photos of him in her directory. Those I'm removing because El Rey is *not* a Jedi. I've also removed all the photos of Finn from his Stormtrooper days since he's indistinguishable from any other Stormtrooper when wearing his helmet and I didn't want to confuse the model.

[4]:
```
data_dir = pathlib.Path('training_set_ternary')

jedi = list(data_dir.glob('jedi/*'))
sith = list(data_dir.glob('sith/*'))
neither = list(data_dir.glob('neither/*'))
num_jedi = len(jedi)
num_sith = len(sith)
num_neither = len(neither)
total_count = num_jedi + num_sith + num_neither

print(num_jedi, 'Jedi images, ', num_sith, 'Sith images, and ', num_neither,␣
 ↪'Neither images')
```

```
745 Jedi images,  637 Sith images, and  3142 Neither images
```

[5]:
```
print('A Jedi looks like this:')
PIL.Image.open(jedi[random.randint(0,num_jedi)])
```

```
A Jedi looks like this:
```

[5]:

[6]:
```
print('A Sith looks like this:')
PIL.Image.open(sith[random.randint(0,num_sith)])
```

```
A Sith looks like this:
```

[6]:

[7]:
```
print('These people are neither Jedi nor Sith:')
display(PIL.Image.open(neither[random.randint(0,num_neither)]))
display(PIL.Image.open(neither[random.randint(0,num_neither)]))
display(PIL.Image.open(neither[random.randint(0,num_neither)]))
```

```
These people are neither Jedi nor Sith:
```


GALACTIC SENATOR
BAIL ORGANA
copyright by Frank Hnatek (frank-hnatek@starwars-union.de) ... April 2005

[8]:
```
# Create a dataset from the input images, using 80% for training and 20% for␣
↪validation
train_ds = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    validation_split=0.2,
    subset='training',
    seed=213,
    image_size=(img_height,img_width),
    batch_size=batch_size)
```

```
class_names = train_ds.class_names
```

```
Found 4524 files belonging to 3 classes.
Using 3620 files for training.
```

```
2022-07-07 10:50:09.408742: I tensorflow/core/platform/cpu_feature_guard.cc:151]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2022-07-07 10:50:10.281023: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1525] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 4430 MB memory:  -> device: 0,
name: NVIDIA GeForce RTX 2060 SUPER, pci bus id: 0000:01:00.0, compute
capability: 7.5
```

[9]:
```
# Validation dataset
val_ds = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    validation_split=0.2,
    subset='validation',
    seed=213,
    image_size=(img_height,img_width),
    batch_size=batch_size)
```

```
Found 4524 files belonging to 3 classes.
Using 904 files for validation.
```

1.5 I/O Tuning

We'll use the AUTOTUNE feature of `tf.data` to tune the I/O performance parameters.

[10]:
```
AUTOTUNE = tf.data.AUTOTUNE

train_ds = train_ds.cache().prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

1.6 Model

The network model is a sequence of layers of convolutions alternating with max pooling functions. Each convolution layer produces a feature map as an output. The max pooling layers reduce the size of each feature map to maximize the magnitude of the observed features. The final layers are "fully-connected" layers, meaning they form a proper deep neural net.

Conceptually, we can think of it as a two-layer model: Convolutions to produce feature maps, which are exaggerated impressions of prominent features of the image, and a neural net that looks at the features and finds patterns in them.

[11]:
```
model = tf.keras.Sequential()
```

1.6.1 Layer 0: Data Augmentation

Since we have so few training images to work with, we will use some data augmentation techniques built into TensorFlow to pad out our learning dataset.

The first layer is an input layer. Since our images are 400x400px color images, we need an input tensor shape of 400x400x3. This layer will randomly flip the image horizontally, vertically, both, or none.

The second layer will randomly rotate the image left or right by up to 20%.

The third layer will randomly adjust the contrast by 20%.

The fourth layer will randomly zoom into a part of the image by 20-30%.

[12]:
```
model.add(tf.keras.layers.RandomFlip("horizontal_and_vertical",
                                     input_shape=(img_height, img_width, 3)))
model.add(tf.keras.layers.RandomRotation(0.2))
model.add(tf.keras.layers.RandomContrast(0.2))
model.add(tf.keras.layers.RandomZoom(height_factor=(-0.3, -0.2)))
```

1.6.2 Layer 0.5 Data Normalization

Neurons operate best on values between 0 and 1. Let's remap the color values to that domain from their current range of 0-255.

[13]:
```
model.add(tf.keras.layers.Rescaling(1./255))
```

1.6.3 Layer 1: Input Convolution

We'll use a ReLU activation function so we deliberately fail to backpropagate any negative bias, which saves us a bit of computational overhead. We'll use 16 filters to start with and a 3x3 kernel, which are good values to use unless you have good reason to use other ones (see many, many papers on why 3x3 is the optimal kernel size for pretty much any CNN).

[14]:
```
model.add(tf.keras.layers.Conv2D(16,
                                 (3,3),
                                 activation='relu'))
```

1.6.4 Layer 2: Dropout

To reduce overfitting, we will randomly knock out 1/5 of the neurons in the network.

[15]:
```
model.add(tf.keras.layers.Dropout(rate=0.2))
```

1.6.5 Layer 3: Max Pooling

Takes the output of the previous layer (a feature map) and pools each quadrant, taking the maximum value thereof.

[16]:
```
model.add(tf.keras.layers.MaxPooling2D(2, 2))
```

1.6.6 Layer 4: Second Convolution

Convolves the maximized pooled feature map from Layer 1, using 32 filters this time, and still using a ReLU activation function to eliminate negative values.

[17]:
```
model.add(tf.keras.layers.Conv2D(32, (3,3), activation='relu'))
```

1.6.7 Layer 5: Max Pooling

Another max pooling layer to separate the convolutions like thin paper sheets separate deli slices.

[18]:
```
model.add(tf.keras.layers.MaxPooling2D(2, 2))
```

1.6.8 Layer 6: Third Convolution

This time, we're bumping the filters up to 64, because want to learn more from less input data.

[19]:
```
model.add(tf.keras.layers.Conv2D(64, (3,3), activation='relu'))
```

1.6.9 Layer 7: Max Pooling

You guessed it, another max pooling layer

[20]:
```
model.add(tf.keras.layers.MaxPooling2D(2, 2))
```

1.6.10 Layer 8: Fourth Convolution

Same as the third.

[21]:
```
model.add(tf.keras.layers.Conv2D(64, (3,3), activation='relu'))
```

1.6.11 Layer 9: Max Pooling

Lather, rinse, repeat.

[22]:
```
model.add(tf.keras.layers.MaxPooling2D(2, 2))
```

1.6.12 Layer 10: Fifth Convolution

Same as the third and fourth.

[23]:
```
model.add(tf.keras.layers.Conv2D(64, (3,3), activation='relu'))
```

1.6.13 Layer 11: Max Pooling

Last one.

[24]:
```
model.add(tf.keras.layers.MaxPooling2D(2, 2))
```

1.6.14 Layer 12: Flattening

The output shape of the last max pooling layer is 10x10x64. Since we want to learn from our feature map by using a Dense Neural Net layer, we need to flatten our 3 dimensional tensor into a 1-dimensional vector. 10 times 10 is 100 and 100 times 64 is 6400. Therefore, the input to our DNN will be a vector containing 6400 terms.

[25]:
```
model.add(tf.keras.layers.Flatten())
```

1.6.15 Layer 13: Dropout

To reduce overfitting, we will randomly knock out 1/5 of the neurons in the network.

[26]:
```
model.add(tf.keras.layers.Dropout(rate=0.2))
```

1.6.16 Layer 14: Dense Neural Net

This is the layer that actually *makes inferences* about the input data, recognizing connections between data points. We're again using ReLU as the activation function to prevent backpropagation of negative bias, which adds complexity to the computation. We're using 3,201 neurons this time. (Input is 6400 data points, output is 3 data points, mean of those two is 3201.)

[27]:
```
model.add(tf.keras.layers.Dense(3201, activation='relu'))
```

1.6.17 Layer 15: Dropout

To reduce overfitting, we will randomly knock out 1/5 of the neurons in the network.

[28]:
```
model.add(tf.keras.layers.Dropout(rate=0.2))
```

1.6.18 Layer 16: Ouput DNN

Our final layer is a dense neural network consisting of three neurons. This layer takes the 3201 values from the previous layer and makes the final call, applying a softmax function to represent the probability of membership in each of the three possible classes.

(Fun fact: we really should be using an argmax function, but the discontinuity causes the derivative of the function to be zero, which would mean that the weight fails to update during backpropagation, so we use a "softer" maximum function instead, which accomplishes the same thing after rounding.)

[29]:
```
model.add(tf.keras.layers.Dense(3, activation='softmax'))
```

1.6.19 Model Summary

16 layers this time (we added some dropouts and augmentation layers) and ready to rock!

[30]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 random_flip (RandomFlip)    (None, 400, 400, 3)       0

 random_rotation (RandomRota  (None, 400, 400, 3)      0
 tion)

 random_contrast (RandomCont  (None, 400, 400, 3)      0
 rast)

 random_zoom (RandomZoom)    (None, 400, 400, 3)       0

 rescaling (Rescaling)       (None, 400, 400, 3)       0

 conv2d (Conv2D)             (None, 398, 398, 16)      448

 dropout (Dropout)           (None, 398, 398, 16)      0

 max_pooling2d (MaxPooling2D  (None, 199, 199, 16)     0
 )

 conv2d_1 (Conv2D)           (None, 197, 197, 32)      4640

 max_pooling2d_1 (MaxPooling  (None, 98, 98, 32)       0
 2D)

 conv2d_2 (Conv2D)           (None, 96, 96, 64)        18496

 max_pooling2d_2 (MaxPooling  (None, 48, 48, 64)       0
 2D)

 conv2d_3 (Conv2D)           (None, 46, 46, 64)        36928

 max_pooling2d_3 (MaxPooling  (None, 23, 23, 64)       0
 2D)

 conv2d_4 (Conv2D)           (None, 21, 21, 64)        36928

 max_pooling2d_4 (MaxPooling  (None, 10, 10, 64)       0
 2D)
```

```
 flatten (Flatten)           (None, 6400)              0

 dropout_1 (Dropout)         (None, 6400)              0

 dense (Dense)               (None, 3201)              20489601

 dropout_2 (Dropout)         (None, 3201)              0

 dense_1 (Dense)             (None, 3)                 9606

=================================================================
Total params: 20,596,647
Trainable params: 20,596,647
Non-trainable params: 0
_________________________________________________________________
```

1.7 Detour into Feature Maps

We're going to create a second model that outputs the feature maps from each of the convolution layers when we ask it to predict.

[31]:
```
conv_layer_idx = [5, 8, 10, 12, 14]
feature_outputs = [model.layers[i].output for i in conv_layer_idx]
feature_model = tf.keras.Model(inputs=model.inputs, outputs=feature_outputs)
```

1.7.1 Class Weights

An additional hyperparameter for this network will be relative class weights. Without this, the model has nearly a 3 to 1 chance of predicting 'neither' before even *looking* at an image. Adding relative weighting evens those odds a bit.

[32]:
```
jedi_weight = (1 / num_jedi) * (total_count / 2.0)
neither_weight = (1 / num_neither) * (total_count / 2.0)
sith_weight = (1 / num_sith) * (total_count / 2.0)

class_weight = {0: jedi_weight, 1: neither_weight, 2: sith_weight}
```

1.8 Training

Our model will be trained by minimizing the cross-entropy loss function

Cross-entropy loss function:

$$H(p, q) = -\sum_i p_i \log q_i = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

where y is the dependent variable, per category ("Jedi-ness", "Sith-ness", or "Neither-ness"), p is the probability that the image shows a member of said category, and q is the *estimated* probability

that the image shows a member of said category. The formula relates the magnitude (H) of how far off the mark the guess (q) was from the real value (p).

See Cross entropy to better understand why. We will be applying the ~~Root Mean Square Propagation (RMSProp) optimizer~~ Adam optimizer function to adjust learning rate dynamically, because naïve RMSE propagation would be unduly affected by our class weighting, **w**, and we want it to be neutral. Adam does that for us. It's basically RMSProp with an added momentum term and continually-updated bias correction.

[33]:
```
model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy'])
```

1.8.1 Train yourself to let go of everything you fear to lose.

A more complex problem needs more training. We'll train for 95 epochs.

[34]:
```
history = model.fit(
    train_ds,
    epochs=95,
    verbose=1,
    validation_data=val_ds,
    class_weight=class_weight,
)
```

```
Epoch 1/95

2022-07-07 10:50:13.651423: I tensorflow/stream_executor/cuda/cuda_dnn.cc:368]
Loaded cuDNN version 8101

114/114 [==============================] - 22s 142ms/step - loss: 1.5300 -
accuracy: 0.4094 - val_loss: 0.9661 - val_accuracy: 0.3274
Epoch 2/95
114/114 [==============================] - 11s 93ms/step - loss: 1.3818 -
accuracy: 0.3953 - val_loss: 1.0438 - val_accuracy: 0.3982
Epoch 3/95
114/114 [==============================] - 10s 91ms/step - loss: 1.3943 -
accuracy: 0.3981 - val_loss: 1.1180 - val_accuracy: 0.3529
Epoch 4/95
114/114 [==============================] - 10s 91ms/step - loss: 1.3578 -
accuracy: 0.4119 - val_loss: 1.0172 - val_accuracy: 0.4336
Epoch 5/95
114/114 [==============================] - 10s 92ms/step - loss: 1.3396 -
accuracy: 0.4450 - val_loss: 1.0956 - val_accuracy: 0.4347
Epoch 6/95
114/114 [==============================] - 10s 91ms/step - loss: 1.2648 -
accuracy: 0.4707 - val_loss: 1.0400 - val_accuracy: 0.4524
Epoch 7/95
114/114 [==============================] - 10s 91ms/step - loss: 1.2527 -
```

```
accuracy: 0.4732 - val_loss: 0.9065 - val_accuracy: 0.5442
Epoch 8/95
114/114 [==============================] - 10s 91ms/step - loss: 1.2588 -
accuracy: 0.4920 - val_loss: 0.9607 - val_accuracy: 0.5122
Epoch 9/95
114/114 [==============================] - 10s 91ms/step - loss: 1.2623 -
accuracy: 0.5052 - val_loss: 0.8820 - val_accuracy: 0.5741
Epoch 10/95
114/114 [==============================] - 10s 91ms/step - loss: 1.2593 -
accuracy: 0.5243 - val_loss: 0.9279 - val_accuracy: 0.5221
Epoch 11/95
114/114 [==============================] - 10s 92ms/step - loss: 1.2495 -
accuracy: 0.5077 - val_loss: 0.9111 - val_accuracy: 0.5686
Epoch 12/95
114/114 [==============================] - 10s 92ms/step - loss: 1.1963 -
accuracy: 0.5461 - val_loss: 0.8955 - val_accuracy: 0.5874
Epoch 13/95
114/114 [==============================] - 10s 91ms/step - loss: 1.2063 -
accuracy: 0.5276 - val_loss: 0.8510 - val_accuracy: 0.5741
Epoch 14/95
114/114 [==============================] - 10s 91ms/step - loss: 1.1657 -
accuracy: 0.5486 - val_loss: 0.8932 - val_accuracy: 0.5575
Epoch 15/95
114/114 [==============================] - 10s 92ms/step - loss: 1.1700 -
accuracy: 0.5508 - val_loss: 0.8855 - val_accuracy: 0.5243
Epoch 16/95
114/114 [==============================] - 11s 93ms/step - loss: 1.1952 -
accuracy: 0.5616 - val_loss: 0.8264 - val_accuracy: 0.6250
Epoch 17/95
114/114 [==============================] - 11s 92ms/step - loss: 1.1385 -
accuracy: 0.5707 - val_loss: 0.7976 - val_accuracy: 0.6383
Epoch 18/95
114/114 [==============================] - 10s 92ms/step - loss: 1.1310 -
accuracy: 0.5859 - val_loss: 0.7578 - val_accuracy: 0.6748
Epoch 19/95
114/114 [==============================] - 10s 92ms/step - loss: 1.1181 -
accuracy: 0.5981 - val_loss: 0.8103 - val_accuracy: 0.6195
Epoch 20/95
114/114 [==============================] - 10s 92ms/step - loss: 1.1247 -
accuracy: 0.5831 - val_loss: 0.8784 - val_accuracy: 0.5863
Epoch 21/95
114/114 [==============================] - 11s 92ms/step - loss: 1.1015 -
accuracy: 0.5859 - val_loss: 0.7475 - val_accuracy: 0.6858
Epoch 22/95
114/114 [==============================] - 10s 92ms/step - loss: 1.0780 -
accuracy: 0.6196 - val_loss: 0.8072 - val_accuracy: 0.6228
Epoch 23/95
114/114 [==============================] - 10s 92ms/step - loss: 1.0662 -
```

```
accuracy: 0.6188 - val_loss: 0.7841 - val_accuracy: 0.6350
Epoch 24/95
114/114 [==============================] - 11s 93ms/step - loss: 1.0679 -
accuracy: 0.6210 - val_loss: 0.7754 - val_accuracy: 0.6737
Epoch 25/95
114/114 [==============================] - 11s 93ms/step - loss: 1.0598 -
accuracy: 0.6276 - val_loss: 0.7212 - val_accuracy: 0.7069
Epoch 26/95
114/114 [==============================] - 11s 93ms/step - loss: 1.0574 -
accuracy: 0.6249 - val_loss: 0.6672 - val_accuracy: 0.7190
Epoch 27/95
114/114 [==============================] - 10s 92ms/step - loss: 1.0245 -
accuracy: 0.6398 - val_loss: 0.7419 - val_accuracy: 0.6726
Epoch 28/95
114/114 [==============================] - 11s 93ms/step - loss: 1.0285 -
accuracy: 0.6436 - val_loss: 0.7564 - val_accuracy: 0.6681
Epoch 29/95
114/114 [==============================] - 10s 91ms/step - loss: 0.9822 -
accuracy: 0.6616 - val_loss: 0.8180 - val_accuracy: 0.6327
Epoch 30/95
114/114 [==============================] - 10s 92ms/step - loss: 0.9537 -
accuracy: 0.6685 - val_loss: 0.8334 - val_accuracy: 0.6460
Epoch 31/95
114/114 [==============================] - 10s 91ms/step - loss: 0.9654 -
accuracy: 0.6630 - val_loss: 0.8158 - val_accuracy: 0.6239
Epoch 32/95
114/114 [==============================] - 10s 92ms/step - loss: 1.0119 -
accuracy: 0.6497 - val_loss: 0.7696 - val_accuracy: 0.6471
Epoch 33/95
114/114 [==============================] - 10s 91ms/step - loss: 0.9581 -
accuracy: 0.6608 - val_loss: 0.6939 - val_accuracy: 0.6980
Epoch 34/95
114/114 [==============================] - 11s 92ms/step - loss: 0.9563 -
accuracy: 0.6616 - val_loss: 0.6568 - val_accuracy: 0.7389
Epoch 35/95
114/114 [==============================] - 10s 92ms/step - loss: 0.9179 -
accuracy: 0.6854 - val_loss: 0.6822 - val_accuracy: 0.7223
Epoch 36/95
114/114 [==============================] - 10s 91ms/step - loss: 0.9580 -
accuracy: 0.6685 - val_loss: 0.7867 - val_accuracy: 0.6836
Epoch 37/95
114/114 [==============================] - 10s 92ms/step - loss: 0.9387 -
accuracy: 0.6862 - val_loss: 0.6653 - val_accuracy: 0.7456
Epoch 38/95
114/114 [==============================] - 10s 92ms/step - loss: 0.9328 -
accuracy: 0.6732 - val_loss: 0.6634 - val_accuracy: 0.7201
Epoch 39/95
114/114 [==============================] - 11s 92ms/step - loss: 0.9061 -
```

```
accuracy: 0.6892 - val_loss: 0.7847 - val_accuracy: 0.6737
Epoch 40/95
114/114 [==============================] - 10s 92ms/step - loss: 0.8890 -
accuracy: 0.6967 - val_loss: 0.7650 - val_accuracy: 0.6792
Epoch 41/95
114/114 [==============================] - 10s 92ms/step - loss: 0.9079 -
accuracy: 0.6997 - val_loss: 0.7849 - val_accuracy: 0.6604
Epoch 42/95
114/114 [==============================] - 10s 92ms/step - loss: 0.8664 -
accuracy: 0.7122 - val_loss: 0.7449 - val_accuracy: 0.6858
Epoch 43/95
114/114 [==============================] - 10s 92ms/step - loss: 0.8772 -
accuracy: 0.6992 - val_loss: 0.7199 - val_accuracy: 0.6847
Epoch 44/95
114/114 [==============================] - 10s 92ms/step - loss: 0.8579 -
accuracy: 0.7019 - val_loss: 0.6962 - val_accuracy: 0.6958
Epoch 45/95
114/114 [==============================] - 10s 92ms/step - loss: 0.8225 -
accuracy: 0.7243 - val_loss: 0.7054 - val_accuracy: 0.6881
Epoch 46/95
114/114 [==============================] - 10s 92ms/step - loss: 0.8184 -
accuracy: 0.7130 - val_loss: 0.6949 - val_accuracy: 0.7058
Epoch 47/95
114/114 [==============================] - 10s 92ms/step - loss: 0.8491 -
accuracy: 0.6961 - val_loss: 0.7196 - val_accuracy: 0.6847
Epoch 48/95
114/114 [==============================] - 11s 92ms/step - loss: 0.8045 -
accuracy: 0.7356 - val_loss: 0.5959 - val_accuracy: 0.7533
Epoch 49/95
114/114 [==============================] - 11s 93ms/step - loss: 0.7812 -
accuracy: 0.7456 - val_loss: 0.6976 - val_accuracy: 0.7069
Epoch 50/95
114/114 [==============================] - 10s 92ms/step - loss: 0.8091 -
accuracy: 0.7323 - val_loss: 0.7802 - val_accuracy: 0.6715
Epoch 51/95
114/114 [==============================] - 11s 94ms/step - loss: 0.7662 -
accuracy: 0.7367 - val_loss: 0.6694 - val_accuracy: 0.7412
Epoch 52/95
114/114 [==============================] - 11s 92ms/step - loss: 0.7581 -
accuracy: 0.7373 - val_loss: 0.7144 - val_accuracy: 0.7046
Epoch 53/95
114/114 [==============================] - 11s 92ms/step - loss: 0.7972 -
accuracy: 0.7406 - val_loss: 0.6338 - val_accuracy: 0.7356
Epoch 54/95
114/114 [==============================] - 10s 92ms/step - loss: 0.7526 -
accuracy: 0.7439 - val_loss: 0.5927 - val_accuracy: 0.7533
Epoch 55/95
114/114 [==============================] - 10s 92ms/step - loss: 0.7582 -
```

```
accuracy: 0.7481 - val_loss: 0.6431 - val_accuracy: 0.7511
Epoch 56/95
114/114 [==============================] - 10s 92ms/step - loss: 0.7408 -
accuracy: 0.7398 - val_loss: 0.5873 - val_accuracy: 0.7699
Epoch 57/95
114/114 [==============================] - 10s 92ms/step - loss: 0.7310 -
accuracy: 0.7525 - val_loss: 0.5776 - val_accuracy: 0.7821
Epoch 58/95
114/114 [==============================] - 10s 92ms/step - loss: 0.7452 -
accuracy: 0.7561 - val_loss: 0.5840 - val_accuracy: 0.7754
Epoch 59/95
114/114 [==============================] - 10s 92ms/step - loss: 0.7670 -
accuracy: 0.7436 - val_loss: 0.6058 - val_accuracy: 0.7511
Epoch 60/95
114/114 [==============================] - 10s 92ms/step - loss: 0.6962 -
accuracy: 0.7622 - val_loss: 0.6417 - val_accuracy: 0.7279
Epoch 61/95
114/114 [==============================] - 10s 92ms/step - loss: 0.7565 -
accuracy: 0.7486 - val_loss: 0.6257 - val_accuracy: 0.7732
Epoch 62/95
114/114 [==============================] - 10s 92ms/step - loss: 0.7188 -
accuracy: 0.7500 - val_loss: 0.5954 - val_accuracy: 0.7644
Epoch 63/95
114/114 [==============================] - 10s 92ms/step - loss: 0.6983 -
accuracy: 0.7674 - val_loss: 0.5481 - val_accuracy: 0.7799
Epoch 64/95
114/114 [==============================] - 10s 92ms/step - loss: 0.6752 -
accuracy: 0.7710 - val_loss: 0.5857 - val_accuracy: 0.7666
Epoch 65/95
114/114 [==============================] - 10s 92ms/step - loss: 0.7194 -
accuracy: 0.7715 - val_loss: 0.5384 - val_accuracy: 0.7765
Epoch 66/95
114/114 [==============================] - 10s 92ms/step - loss: 0.6811 -
accuracy: 0.7713 - val_loss: 0.5800 - val_accuracy: 0.7743
Epoch 67/95
114/114 [==============================] - 10s 92ms/step - loss: 0.6733 -
accuracy: 0.7782 - val_loss: 0.5678 - val_accuracy: 0.7699
Epoch 68/95
114/114 [==============================] - 10s 92ms/step - loss: 0.6651 -
accuracy: 0.7715 - val_loss: 0.5973 - val_accuracy: 0.7721
Epoch 69/95
114/114 [==============================] - 10s 91ms/step - loss: 0.6991 -
accuracy: 0.7539 - val_loss: 0.7082 - val_accuracy: 0.7102
Epoch 70/95
114/114 [==============================] - 10s 92ms/step - loss: 0.6653 -
accuracy: 0.7765 - val_loss: 0.6260 - val_accuracy: 0.7622
Epoch 71/95
114/114 [==============================] - 10s 92ms/step - loss: 0.6542 -
```

```
accuracy: 0.7641 - val_loss: 0.5859 - val_accuracy: 0.7489
Epoch 72/95
114/114 [==============================] - 10s 91ms/step - loss: 0.6564 -
accuracy: 0.7843 - val_loss: 0.5956 - val_accuracy: 0.7533
Epoch 73/95
114/114 [==============================] - 10s 92ms/step - loss: 0.6639 -
accuracy: 0.7895 - val_loss: 0.6072 - val_accuracy: 0.7677
Epoch 74/95
114/114 [==============================] - 10s 92ms/step - loss: 0.6347 -
accuracy: 0.8036 - val_loss: 0.6252 - val_accuracy: 0.7655
Epoch 75/95
114/114 [==============================] - 10s 92ms/step - loss: 0.6434 -
accuracy: 0.7804 - val_loss: 0.6278 - val_accuracy: 0.7699
Epoch 76/95
114/114 [==============================] - 11s 93ms/step - loss: 0.6456 -
accuracy: 0.7917 - val_loss: 0.5484 - val_accuracy: 0.8053
Epoch 77/95
114/114 [==============================] - 10s 92ms/step - loss: 0.6248 -
accuracy: 0.7936 - val_loss: 0.6062 - val_accuracy: 0.7666
Epoch 78/95
114/114 [==============================] - 10s 92ms/step - loss: 0.6282 -
accuracy: 0.7945 - val_loss: 0.6880 - val_accuracy: 0.7445
Epoch 79/95
114/114 [==============================] - 10s 92ms/step - loss: 0.5763 -
accuracy: 0.8116 - val_loss: 0.5515 - val_accuracy: 0.8042
Epoch 80/95
114/114 [==============================] - 10s 92ms/step - loss: 0.5931 -
accuracy: 0.8077 - val_loss: 0.6361 - val_accuracy: 0.7677
Epoch 81/95
114/114 [==============================] - 10s 92ms/step - loss: 0.5928 -
accuracy: 0.8030 - val_loss: 0.7326 - val_accuracy: 0.7323
Epoch 82/95
114/114 [==============================] - 10s 91ms/step - loss: 0.6142 -
accuracy: 0.8102 - val_loss: 0.6043 - val_accuracy: 0.7600
Epoch 83/95
114/114 [==============================] - 11s 93ms/step - loss: 0.5727 -
accuracy: 0.8149 - val_loss: 0.6140 - val_accuracy: 0.7633
Epoch 84/95
114/114 [==============================] - 11s 93ms/step - loss: 0.5837 -
accuracy: 0.8058 - val_loss: 0.5637 - val_accuracy: 0.7987
Epoch 85/95
114/114 [==============================] - 11s 93ms/step - loss: 0.5601 -
accuracy: 0.8191 - val_loss: 0.5712 - val_accuracy: 0.8031
Epoch 86/95
114/114 [==============================] - 11s 93ms/step - loss: 0.5893 -
accuracy: 0.8011 - val_loss: 0.6224 - val_accuracy: 0.7633
Epoch 87/95
114/114 [==============================] - 10s 91ms/step - loss: 0.5346 -
```

```
accuracy: 0.8232 - val_loss: 0.6272 - val_accuracy: 0.7743
Epoch 88/95
114/114 [==============================] - 11s 95ms/step - loss: 0.5297 -
accuracy: 0.8215 - val_loss: 0.6244 - val_accuracy: 0.7832
Epoch 89/95
114/114 [==============================] - 11s 93ms/step - loss: 0.5742 -
accuracy: 0.8152 - val_loss: 0.6179 - val_accuracy: 0.7666
Epoch 90/95
114/114 [==============================] - 10s 90ms/step - loss: 0.5537 -
accuracy: 0.8102 - val_loss: 0.6972 - val_accuracy: 0.7367
Epoch 91/95
114/114 [==============================] - 10s 90ms/step - loss: 0.5339 -
accuracy: 0.8260 - val_loss: 0.6017 - val_accuracy: 0.7644
Epoch 92/95
114/114 [==============================] - 10s 90ms/step - loss: 0.5732 -
accuracy: 0.8207 - val_loss: 0.6120 - val_accuracy: 0.7677
Epoch 93/95
114/114 [==============================] - 10s 90ms/step - loss: 0.5578 -
accuracy: 0.8171 - val_loss: 0.5776 - val_accuracy: 0.7876
Epoch 94/95
114/114 [==============================] - 10s 90ms/step - loss: 0.4847 -
accuracy: 0.8470 - val_loss: 0.6964 - val_accuracy: 0.7456
Epoch 95/95
114/114 [==============================] - 10s 90ms/step - loss: 0.5432 -
accuracy: 0.8320 - val_loss: 0.6934 - val_accuracy: 0.7367
```

1.9 Jedi, Sith, or Neither?

Now let's give it our own images and see if it thinks the image shows a Jedi, a Sith, or neither one.

[35]:
```
import ipywidgets as widgets
import io

uploader = widgets.FileUpload()
uploader
```

```
FileUpload(value={}, description='Upload')
```

[98]:
```
for name, file_info in uploader.value.items():
    our_image = PIL.Image.open(io.BytesIO(file_info['content']))

our_image
```

[98]:

[99]:
```
our_image = PIL.Image.open(io.BytesIO(file_info['content']))
our_image = tf.image.resize(our_image, [img_height,img_width])
our_image_array = our_image[:,:,:3] # discard alpha channel
tf.keras.utils.array_to_img(our_image_array)
```

[99]:

[100]:
```
our_image_array = tf.expand_dims(our_image_array, 0) # Create a batch of a␣
↪single image
```

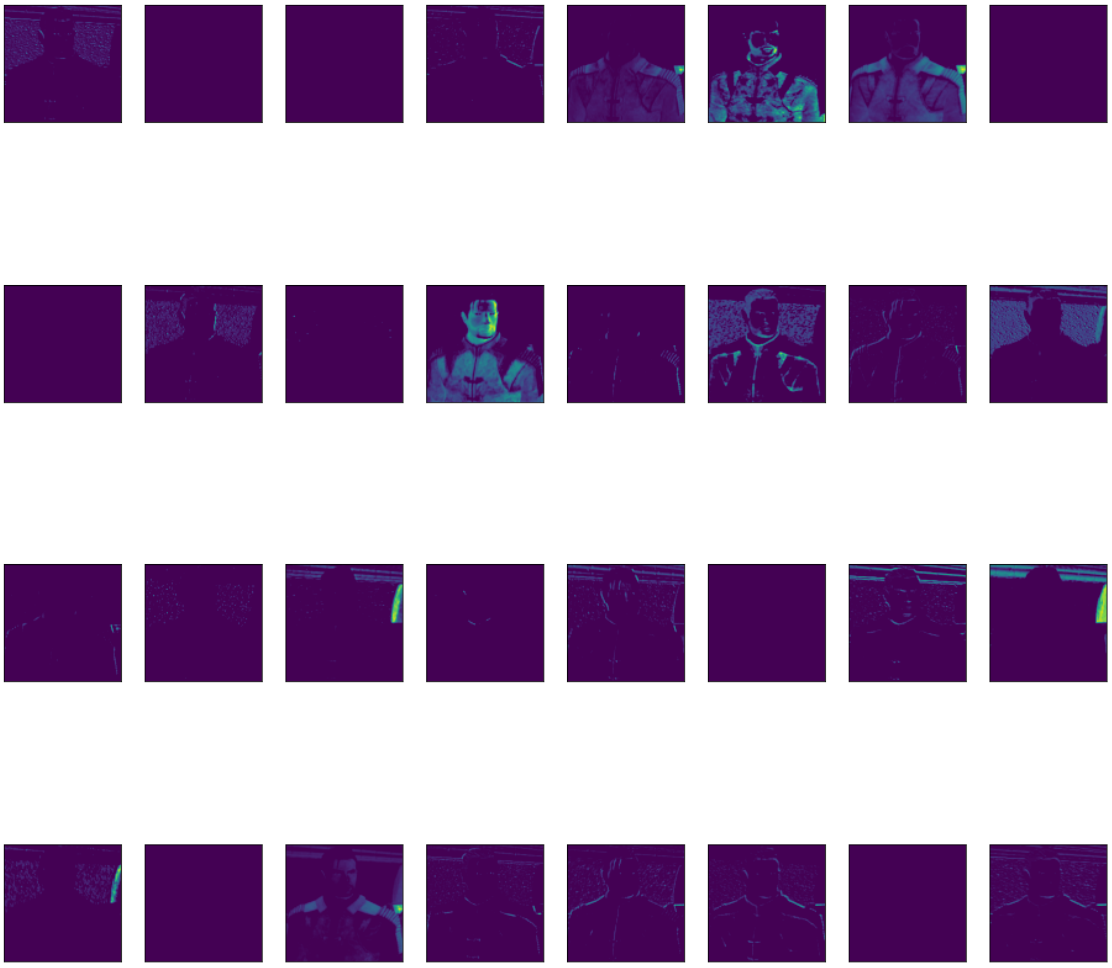
[101]:
```
our_predictions = model.predict(our_image_array)

print(f"{our_predictions[0][0]:.0%} likelihood that this is a picture of a␣
↪{class_names[0]}, {our_predictions[0][2]:.0%} likelihood that this is a␣
↪picture of a {class_names[2]}, and {our_predictions[0][1]:.0%} likelihood␣
↪that this picture shows {class_names[1]} a {class_names[0]} nor a␣
↪{class_names[2]}.")
```

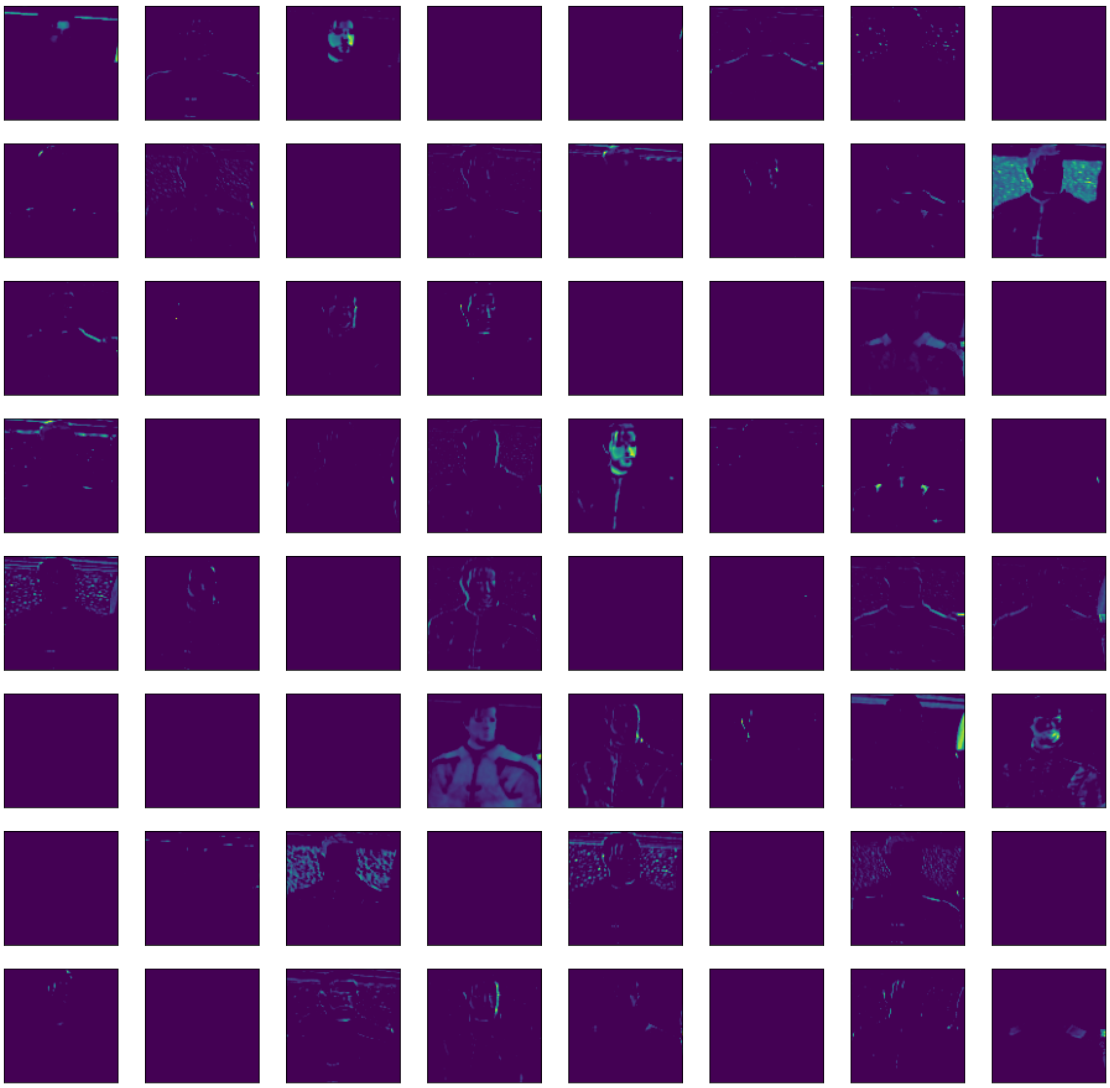
```
9% likelihood that this is a picture of a jedi, 5% likelihood that this is a
picture of a sith, and 86% likelihood that this picture shows neither a jedi nor
a sith.
```

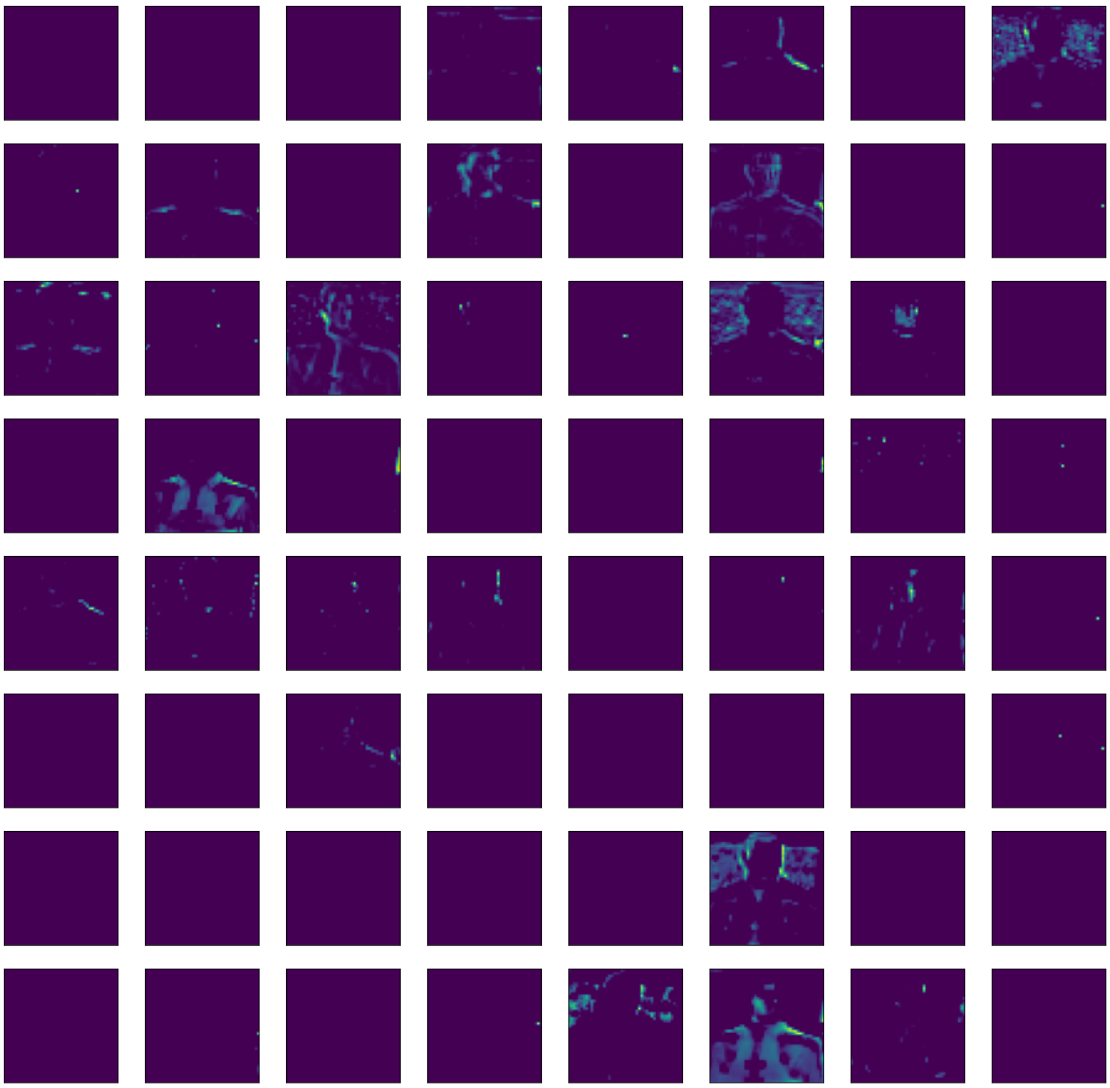
1.9.1 How did our model see this image?

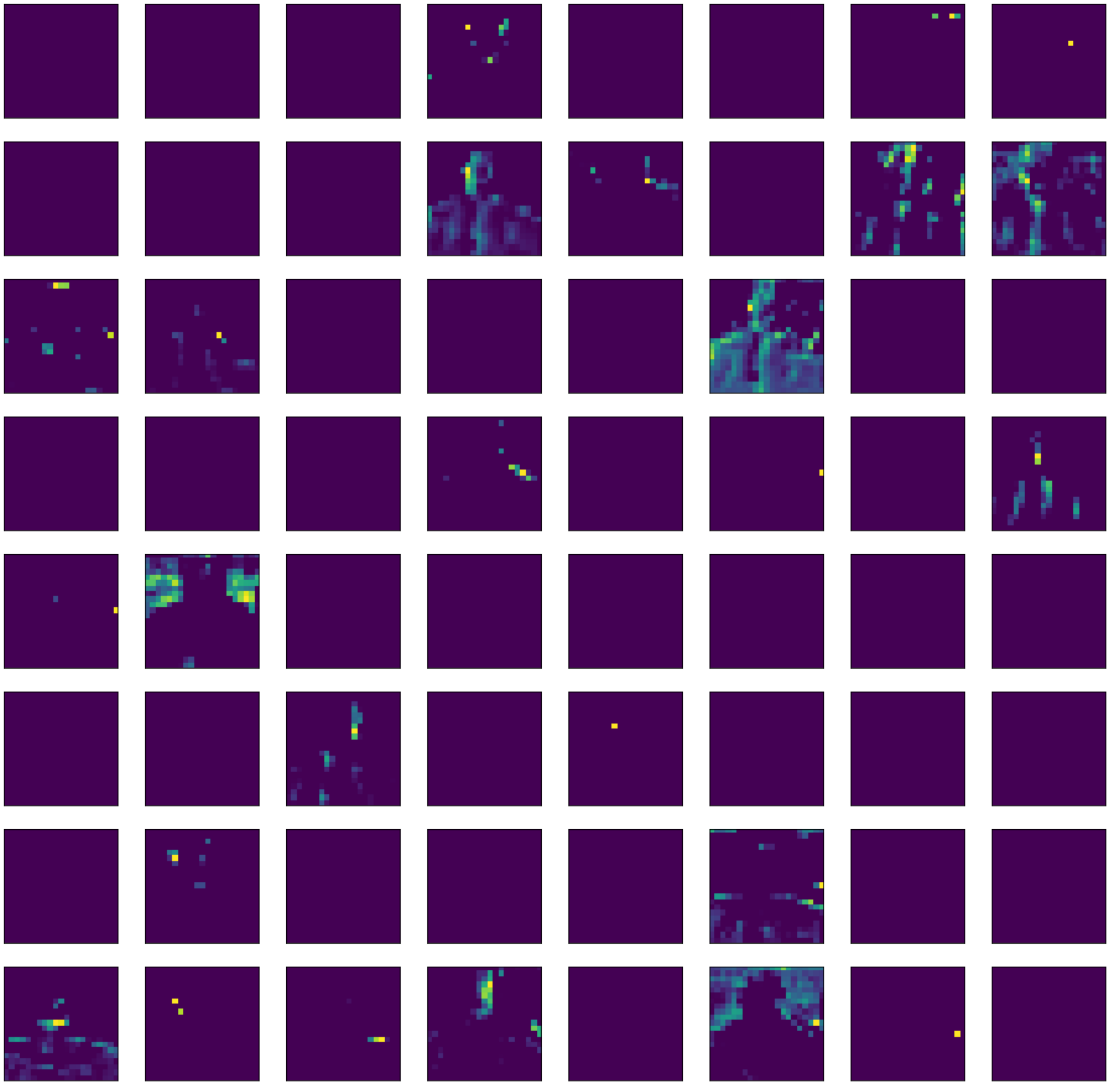
We'll ask the second model (`feature_model`) to make a prediction here, which you recall, causes it to output the feature maps.

[102]:
```
from matplotlib import pyplot

feature_maps = feature_model.predict(our_image_array)
sizes = [16, 32, 64, 64, 64]
row_size = 8

for map_index, feature_map in enumerate(feature_maps):
    current_index = 1
    num_rows = int(sizes[map_index]/row_size)
    figure = pyplot.figure(figsize=(16,16))

    for row in range(num_rows):
        for column in range(row_size):
            ax = figure.add_subplot(num_rows, row_size, current_index)
            ax.set_xticks([])
            ax.set_yticks([])

            # use grayscale filter
            pyplot.imshow(feature_map[0, :, :, current_index-1])
            current_index += 1

    # show the plot
    pyplot.show()
```









[ ]:

[ ]: