Jedi vs. Sith

July 7, 2022

1 Jedi vs. Sith

A binary image classifier example

1.1 Goal

The goal of this notebook is to demonstrate the functionality of a binary 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'.

[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 (200x200).

[3]:
```
batch_size = 32
img_height = 200
img_width = 200
```

1.4 Training Dataset

We have a training dataset of hand-curated images, all sized to 200x200px in two subdirectories called "jedi" and "sith". We'll load them here.

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

jedi = list(data_dir.glob('jedi/*'))
sith = list(data_dir.glob('sith/*'))
num_jedi = len(jedi)
num_sith = len(sith)

print(num_jedi, 'Jedi images and', num_sith, 'Sith images')
```

```
745 Jedi images and 637 Sith 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]:
```
# 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=123,
    image_size=(img_height,img_width),
    batch_size=batch_size)

class_names = train_ds.class_names
```

```
Found 1382 files belonging to 2 classes.
Using 1106 files for training.
```

```
2022-07-07 09:57:14.650624: 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 09:57:15.359998: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1525] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 1472 MB memory:  -> device: 0,
name: NVIDIA GeForce RTX 2060 SUPER, pci bus id: 0000:01:00.0, compute
capability: 7.5
```

[8]:
```
# Validation dataset
val_ds = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    validation_split=0.2,
    subset='validation',
    seed=123,
    image_size=(img_height,img_width),
    batch_size=batch_size)
```

```
Found 1382 files belonging to 2 classes.
Using 276 files for validation.
```

1.5 I/O Tuning

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

[9]:
```
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.

[10]:
```
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 200x200px color images, we need an input tensor shape of 200x200x3. 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%.

[11]:
```
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.

[12]:
```
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).

[13]:
```
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.

[14]:
```
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.

[15]:
```
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.

[16]:
```
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.

[17]:
```
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.

[18]:
```
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

[19]:
```
model.add(tf.keras.layers.MaxPooling2D(2, 2))
```

1.6.10 Layer 8: Fourth Convolution

Same as the third.

[20]:
```
model.add(tf.keras.layers.Conv2D(64, (3,3), activation='relu'))
```

1.6.11 Layer 9: Max Pooling

Lather, rinse, repeat.

[21]:
```
model.add(tf.keras.layers.MaxPooling2D(2, 2))
```

1.6.12 Layer 10: Fifth Convolution

Same as the third and fourth.

[22]:
```
model.add(tf.keras.layers.Conv2D(64, (3,3), activation='relu'))
```

1.6.13 Layer 11: Max Pooling

Last one.

[23]:
```
model.add(tf.keras.layers.MaxPooling2D(2, 2))
```

1.6.14 Layer 12: Flattening

The output shape of the last max pooling layer is 4x4x64. 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. 4 times 4 is 16 and 16 times 64 is 1024. Therefore, the input to our DNN will be a vector containing 1024 terms.

[24]:
```
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.

[25]:
```
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 want to have one connection (because we have only two possible outcomes - a binary classifier) per pair of adjacent data points. ~~We will use 257 neurons, the mean of the input layer (512) and output layer (1).~~ We'll use 512 neurons. 257 gives decent performance overall but for some reason gets confused about people with dark skin. 512 neurons get less confused. We're again using ReLU as the activation function to prevent backpropagation of negative bias, which adds complexity to the computation.

[26]:
```
model.add(tf.keras.layers.Dense(512, activation='relu'))
```

1.7 Layer 15: Dropout

A final dropout layer that randomly knocks out 1/5 of the neurons in the network.

[27]:
```
model.add(tf.keras.layers.Dropout(rate=0.2))
```

1.7.1 Layer 16: Ouput DNN

Our final layer is a dense neural network consisting of a single neuron - we only want one answer. This layer takes the 512 answers from the previous layer and makes the final call, applying a sigmoid function to maximize the magnitude of our single yes or no option: Jedi or Sith.

(Fun fact: we really should be using a step 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 sigmoid function instead, which accomplishes the same thing after rounding.)

[28]:
```
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
```

1.7.2 Model Summary

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

[29]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 random_flip (RandomFlip)    (None, 200, 200, 3)       0

 random_rotation (RandomRota  (None, 200, 200, 3)      0
 tion)

 random_contrast (RandomCont  (None, 200, 200, 3)      0
 rast)

 random_zoom (RandomZoom)    (None, 200, 200, 3)       0

 rescaling (Rescaling)       (None, 200, 200, 3)       0

 conv2d (Conv2D)             (None, 198, 198, 16)      448

 dropout (Dropout)           (None, 198, 198, 16)      0

 max_pooling2d (MaxPooling2D  (None, 99, 99, 16)       0
 )

 conv2d_1 (Conv2D)           (None, 97, 97, 32)        4640

 max_pooling2d_1 (MaxPooling  (None, 48, 48, 32)       0
 2D)

 conv2d_2 (Conv2D)           (None, 46, 46, 64)        18496

 max_pooling2d_2 (MaxPooling  (None, 23, 23, 64)       0
 2D)

 conv2d_3 (Conv2D)           (None, 21, 21, 64)        36928

 max_pooling2d_3 (MaxPooling  (None, 10, 10, 64)       0
 2D)

 conv2d_4 (Conv2D)           (None, 8, 8, 64)          36928

 max_pooling2d_4 (MaxPooling  (None, 4, 4, 64)         0
 2D)

 flatten (Flatten)           (None, 1024)              0
```

```
 dropout_1 (Dropout)         (None, 1024)              0

 dense (Dense)               (None, 512)               524800

 dropout_2 (Dropout)         (None, 512)               0

 dense_1 (Dense)             (None, 1)                 513

=================================================================
Total params: 622,753
Trainable params: 622,753
Non-trainable params: 0
_________________________________________________________________
```

1.8 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.

[30]:
```
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)
```

1.9 Training

Our model will be trained by applying a binary cross-entropy loss function. See Understanding Binary Cross-Entropy Loss to better understand why.

Binary cross-entropy loss function:

$$H(p,q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log(p(y_i)) + (1-y_i) \cdot \log(1-p(y_i))$$

where y is the dependent variable ("Sith-ness"), $p(y)$ is the probability that the image shows a Sith, and q is the *estimated* probability that the image shows a Sith. The formula relates the magnitude (H) of how far off the mark the guess (q) was from the real value (p).

We will be applying the Root Mean Square Propagation (RMSProp) optimizer function to adjust learning rate dynamically. Read more about it here. Our baseline learning rate will be 0.0001 to prevent model divergence.

Root mean squre error:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}{N}}$$

[31]:
```
model.compile(
    loss='binary_crossentropy',
    optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.0001),
    metrics='accuracy')
```

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

~~We'll train for 6 epochs. 6-8 is a good range for this network. I started with 10, which overfit the model readily, even with dropouts.~~

Now that we have the values in the correct domain ([0..1]), I can train for longer without it overfitting, leading to better results.

[32]:
```
history = model.fit(
    train_ds,
    epochs=29,
    verbose=1,
    validation_data=val_ds,)
```

```
Epoch 1/29

2022-07-07 09:57:18.495223: I tensorflow/stream_executor/cuda/cuda_dnn.cc:368]
Loaded cuDNN version 8101

35/35 [==============================] - 10s 144ms/step - loss: 0.6778 -
accuracy: 0.5570 - val_loss: 0.6353 - val_accuracy: 0.7536
Epoch 2/29
35/35 [==============================] - 2s 45ms/step - loss: 0.5887 - accuracy:
0.7197 - val_loss: 0.5560 - val_accuracy: 0.6993
Epoch 3/29
35/35 [==============================] - 2s 54ms/step - loss: 0.4970 - accuracy:
0.7731 - val_loss: 0.5407 - val_accuracy: 0.7210
Epoch 4/29
35/35 [==============================] - 2s 55ms/step - loss: 0.4791 - accuracy:
0.7794 - val_loss: 0.6380 - val_accuracy: 0.6775
Epoch 5/29
35/35 [==============================] - 2s 54ms/step - loss: 0.4611 - accuracy:
0.7821 - val_loss: 0.5018 - val_accuracy: 0.7319
Epoch 6/29
35/35 [==============================] - 2s 53ms/step - loss: 0.4617 - accuracy:
0.7875 - val_loss: 0.6678 - val_accuracy: 0.6739
Epoch 7/29
35/35 [==============================] - 2s 54ms/step - loss: 0.4575 - accuracy:
0.8011 - val_loss: 0.6357 - val_accuracy: 0.6848
Epoch 8/29
35/35 [==============================] - 2s 54ms/step - loss: 0.4532 - accuracy:
0.7975 - val_loss: 0.5528 - val_accuracy: 0.7210
Epoch 9/29
35/35 [==============================] - 2s 53ms/step - loss: 0.4458 - accuracy:
0.8065 - val_loss: 0.5531 - val_accuracy: 0.7174
Epoch 10/29
35/35 [==============================] - 2s 49ms/step - loss: 0.4423 - accuracy:
0.7957 - val_loss: 0.5559 - val_accuracy: 0.7138
Epoch 11/29
35/35 [==============================] - 2s 50ms/step - loss: 0.4433 - accuracy:
```

```
0.8065 - val_loss: 0.5865 - val_accuracy: 0.7174
Epoch 12/29
35/35 [==============================] - 2s 55ms/step - loss: 0.4299 - accuracy:
0.8156 - val_loss: 0.7279 - val_accuracy: 0.6703
Epoch 13/29
35/35 [==============================] - 2s 54ms/step - loss: 0.4257 - accuracy:
0.8119 - val_loss: 0.5076 - val_accuracy: 0.7536
Epoch 14/29
35/35 [==============================] - 2s 53ms/step - loss: 0.4261 - accuracy:
0.8192 - val_loss: 0.6994 - val_accuracy: 0.6739
Epoch 15/29
35/35 [==============================] - 2s 55ms/step - loss: 0.4008 - accuracy:
0.8336 - val_loss: 0.7878 - val_accuracy: 0.6703
Epoch 16/29
35/35 [==============================] - 2s 55ms/step - loss: 0.4216 - accuracy:
0.8146 - val_loss: 0.7458 - val_accuracy: 0.6667
Epoch 17/29
35/35 [==============================] - 2s 55ms/step - loss: 0.4008 - accuracy:
0.8282 - val_loss: 0.7834 - val_accuracy: 0.6558
Epoch 18/29
35/35 [==============================] - 2s 52ms/step - loss: 0.3941 - accuracy:
0.8318 - val_loss: 0.4646 - val_accuracy: 0.7790
Epoch 19/29
35/35 [==============================] - 2s 48ms/step - loss: 0.4000 - accuracy:
0.8219 - val_loss: 0.4858 - val_accuracy: 0.7717
Epoch 20/29
35/35 [==============================] - 2s 56ms/step - loss: 0.3844 - accuracy:
0.8373 - val_loss: 0.5244 - val_accuracy: 0.7536
Epoch 21/29
35/35 [==============================] - 2s 55ms/step - loss: 0.3842 - accuracy:
0.8373 - val_loss: 0.5533 - val_accuracy: 0.7391
Epoch 22/29
35/35 [==============================] - 2s 55ms/step - loss: 0.3911 - accuracy:
0.8318 - val_loss: 0.4378 - val_accuracy: 0.8080
Epoch 23/29
35/35 [==============================] - 2s 53ms/step - loss: 0.3850 - accuracy:
0.8345 - val_loss: 0.5988 - val_accuracy: 0.7246
Epoch 24/29
35/35 [==============================] - 2s 55ms/step - loss: 0.3747 - accuracy:
0.8382 - val_loss: 0.5264 - val_accuracy: 0.7572
Epoch 25/29
35/35 [==============================] - 2s 57ms/step - loss: 0.3887 - accuracy:
0.8363 - val_loss: 0.7342 - val_accuracy: 0.6920
Epoch 26/29
35/35 [==============================] - 2s 54ms/step - loss: 0.3733 - accuracy:
0.8535 - val_loss: 0.5228 - val_accuracy: 0.7609
Epoch 27/29
35/35 [==============================] - 2s 46ms/step - loss: 0.3833 - accuracy:
```

```
0.8282 - val_loss: 0.9026 - val_accuracy: 0.6413
Epoch 28/29
35/35 [==============================] - 2s 55ms/step - loss: 0.3554 - accuracy:
0.8463 - val_loss: 0.9145 - val_accuracy: 0.6486
Epoch 29/29
35/35 [==============================] - 2s 55ms/step - loss: 0.3686 - accuracy:
0.8454 - val_loss: 0.7017 - val_accuracy: 0.6957
```

1.10 Jedi or Sith?

Now let's give it our own images and see if it thinks the image shows a Jedi or a Sith.

[33]:
```
import ipywidgets as widgets
import io

uploader = widgets.FileUpload()
uploader
```

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

[87]:
```
for name, file_info in uploader.value.items():
    our_image = PIL.Image.open(io.BytesIO(file_info['content']))

our_image
```

[87]:

[88]:
```
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)
```

[88]:

[89]:
```
our_image_array = tf.expand_dims(our_image_array, 0) # Create a batch of a␣
↪single image
```

[90]:
```
our_predictions = model.predict(our_image_array)
our_score = our_predictions[0][0]
our_class = class_names[int(np.around(our_score))]

print(our_score)

print("This image most likely shows a {}".format(our_class))
```

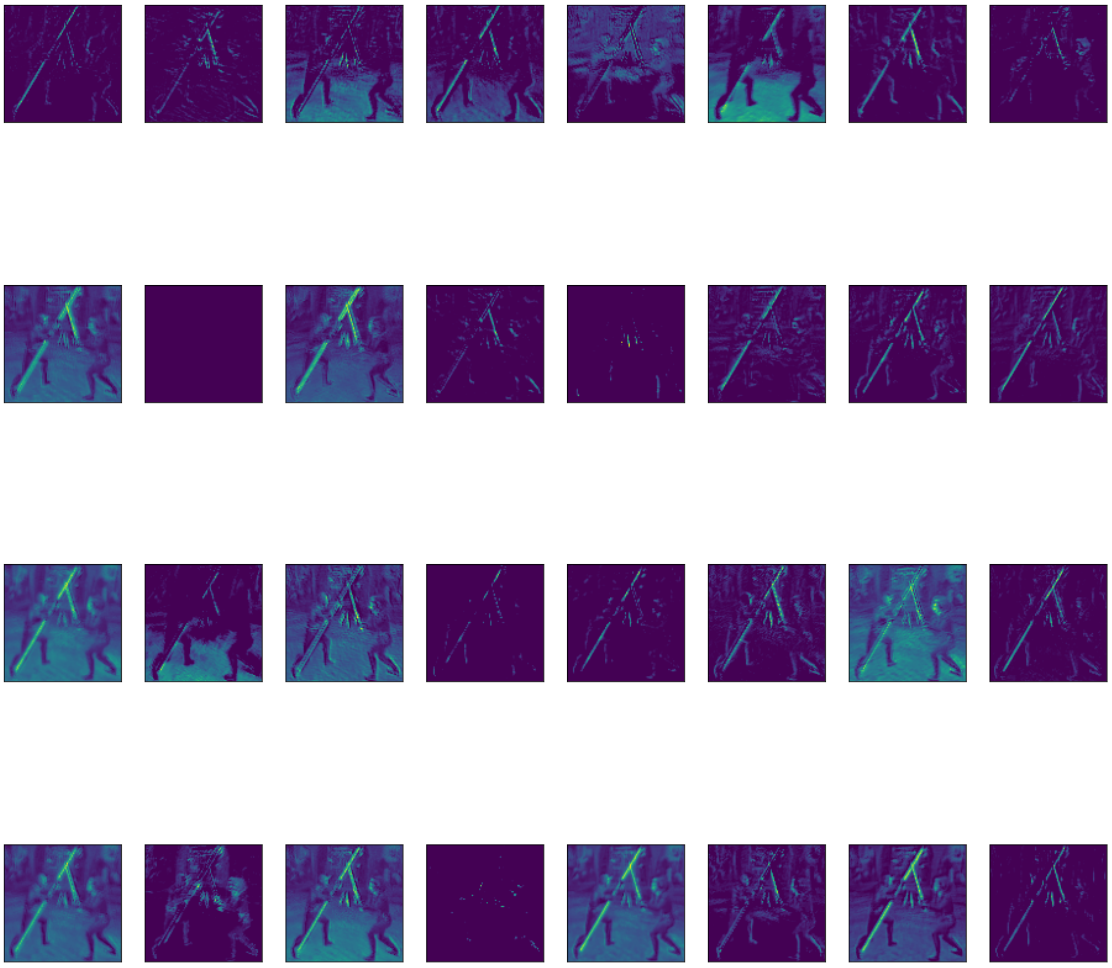
```
0.9903965
This image most likely shows a sith
```

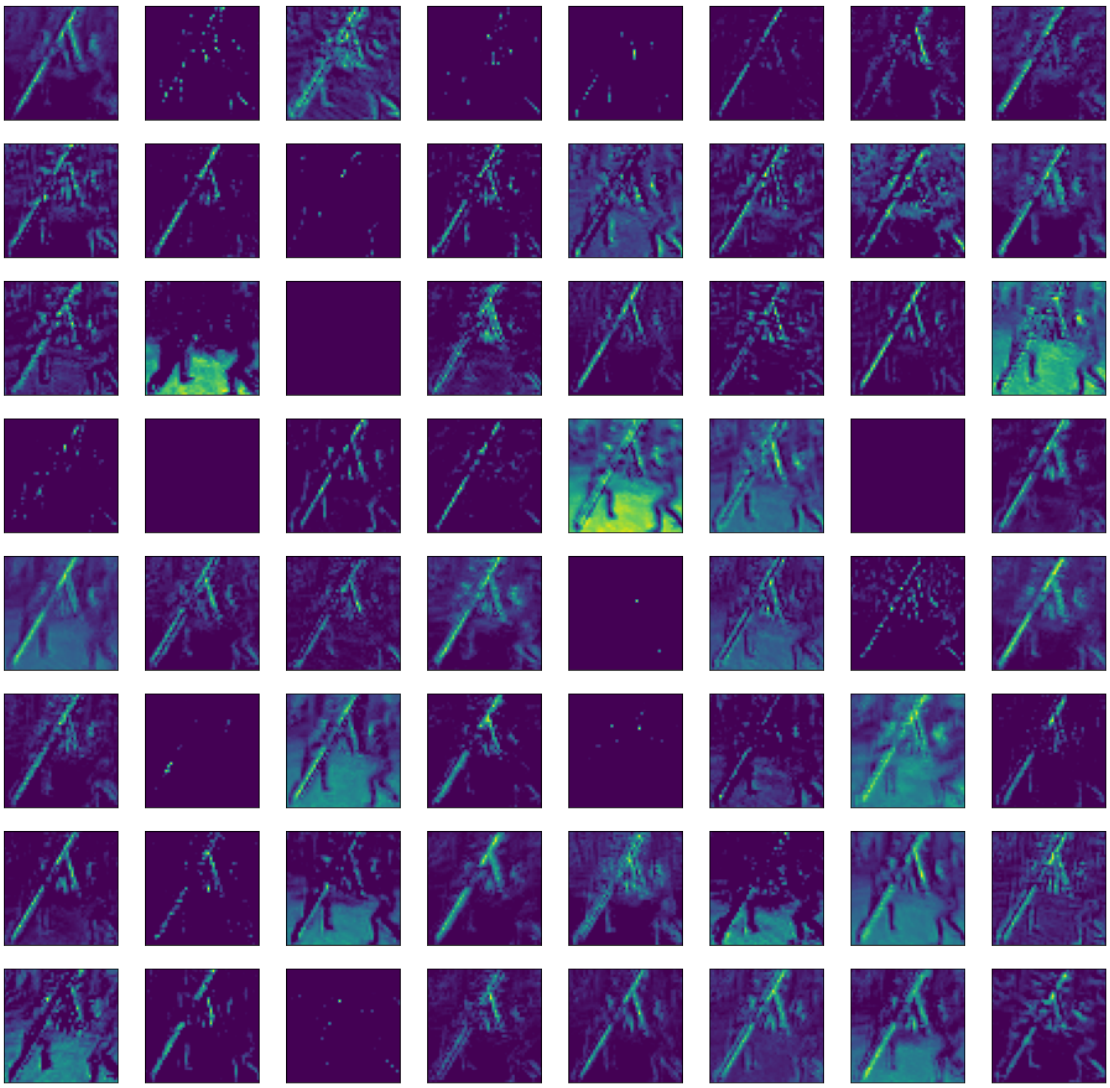
1.10.1 How did our model see this image?

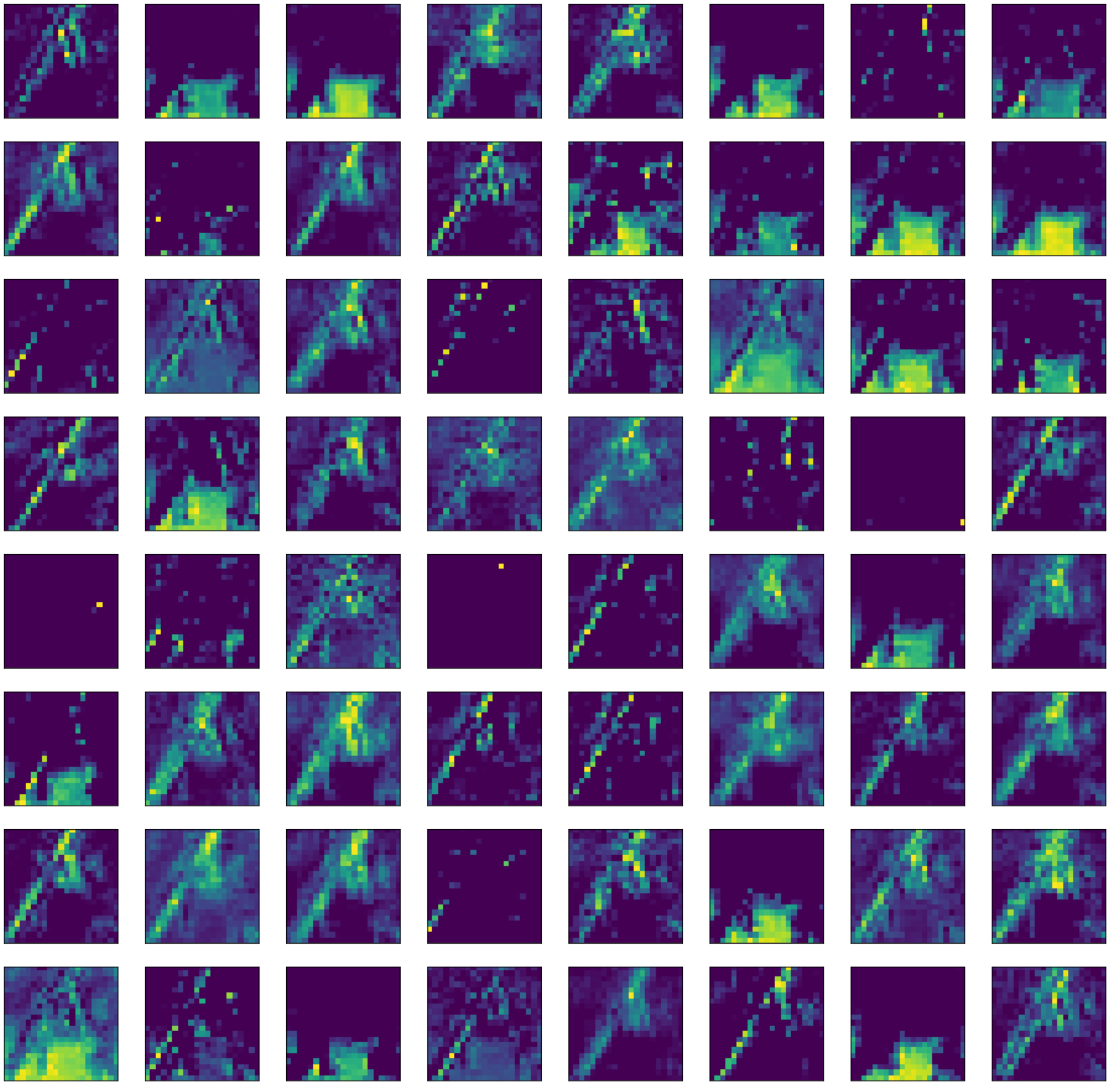
We'll ask the second model (`feature_model`) to make a prediction here, which you recall, causes it to output the feature maps.

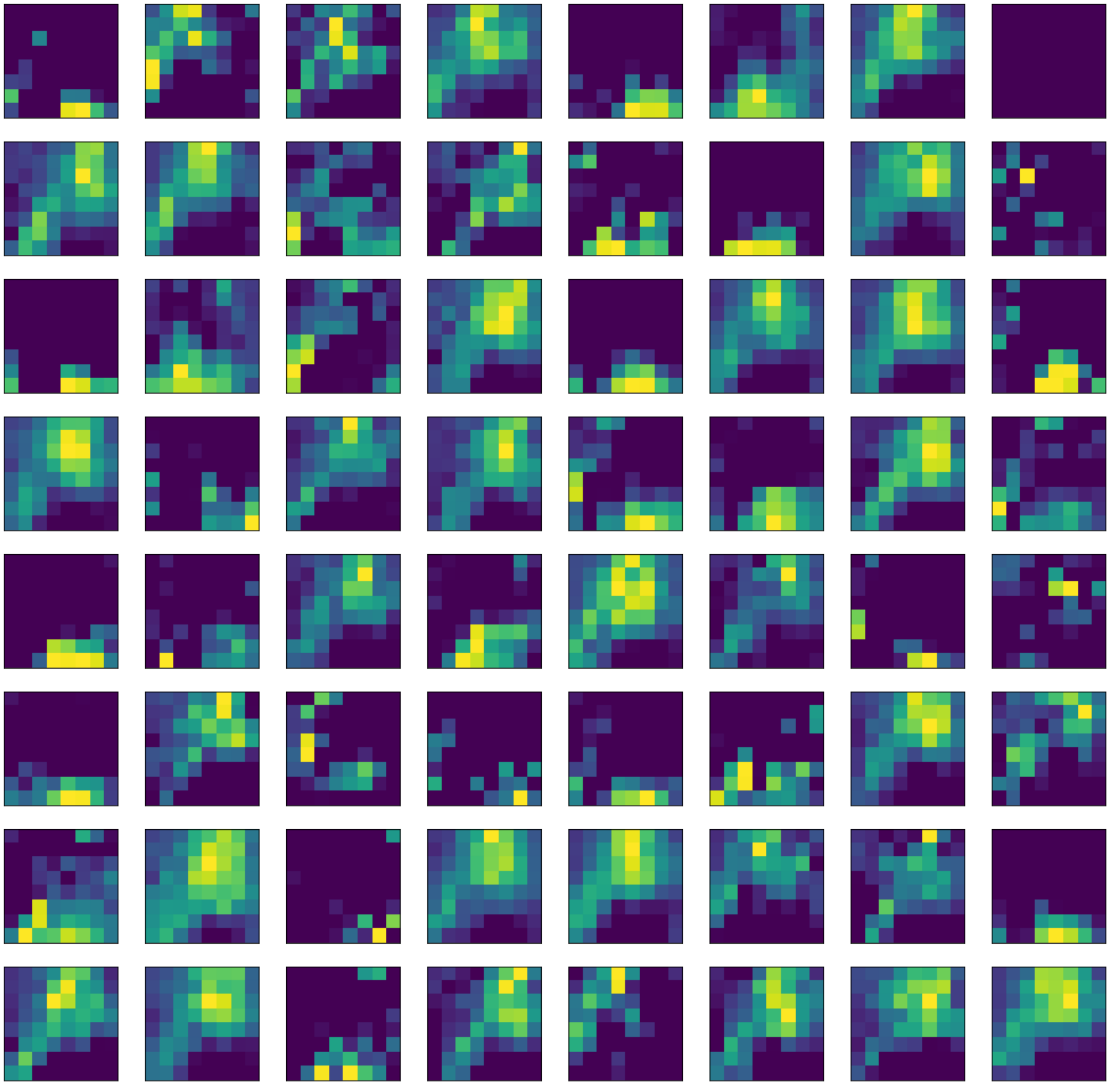
[91]:
```
from matplotlib import pyplot

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()
```









[ ]:

[ ]: