```
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
```

Download and explore the dataset

```
from google.colab import drive
drive.mount('/content/drive')
```

```
     Mounted at /content/drive
```

```
!unzip /content/drive/MyDrive/plant_and_weed_update/plants-leaf.zip -d leaf_photos
```

```
import pathlib
data_dir = "leaf_photos"
data_dir = pathlib.Path(data_dir)
```

dataset images count.

```
image_count = len(list(data_dir.glob('*/*.jpg')))
print(image_count)
```

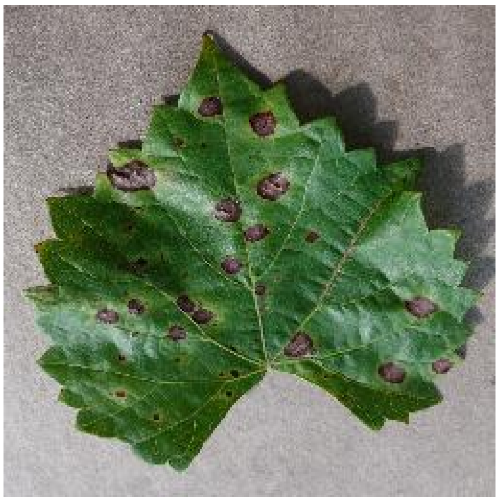
```
     1175
```

Here are some Apple___healthy:

```
Apple___healthy = list(data_dir.glob('Apple___healthy/*'))
PIL.Image.open(str(Apple___healthy[0]))
```

And some Grape___Black_rot:

```
Grape___Black_rot = list(data_dir.glob('Grape___Black_rot/*'))
PIL.Image.open(str(Grape___Black_rot[0]))
```



Load using keras.preprocessing

Let's load these images off disk using the helpful image_dataset_from_directory utility. This will take you from a directory of images on disk to a `tf.data.Dataset` in just a couple lines of code. If you like, you can also write your own data loading code from scratch by visiting the load images tutorial.

Create a dataset

Define some parameters for the loader:

```
batch_size = 32
img_height = 180
img_width = 180
```

It's good practice to use a validation split when developing your model. Let's use 80% of the images for training, and 20% for validation.

```
train_ds = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

```
Found 6268 files belonging to 16 classes.
Using 5015 files for training.
```

```
val_ds = tf.keras.preprocessing.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 6268 files belonging to 16 classes.
Using 1253 files for validation.
```

You can find the class names in the `class_names` attribute on these datasets. These correspond to the directory names in alphabetical order.

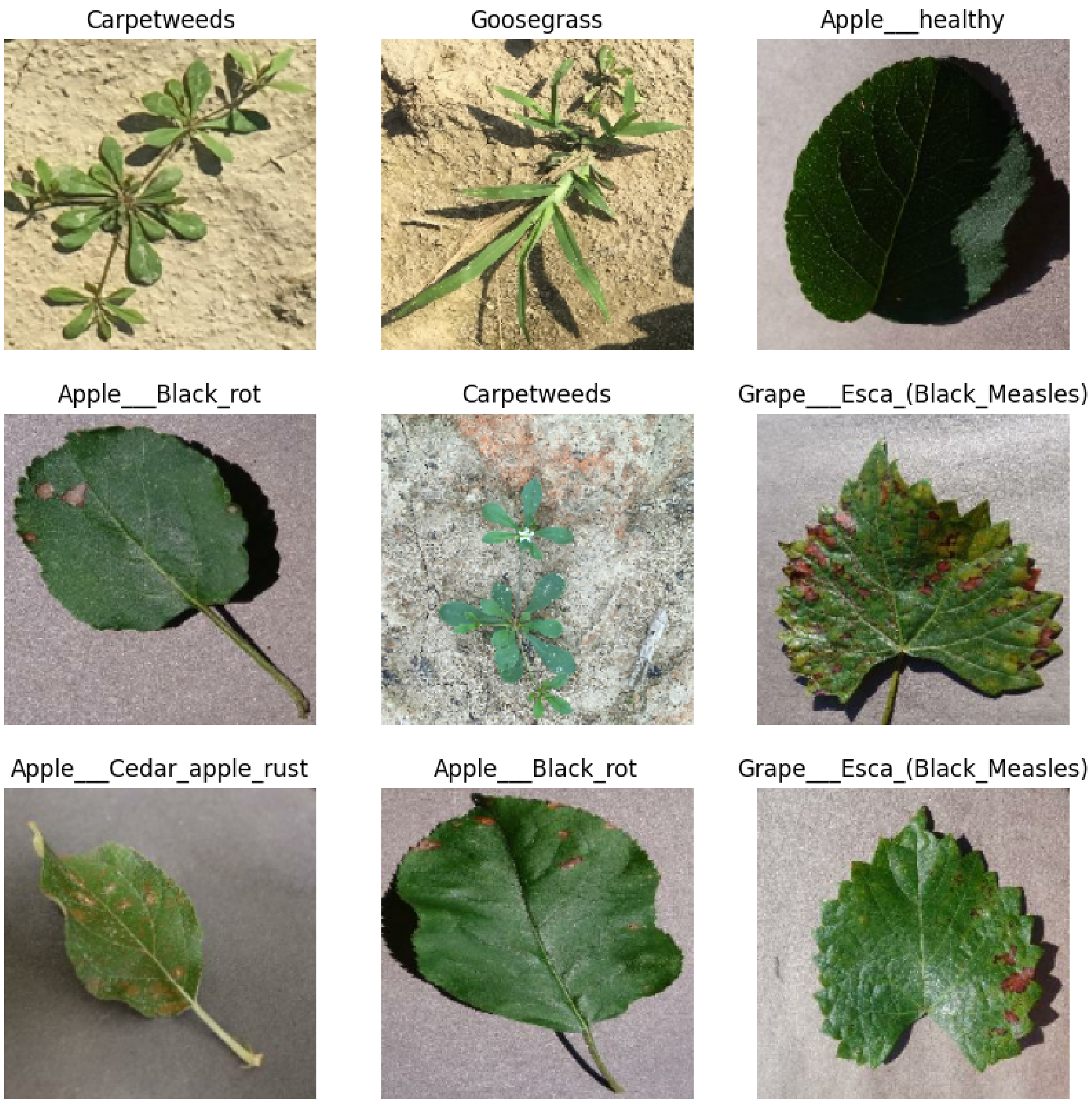
```
class_names = train_ds.class_names
print(class_names)
```

```
['Apple___Apple_scab', 'Apple___Black_rot', 'Apple___Cedar_apple_rust', 'Apple___healthy
```

Visualize the data

Here are the first 9 images from the training dataset.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(images[i].numpy().astype("uint8"))
    plt.title(class_names[labels[i]])
    plt.axis("off")
```



```
for image_batch, labels_batch in train_ds:
  print(image_batch.shape)
  print(labels_batch.shape)
  break
```

```
(32, 180, 180, 3)
(32,)
```

Configure the dataset for performance

Let's make sure to use buffered prefetching so you can yield data from disk without having I/O become blocking. These are two important methods you should use when loading data.

`Dataset.cache()` keeps the images in memory after they're loaded off disk during the first epoch. This will ensure the dataset does not become a bottleneck while training your model. If your dataset is too large to fit into memory, you can also use this method to create a performant on-disk cache.

`Dataset.prefetch()` overlaps data preprocessing and model execution while training.

```
AUTOTUNE = tf.data.experimental.AUTOTUNE

train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

Standardize the data

The RGB channel values are in the `[0, 255]` range. This is not ideal for a neural network; in general you should seek to make your input values small. Here, you will standardize values to be in the `[0, 1]` range by using a Rescaling layer.

```
normalization_layer = layers.experimental.preprocessing.Rescaling(1./255)
```

Note: The Keras Preprocessing utilities and layers introduced in this section are currently experimental and may change.

There are two ways to use this layer. You can apply it to the dataset by calling map:

```
normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
image_batch, labels_batch = next(iter(normalized_ds))
first_image = image_batch[0]
# Notice the pixels values are now in `[0,1]`.
print(np.min(first_image), np.max(first_image))
```

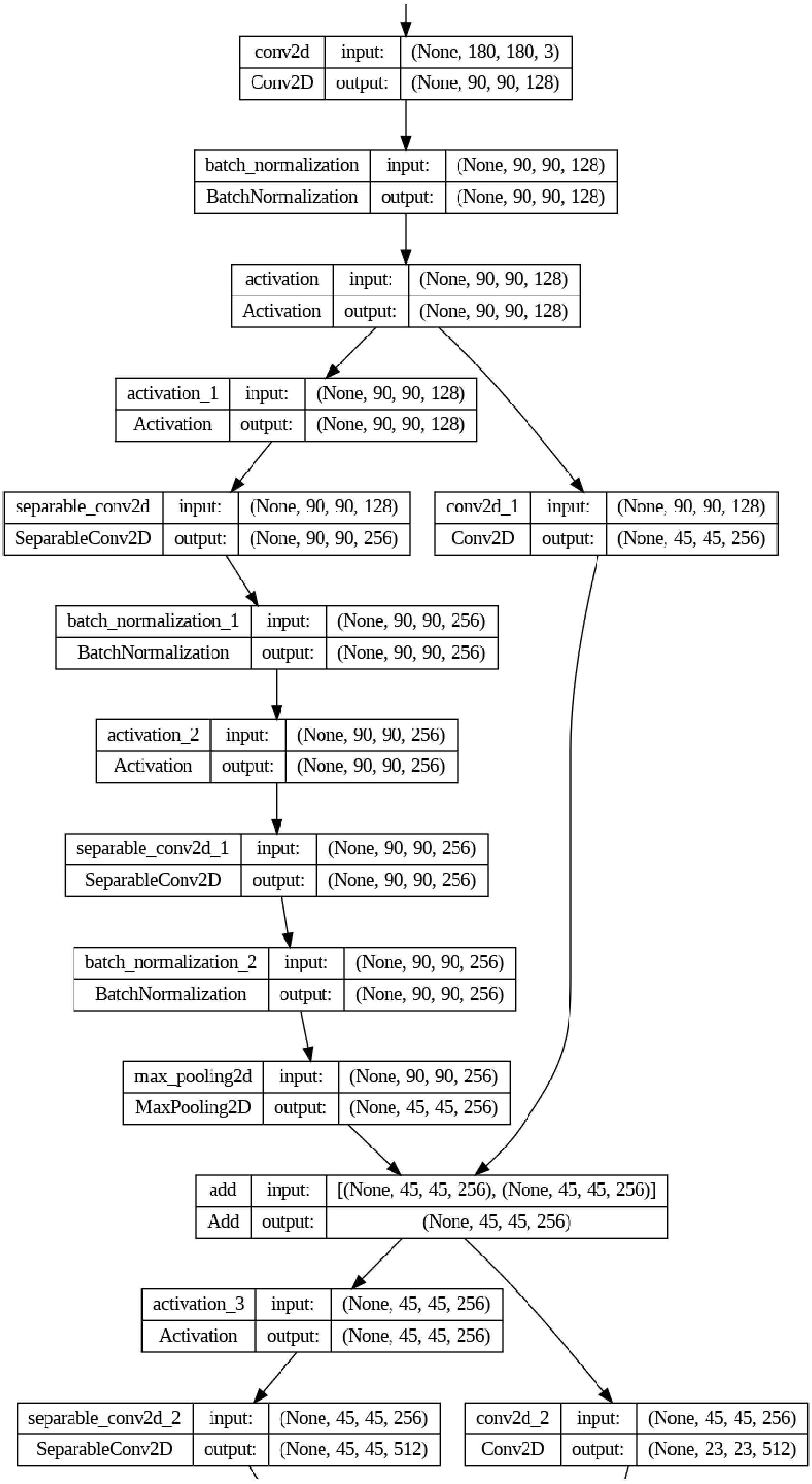
```
     0.0 0.97975516
```

Create the model

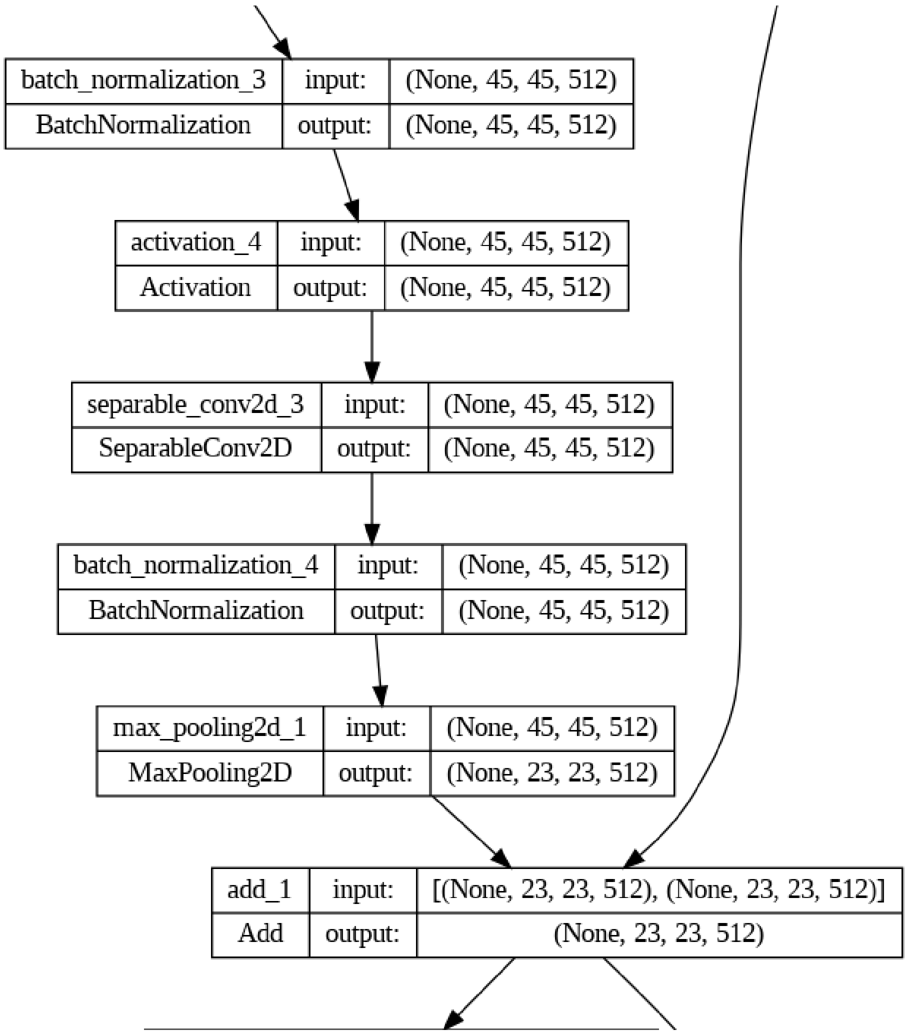
The model consists of three convolution blocks with a max pool layer in each of them. There's a fully connected layer with 128 units on top of it that is activated by a `relu` activation function. This model has not been tuned for high accuracy, the goal of this tutorial is to show a standard approach.

```python
def make_model(input_shape, num_classes):
    inputs = keras.Input(shape=input_shape)

    # Entry block
    x = layers.Rescaling(1.0 / 255)(inputs)
    x = layers.Conv2D(128, 3, strides=2, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    previous_block_activation = x  # Set aside residual

    for size in [256, 512, 728]:
        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(size, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(size, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

        # Project residual
        residual = layers.Conv2D(size, 1, strides=2, padding="same")(
            previous_block_activation
        )
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual

    x = layers.SeparableConv2D(1024, 3, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    x = layers.GlobalAveragePooling2D()(x)
    if num_classes == 2:
        units = 1
    else:
        units = num_classes

    x = layers.Dropout(0.25)(x)
    # We specify activation=None so as to return logits
    outputs = layers.Dense(units, activation=None)(x)
    return keras.Model(inputs, outputs)
```

```
num_classes = len(class_names)

model = make_model(input_shape=(180, 180) + (3,), num_classes=2)
keras.utils.plot_model(model, show_shapes=True)

# model = Sequential([
#   layers.experimental.preprocessing.Rescaling(1./255, input_shape=(img_height, img_width, 
#   layers.Conv2D(16, 3, padding='same', activation='relu'),
#   layers.MaxPooling2D(),
#   layers.Conv2D(32, 3, padding='same', activation='relu'),
#   layers.MaxPooling2D(),
#   layers.Conv2D(64, 3, padding='same', activation='relu'),
#   layers.MaxPooling2D(),
#   layers.Flatten(),
#   layers.Dense(128, activation='relu'),
#   layers.Dense(num_classes)
# ])
```



conv2d | input: | (None, 180, 180, 3)
Conv2D | output: | (None, 90, 90, 128)
batch_normalization | input: | (None, 90, 90, 128)
BatchNormalization | output: | (None, 90, 90, 128)
activation | input: | (None, 90, 90, 128)
Activation | output: | (None, 90, 90, 128)
activation_1 | input: | (None, 90, 90, 128)
Activation | output: | (None, 90, 90, 128)
separable_conv2d | input: | (None, 90, 90, 128)
SeparableConv2D | output: | (None, 90, 90, 256)
conv2d_1 | input: | (None, 90, 90, 128)
Conv2D | output: | (None, 45, 45, 256)
batch_normalization_1 | input: | (None, 90, 90, 256)
BatchNormalization | output: | (None, 90, 90, 256)
activation_2 | input: | (None, 90, 90, 256)
Activation | output: | (None, 90, 90, 256)
separable_conv2d_1 | input: | (None, 90, 90, 256)
SeparableConv2D | output: | (None, 90, 90, 256)
batch_normalization_2 | input: | (None, 90, 90, 256)
BatchNormalization | output: | (None, 90, 90, 256)
max_pooling2d | input: | (None, 90, 90, 256)
MaxPooling2D | output: | (None, 45, 45, 256)
add | input: | [(None, 45, 45, 256), (None, 45, 45, 256)]
Add | output: | (None, 45, 45, 256)
activation_3 | input: | (None, 45, 45, 256)
Activation | output: | (None, 45, 45, 256)
separable_conv2d_2 | input: | (None, 45, 45, 256)
SeparableConv2D | output: | (None, 45, 45, 512)
conv2d_2 | input: | (None, 45, 45, 256)
Conv2D | output: | (None, 23, 23, 512)


batch_normalization_3
input:
(None, 45, 45, 512)
BatchNormalization
output:
(None, 45, 45, 512)
activation_4
input:
(None, 45, 45, 512)
Activation
output:
(None, 45, 45, 512)
separable_conv2d_3
input:
(None, 45, 45, 512)
SeparableConv2D
output:
(None, 45, 45, 512)
batch_normalization_4
input:
(None, 45, 45, 512)
BatchNormalization
output:
(None, 45, 45, 512)
max_pooling2d_1
input:
(None, 45, 45, 512)
MaxPooling2D
output:
(None, 23, 23, 512)
add_1
input:
[(None, 23, 23, 512), (None, 23, 23, 512)]
Add
output:
(None, 23, 23, 512)

Compile the model

For this tutorial, choose the `optimizers.Adam` optimizer and `losses.SparseCategoricalCrossentropy` loss function. To view training and validation accuracy for each training epoch, pass the `metrics` argument.

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Model summary

View all the layers of the network using the model's `summary` method:

```
model.summary()
```

```
 add_2 (Add)                 (None, 12, 12, 728)          0         ['max_pooling2d_2[
                                                                     'conv2d_3[0][0]']

 separable_conv2d_6 (Separa  (None, 12, 12, 1024)         753048    ['add_2[0][0]']
 bleConv2D)

 batch_normalization_7 (Bat  (None, 12, 12, 1024)         4096      ['separable_conv2d_
 chNormalization)

 activation_7 (Activation)   (None, 12, 12, 1024)         0         ['batch_normalizat
                                                                    ]

 global_average_pooling2d (  (None, 1024)                 0         ['activation_7[0][
 GlobalAveragePooling2D)

 dropout (Dropout)           (None, 1024)                 0         ['global_average_p
                                                                    0]']

 dense (Dense)               (None, 1)                    1025      ['dropout[0][0]']

==================================================================================
Total params: 2731065 (10.42 MB)
Trainable params: 2722777 (10.39 MB)
Non-trainable params: 8288 (32.38 KB)
```

Train the model

entire processing by the learning algorithm of the entire train-set. The MNIST train set

```
epochs=30
history = model.fit(
  train_ds,
  validation_data=val_ds,
  epochs=epochs
)
```

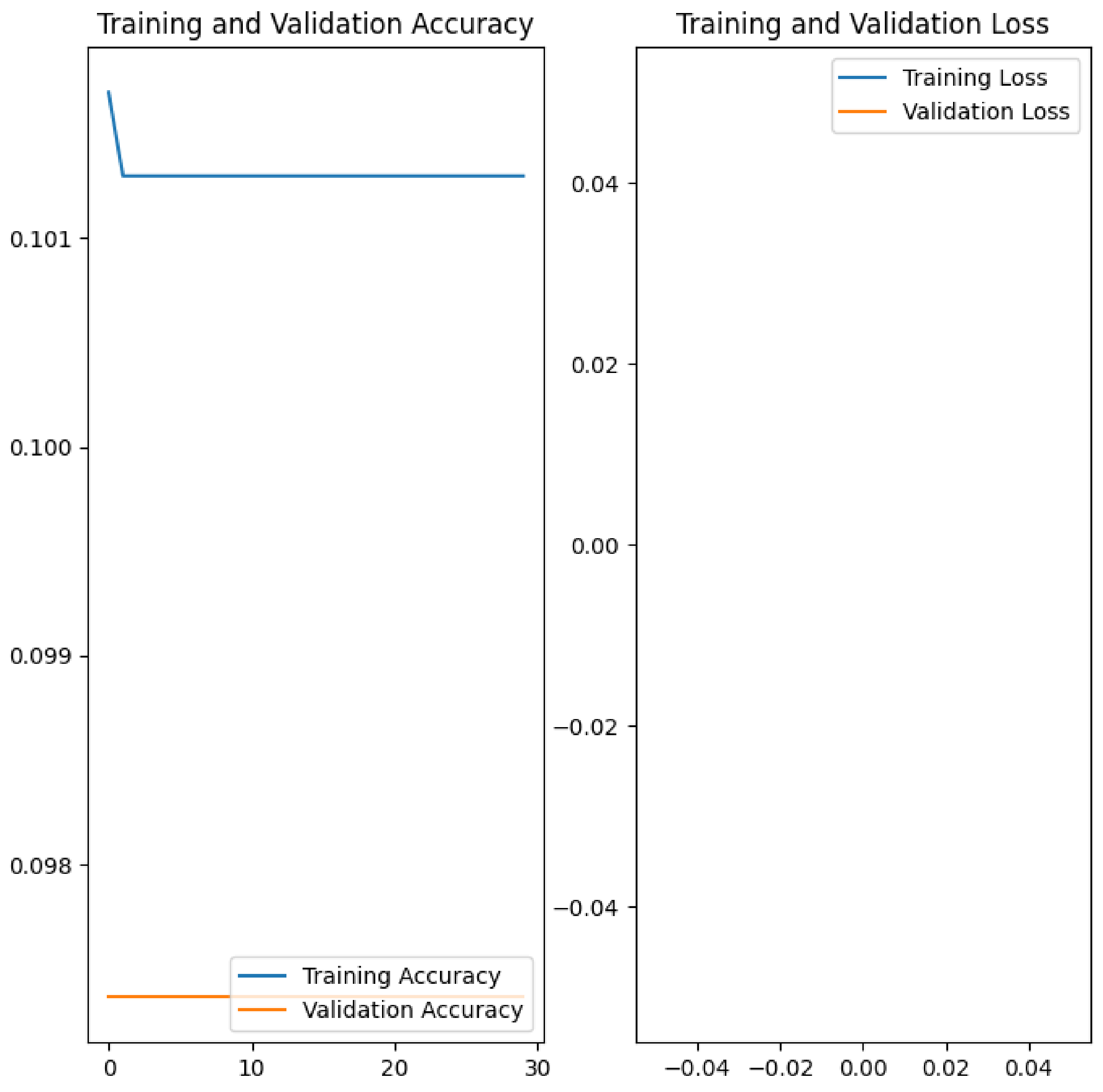
```
Epoch 2/30
157/157 [==============================] - 45s 288ms/step - loss: nan - accuracy: 0.10
Epoch 3/30
157/157 [==============================] - 45s 287ms/step - loss: nan - accuracy: 0.10
Epoch 4/30
157/157 [==============================] - 47s 301ms/step - loss: nan - accuracy: 0.10
Epoch 5/30
157/157 [==============================] - 47s 301ms/step - loss: nan - accuracy: 0.10
Epoch 6/30
157/157 [==============================] - 47s 300ms/step - loss: nan - accuracy: 0.10
Epoch 7/30
157/157 [==============================] - 47s 301ms/step - loss: nan - accuracy: 0.10
Epoch 8/30
157/157 [==============================] - 45s 286ms/step - loss: nan - accuracy: 0.10
Epoch 9/30
157/157 [==============================] - 45s 288ms/step - loss: nan - accuracy: 0.10
Epoch 10/30
157/157 [==============================] - 45s 285ms/step - loss: nan - accuracy: 0.10
Epoch 11/30
```

```
157/157 [==============================] - 45s 286ms/step - loss: nan - accuracy: 0.10
Epoch 13/30
157/157 [==============================] - 45s 287ms/step - loss: nan - accuracy: 0.10
Epoch 14/30
157/157 [==============================] - 45s 287ms/step - loss: nan - accuracy: 0.10
Epoch 15/30
157/157 [==============================] - 45s 287ms/step - loss: nan - accuracy: 0.10
Epoch 16/30
157/157 [==============================] - 45s 286ms/step - loss: nan - accuracy: 0.10
Epoch 17/30
157/157 [==============================] - 47s 300ms/step - loss: nan - accuracy: 0.10
Epoch 18/30
157/157 [==============================] - 45s 286ms/step - loss: nan - accuracy: 0.10
Epoch 19/30
157/157 [==============================] - 45s 287ms/step - loss: nan - accuracy: 0.10
Epoch 20/30
157/157 [==============================] - 45s 287ms/step - loss: nan - accuracy: 0.10
Epoch 21/30
157/157 [==============================] - 45s 286ms/step - loss: nan - accuracy: 0.10
Epoch 22/30
157/157 [==============================] - 45s 286ms/step - loss: nan - accuracy: 0.10
Epoch 23/30
157/157 [==============================] - 45s 286ms/step - loss: nan - accuracy: 0.10
Epoch 24/30
157/157 [==============================] - 47s 299ms/step - loss: nan - accuracy: 0.10
Epoch 25/30
157/157 [==============================] - 47s 300ms/step - loss: nan - accuracy: 0.10
Epoch 26/30
157/157 [==============================] - 47s 300ms/step - loss: nan - accuracy: 0.10
Epoch 27/30
157/157 [==============================] - 45s 286ms/step - loss: nan - accuracy: 0.10
Epoch 28/30
157/157 [==============================] - 45s 287ms/step - loss: nan - accuracy: 0.10
Epoch 29/30
157/157 [==============================] - 45s 285ms/step - loss: nan - accuracy: 0.10
Epoch 30/30
```

Visualize training results

Create plots of loss and accuracy on the training and validation sets.

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```



As you can see from the plots, training accuracy and validation accuracy are off by large margin and the model has achieved only around 60% accuracy on the validation set.

Let's look at what went wrong and try to increase the overall performance of the model.

Overfitting

In the plots above, the training accuracy is increasing linearly over time, whereas validation accuracy stalls around 60% in the training process. Also, the difference in accuracy between training and validation accuracy is noticeable—a sign of overfitting.

When there are a small number of training examples, the model sometimes learns from noises or unwanted details from training examples—to an extent that it negatively impacts the performance of the model on new examples. This phenomenon is known as overfitting. It means that the model will have a difficult time generalizing on a new dataset.

There are multiple ways to fight overfitting in the training process. In this tutorial, you'll use *data augmentation* and add *Dropout* to your model.

Data augmentation

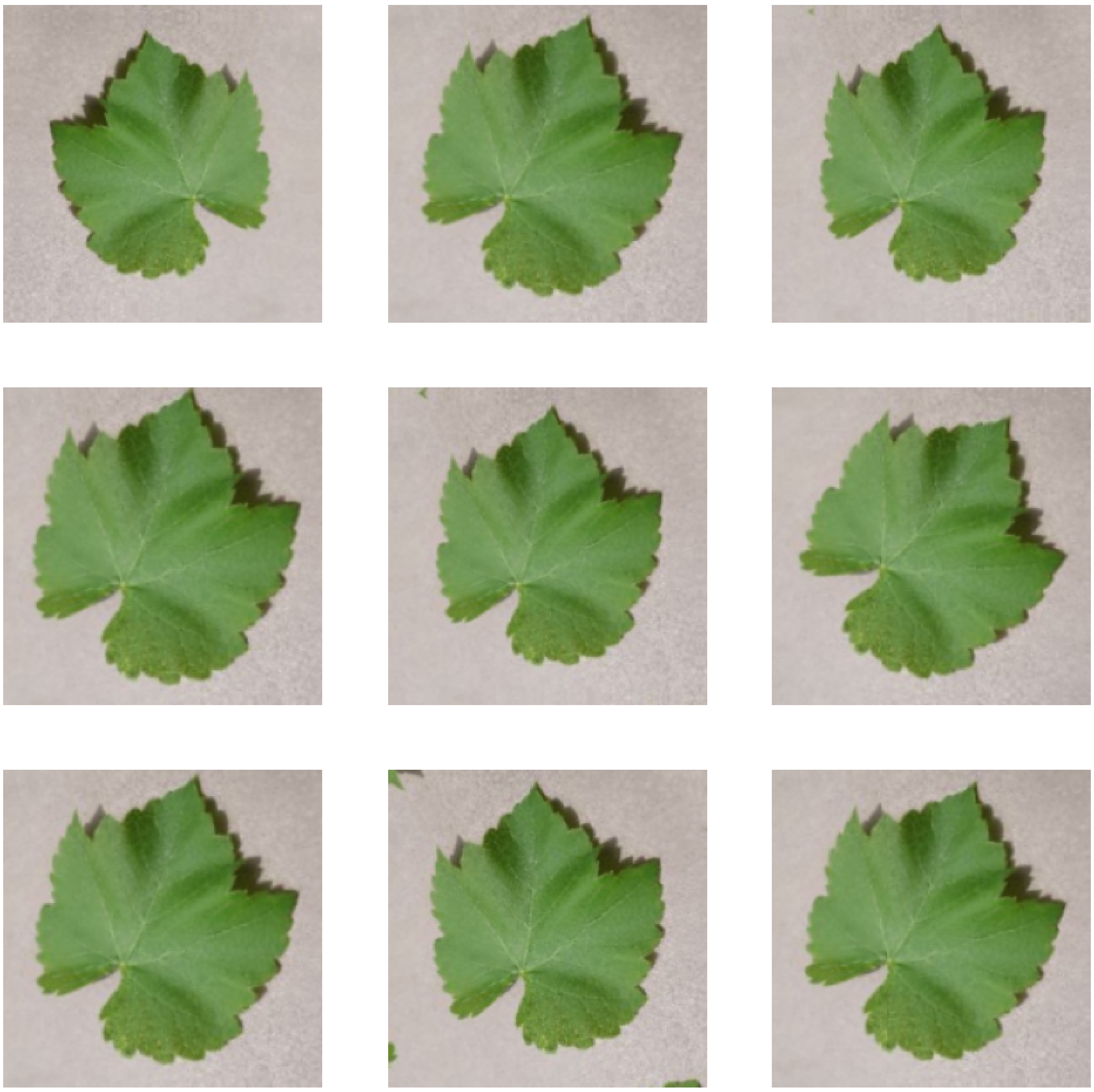
Overfitting generally occurs when there are a small number of training examples. Data augmentation takes the approach of generating additional training data from your existing examples by augmenting them using random transformations that yield believable-looking images. This helps expose the model to more aspects of the data and generalize better.

You will implement data augmentation using experimental Keras Preprocessing Layers. These can be included inside your model like other layers, and run on the GPU.

```
data_augmentation = keras.Sequential(
  [
    layers.experimental.preprocessing.RandomFlip("horizontal",
                                                 input_shape=(img_height,
                                                              img_width,
                                                              3)),
    layers.experimental.preprocessing.RandomRotation(0.1),
    layers.experimental.preprocessing.RandomZoom(0.1),
  ]
)
```

Let's visualize what a few augmented examples look like by applying data augmentation to the same image several times:

```
plt.figure(figsize=(10, 10))
for images, _ in train_ds.take(1):
  for i in range(9):
    augmented_images = data_augmentation(images)
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(augmented_images[0].numpy().astype("uint8"))
    plt.axis("off")
```



You will use data augmentation to train a model in a moment.

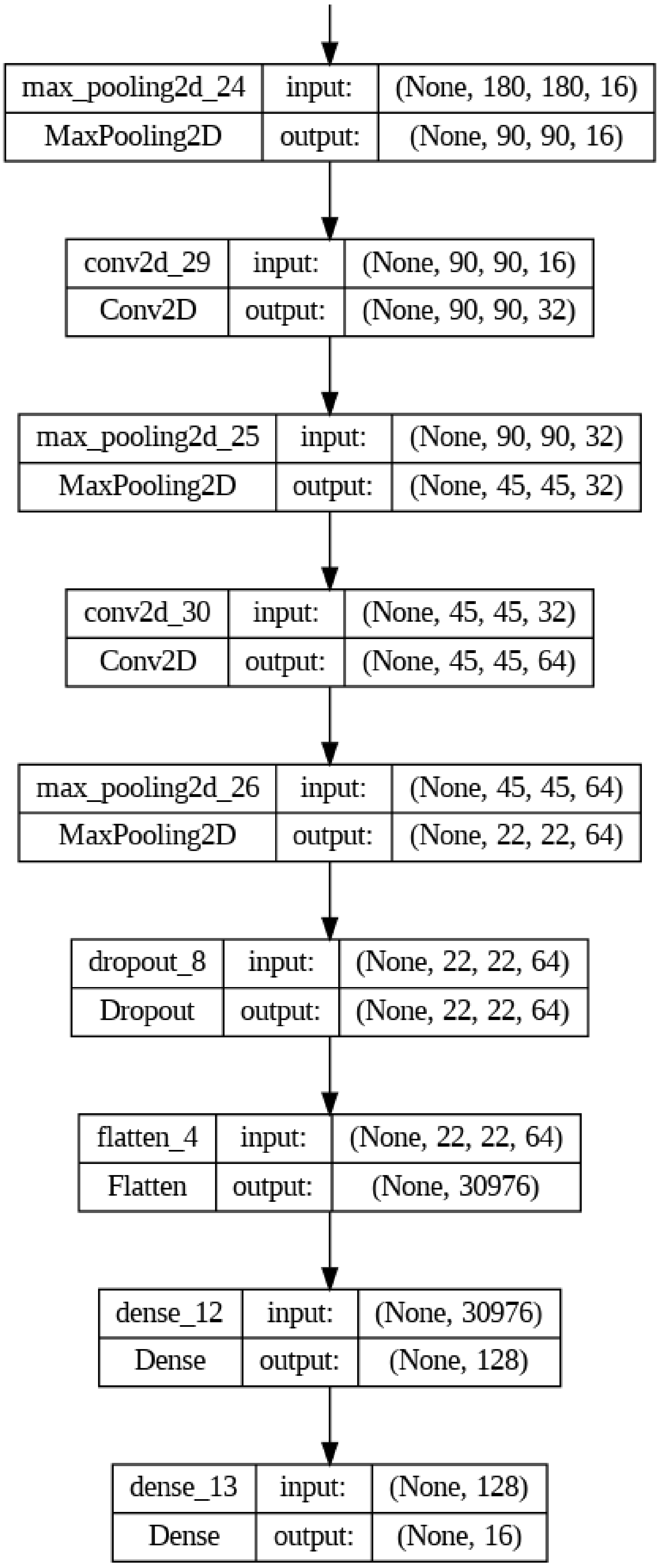
Dropout

Another technique to reduce overfitting is to introduce Dropout to the network, a form of *regularization*.

When you apply Dropout to a layer it randomly drops out (by setting the activation to zero) a number of output units from the layer during the training process. Dropout takes a fractional number as its input value, in the form such as 0.1, 0.2, 0.4, etc. This means dropping out 10%, 20% or 40% of the output units randomly from the applied layer.

Let's create a new neural network using `layers.Dropout`, then train it using augmented images.

```
model = Sequential([
  data_augmentation,
  layers.experimental.preprocessing.Rescaling(1./255),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Dropout(0.2),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])

# num_classes = len(class_names)

# model = make_model(input_shape=(180, 180) + (3,), num_classes=2)
keras.utils.plot_model(model, show_shapes=True)
```


max_pooling2d_24 | input: | (None, 180, 180, 16)
MaxPooling2D | output: | (None, 90, 90, 16)
conv2d_29 | input: | (None, 90, 90, 16)
Conv2D | output: | (None, 90, 90, 32)
max_pooling2d_25 | input: | (None, 90, 90, 32)
MaxPooling2D | output: | (None, 45, 45, 32)
conv2d_30 | input: | (None, 45, 45, 32)
Conv2D | output: | (None, 45, 45, 64)
max_pooling2d_26 | input: | (None, 45, 45, 64)
MaxPooling2D | output: | (None, 22, 22, 64)
dropout_8 | input: | (None, 22, 22, 64)
Dropout | output: | (None, 22, 22, 64)
flatten_4 | input: | (None, 22, 22, 64)
Flatten | output: | (None, 30976)
dense_12 | input: | (None, 30976)
Dense | output: | (None, 128)
dense_13 | input: | (None, 128)
Dense | output: | (None, 16)

Compile and train the model

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

```
model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 sequential_2 (Sequential)   (None, 180, 180, 3)       0         
                                                                 
 rescaling_9 (Rescaling)     (None, 180, 180, 3)       0         
                                                                 
 conv2d_28 (Conv2D)          (None, 180, 180, 16)      448       
                                                                 
 max_pooling2d_24 (MaxPooli  (None, 90, 90, 16)        0         
 ng2D)                                                           
                                                                 
 conv2d_29 (Conv2D)          (None, 90, 90, 32)        4640      
                                                                 
 max_pooling2d_25 (MaxPooli  (None, 45, 45, 32)        0         
 ng2D)                                                           
                                                                 
 conv2d_30 (Conv2D)          (None, 45, 45, 64)        18496     
                                                                 
 max_pooling2d_26 (MaxPooli  (None, 22, 22, 64)        0         
 ng2D)                                                           
                                                                 
 dropout_8 (Dropout)         (None, 22, 22, 64)        0         
                                                                 
 flatten_4 (Flatten)         (None, 30976)             0         
                                                                 
 dense_12 (Dense)            (None, 128)               3965056   
                                                                 
 dense_13 (Dense)            (None, 16)                2064      
```