l05c02_dogs_vs_cats_with_augmentation

September 3, 2021

1 Dogs vs Cats Image Classification With Image Augmentation

Run in Google Colab

View source on GitHub

In this tutorial, we will discuss how to classify images into pictures of cats or pictures of dogs. We'll build an image classifier using `tf.keras.Sequential` model and load data using `tf.keras.preprocessing.image.ImageDataGenerator`.

1.1 Specific concepts that will be covered:

In the process, we will build practical experience and develop intuition around the following concepts

- Building *data input pipelines* using the `tf.keras.preprocessing.image.ImageDataGenerator` class — How can we efficiently work with data on disk to interface with our model?
- *Overfitting* - what is it, how to identify it, and how can we prevent it?
- *Data Augmentation* and *Dropout* - Key techniques to fight overfitting in computer vision tasks that we will incorporate into our data pipeline and image classifier model.

1.2 We will follow the general machine learning workflow:

1. Examine and understand data
2. Build an input pipeline
3. Build our model
4. Train our model
5. Test our model

6. Improve our model/Repeat the process

Before you begin

Before running the code in this notebook, reset the runtime by going to **Runtime -> Reset all runtimes** in the menu above. If you have been working through several notebooks, this will help you avoid reaching Colab's memory limits.

2 Importing packages

Let's start by importing required packages:

- os — to read files and directory structure
- numpy — for some matrix math outside of TensorFlow
- matplotlib.pyplot — to plot the graph and display images in our training and validation data

[2]:
```
import tensorflow as tf
```

[3]:
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

[4]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
```

3 Data Loading

To build our image classifier, we begin by downloading the dataset. The dataset we are using is a filtered version of Dogs vs. Cats dataset from Kaggle (ultimately, this dataset is provided by Microsoft Research).

In previous Colabs, we've used TensorFlow Datasets, which is a very easy and convenient way to use datasets. In this Colab however, we will make use of the class `tf.keras.preprocessing.image.ImageDataGenerator` which will read data from disk. We therefore need to directly download *Dogs vs. Cats* from a URL and unzip it to the Colab filesystem.

[5]:
```
_URL = 'https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.
↪zip'

zip_dir = tf.keras.utils.get_file('cats_and_dogs_filterted.zip', origin=_URL,␣
↪extract=True)
```

```
Downloading data from https://storage.googleapis.com/mledu-
datasets/cats_and_dogs_filtered.zip
68608000/68606236 [==============================] - 1s 0us/step
68616192/68606236 [==============================] - 1s 0us/step
```

The dataset we have downloaded has following directory structure.

We'll now assign variables with the proper file path for the training and validation sets.

[6]:
```
base_dir = os.path.join(os.path.dirname(zip_dir), 'cats_and_dogs_filtered')
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
```

[7]:
```
train_cats_dir = os.path.join(train_dir, 'cats')  # directory with our training␣
↪cat pictures
train_dogs_dir = os.path.join(train_dir, 'dogs')  # directory with our training␣
↪dog pictures
validation_cats_dir = os.path.join(validation_dir, 'cats')  # directory with␣
↪our validation cat pictures
validation_dogs_dir = os.path.join(validation_dir, 'dogs')  # directory with␣
↪our validation dog pictures
```

3.0.1 Understanding our data

Let's look at how many cats and dogs images we have in our training and validation directory

[8]:
```
num_cats_tr = len(os.listdir(train_cats_dir))
num_dogs_tr = len(os.listdir(train_dogs_dir))

num_cats_val = len(os.listdir(validation_cats_dir))
num_dogs_val = len(os.listdir(validation_dogs_dir))

total_train = num_cats_tr + num_dogs_tr
total_val = num_cats_val + num_dogs_val
```

[9]:
```
print('total training cat images:', num_cats_tr)
print('total training dog images:', num_dogs_tr)

print('total validation cat images:', num_cats_val)
print('total validation dog images:', num_dogs_val)
print("--")
print("Total training images:", total_train)
print("Total validation images:", total_val)
```

```
total training cat images: 1000
total training dog images: 1000
total validation cat images: 500
total validation dog images: 500
--
Total training images: 2000
Total validation images: 1000
```

4 Setting Model Parameters

For convenience, let us set up variables that will be used later while pre-processing our dataset and training our network.

[10]:
```
BATCH_SIZE = 100
IMG_SHAPE  = 150 # Our training data consists of images with width of 150␣
 ↪pixels and height of 150 pixels
```

After defining our generators for training and validation images, **flow_from_directory** method will load images from the disk and will apply rescaling and will resize them into required dimensions using single line of code.

5 Data Augmentation

Overfitting often occurs when we have a small number of training examples. One way to fix this problem is to augment our dataset so that it has sufficient number and variety of training examples. Data augmentation takes the approach of generating more training data from existing training samples, by augmenting the samples through random transformations that yield believable-looking images. The goal is that at training time, your model will never see the exact same picture twice. This exposes the model to more aspects of the data, allowing it to generalize better.

In **tf.keras** we can implement this using the same **ImageDataGenerator** class we used before. We can simply pass different transformations we would want to our dataset as a form of arguments and it will take care of applying it to the dataset during our training process.

To start off, let's define a function that can display an image, so we can see the type of augmentation that has been performed. Then, we'll look at specific augmentations that we'll use during training.

[11]:
```
# This function will plot images in the form of a grid with 1 row and 5 columns␣
 ↪where images are placed in each column.
def plotImages(images_arr):
    fig, axes = plt.subplots(1, 5, figsize=(20,20))
    axes = axes.flatten()
    for img, ax in zip(images_arr, axes):
        ax.imshow(img)
    plt.tight_layout()
    plt.show()
```

5.0.1 Flipping the image horizontally

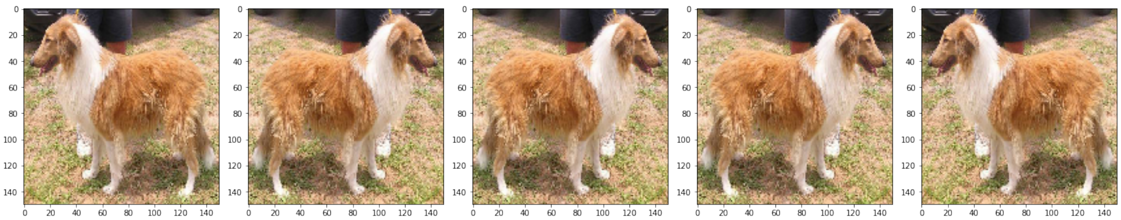
We can begin by randomly applying horizontal flip augmentation to our dataset and seeing how individual images will look after the transformation. This is achieved by passing `horizontal_flip=True` as an argument to the `ImageDataGenerator` class.

[12]:
```
image_gen = ImageDataGenerator(rescale=1./255, horizontal_flip=True)

train_data_gen = image_gen.flow_from_directory(batch_size=BATCH_SIZE,
                                               directory=train_dir,
                                               shuffle=True,
                                              ␣
 ↪target_size=(IMG_SHAPE,IMG_SHAPE))
```

```
Found 2000 images belonging to 2 classes.
```

To see the transformation in action, let's take one sample image from our training set and repeat it five times. The augmentation will be randomly applied (or not) to each repetition.

[13]:
```
augmented_images = [train_data_gen[0][0][0] for i in range(5)]
plotImages(augmented_images)
```



5.0.2 Rotating the image

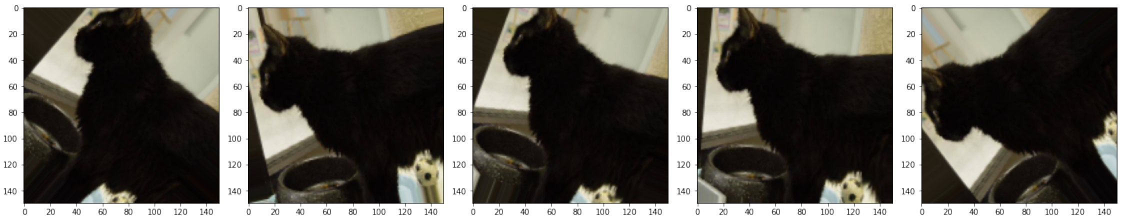
The rotation augmentation will randomly rotate the image up to a specified number of degrees. Here, we'll set it to 45.

[14]:
```
image_gen = ImageDataGenerator(rescale=1./255, rotation_range=45)

train_data_gen = image_gen.flow_from_directory(batch_size=BATCH_SIZE,
                                               directory=train_dir,
                                               shuffle=True,
                                               target_size=(IMG_SHAPE,␣
 ↪IMG_SHAPE))
```

```
Found 2000 images belonging to 2 classes.
```

To see the transformation in action, let's once again take a sample image from our training set and repeat it. The augmentation will be randomly applied (or not) to each repetition.

[15]:
```
augmented_images = [train_data_gen[0][0][0] for i in range(5)]
plotImages(augmented_images)
```



5.0.3 Applying Zoom

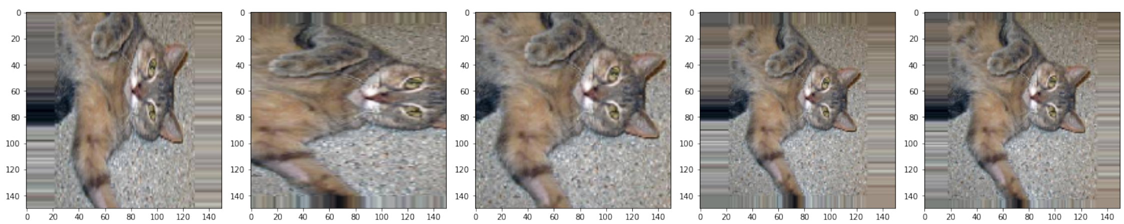
We can also apply Zoom augmentation to our dataset, zooming images up to 50% randomly.

[16]:
```
image_gen = ImageDataGenerator(rescale=1./255, zoom_range=0.5)

train_data_gen = image_gen.flow_from_directory(batch_size=BATCH_SIZE,
                                               directory=train_dir,
                                               shuffle=True,
                                               target_size=(IMG_SHAPE,␣
 ↪IMG_SHAPE))
```

```
Found 2000 images belonging to 2 classes.
```

One more time, take a sample image from our training set and repeat it. The augmentation will be randomly applied (or not) to each repetition.

[17]:
```
augmented_images = [train_data_gen[0][0][0] for i in range(5)]
plotImages(augmented_images)
```



5.0.4 Putting it all together

We can apply all these augmentations, and even others, with just one line of code, by passing the augmentations as arguments with proper values.

Here, we have applied rescale, rotation of 45 degrees, width shift, height shift, horizontal flip, and zoom augmentation to our training images.

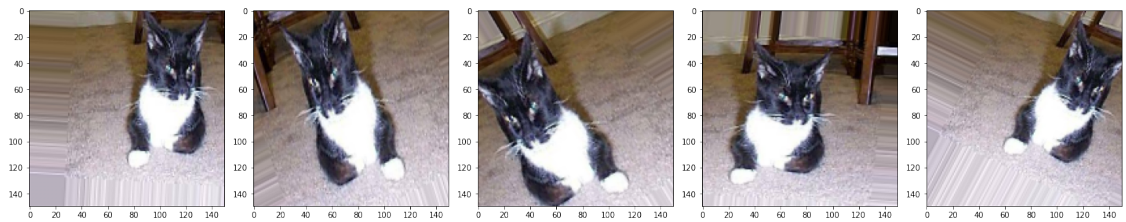
[18]:
```
image_gen_train = ImageDataGenerator(
      rescale=1./255,
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')

train_data_gen = image_gen_train.flow_from_directory(batch_size=BATCH_SIZE,
                                                     directory=train_dir,
                                                     shuffle=True,
                                                     ␣
 ↪target_size=(IMG_SHAPE,IMG_SHAPE),
```

```
                                        class_mode='binary')
```

```
Found 2000 images belonging to 2 classes.
```

Let's visualize how a single image would look like five different times, when we pass these augmentations randomly to our dataset.

[19]:
```
augmented_images = [train_data_gen[0][0][0] for i in range(5)]
plotImages(augmented_images)
```



5.0.5 Creating Validation Data generator

Generally, we only apply data augmentation to our training examples, since the original images should be representative of what our model needs to manage. So, in this case we are only rescaling our validation images and converting them into batches using ImageDataGenerator.

[20]:
```
image_gen_val = ImageDataGenerator(rescale=1./255)

val_data_gen = image_gen_val.flow_from_directory(batch_size=BATCH_SIZE,
                                                 directory=validation_dir,
                                                 target_size=(IMG_SHAPE,␣
 ↪IMG_SHAPE),
                                                 class_mode='binary')
```

```
Found 1000 images belonging to 2 classes.
```

6 Model Creation

6.1 Define the model

The model consists of four convolution blocks with a max pool layer in each of them.

Before the final Dense layers, we're also applying a Dropout probability of 0.5. It means that 50% of the values coming into the Dropout layer will be set to zero. This helps to prevent overfitting.

Then we have a fully connected layer with 512 units, with a `relu` activation function. The model will output class probabilities for two classes — dogs and cats — using `softmax`.

[21]:
```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(150, 150,␣
 ↪3)),
```

```
    tf.keras.layers.MaxPooling2D(2, 2),

    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(2)
])
```

6.1.1 Compiling the model

As usual, we will use the `adam` optimizer. Since we output a softmax categorization, we'll use `sparse_categorical_crossentropy` as the loss function. We would also like to look at training and validation accuracy on each epoch as we train our network, so we are passing in the metrics argument.

[22]:
```
model.compile(optimizer='adam',
              loss=tf.keras.losses.
 ↪SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

6.1.2 Model Summary

Let's look at all the layers of our network using **summary** method.

[23]:
```
model.summary()
```

```
Model: "sequential"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 34, 34, 128)       73856
```

```
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 17, 17, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 7, 7, 128)         0
_________________________________________________________________
dropout (Dropout)            (None, 7, 7, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 6272)              0
_________________________________________________________________
dense (Dense)                (None, 512)               3211776
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 1026
=================================================================
Total params: 3,453,634
Trainable params: 3,453,634
Non-trainable params: 0
_________________________________________________________________
```

6.1.3 Train the model

It's time we train our network.

Since our batches are coming from a generator (`ImageDataGenerator`), we'll use `fit_generator` instead of `fit`.

[24]:
```python
epochs=100
history = model.fit_generator(
    train_data_gen,
    steps_per_epoch=int(np.ceil(total_train / float(BATCH_SIZE))),
    epochs=epochs,
    validation_data=val_data_gen,
    validation_steps=int(np.ceil(total_val / float(BATCH_SIZE)))
)
```

```
/usr/local/lib/python3.7/dist-packages/keras/engine/training.py:1972:
UserWarning: `Model.fit_generator` is deprecated and will be removed in a future
version. Please use `Model.fit`, which supports generators.
  warnings.warn('`Model.fit_generator` is deprecated and '
```

```
Epoch 1/100
20/20 [==============================] - 51s 1s/step - loss: 0.7101 - accuracy:
0.5130 - val_loss: 0.7005 - val_accuracy: 0.5000
Epoch 2/100
20/20 [==============================] - 21s 1s/step - loss: 0.6943 - accuracy:
0.5165 - val_loss: 0.6914 - val_accuracy: 0.5010
Epoch 3/100
20/20 [==============================] - 20s 1s/step - loss: 0.6894 - accuracy:
```

```
0.5340 - val_loss: 0.6851 - val_accuracy: 0.5030
Epoch 4/100
20/20 [==============================] - 20s 1s/step - loss: 0.6754 - accuracy:
0.5665 - val_loss: 0.6451 - val_accuracy: 0.6330
Epoch 5/100
20/20 [==============================] - 20s 1s/step - loss: 0.6570 - accuracy:
0.6020 - val_loss: 0.6254 - val_accuracy: 0.6260
Epoch 6/100
20/20 [==============================] - 20s 1s/step - loss: 0.6502 - accuracy:
0.6085 - val_loss: 0.6891 - val_accuracy: 0.5360
Epoch 7/100
20/20 [==============================] - 20s 1s/step - loss: 0.6575 - accuracy:
0.5760 - val_loss: 0.6240 - val_accuracy: 0.6340
Epoch 8/100
20/20 [==============================] - 20s 1s/step - loss: 0.6450 - accuracy:
0.6165 - val_loss: 0.6331 - val_accuracy: 0.6000
Epoch 9/100
20/20 [==============================] - 20s 1s/step - loss: 0.6471 - accuracy:
0.5965 - val_loss: 0.6160 - val_accuracy: 0.6600
Epoch 10/100
20/20 [==============================] - 20s 1s/step - loss: 0.6413 - accuracy:
0.6315 - val_loss: 0.6186 - val_accuracy: 0.6300
Epoch 11/100
20/20 [==============================] - 20s 1s/step - loss: 0.6421 - accuracy:
0.6145 - val_loss: 0.6262 - val_accuracy: 0.6360
Epoch 12/100
20/20 [==============================] - 20s 1s/step - loss: 0.6249 - accuracy:
0.6295 - val_loss: 0.6096 - val_accuracy: 0.6450
Epoch 13/100
20/20 [==============================] - 20s 1s/step - loss: 0.6211 - accuracy:
0.6535 - val_loss: 0.5975 - val_accuracy: 0.6770
Epoch 14/100
20/20 [==============================] - 20s 1s/step - loss: 0.6360 - accuracy:
0.6395 - val_loss: 0.6276 - val_accuracy: 0.6330
Epoch 15/100
20/20 [==============================] - 20s 1s/step - loss: 0.6124 - accuracy:
0.6575 - val_loss: 0.5773 - val_accuracy: 0.6790
Epoch 16/100
20/20 [==============================] - 20s 1s/step - loss: 0.6019 - accuracy:
0.6740 - val_loss: 0.5737 - val_accuracy: 0.6910
Epoch 17/100
20/20 [==============================] - 20s 1s/step - loss: 0.5855 - accuracy:
0.6925 - val_loss: 0.5724 - val_accuracy: 0.6950
Epoch 18/100
20/20 [==============================] - 20s 1s/step - loss: 0.5747 - accuracy:
0.7065 - val_loss: 0.5598 - val_accuracy: 0.7040
Epoch 19/100
20/20 [==============================] - 20s 1s/step - loss: 0.5908 - accuracy:
```

```
0.6695 - val_loss: 0.5840 - val_accuracy: 0.6750
Epoch 20/100
20/20 [==============================] - 20s 1s/step - loss: 0.5773 - accuracy:
0.7030 - val_loss: 0.5499 - val_accuracy: 0.7220
Epoch 21/100
20/20 [==============================] - 20s 1s/step - loss: 0.5918 - accuracy:
0.6895 - val_loss: 0.5730 - val_accuracy: 0.7050
Epoch 22/100
20/20 [==============================] - 20s 1s/step - loss: 0.5803 - accuracy:
0.7100 - val_loss: 0.5456 - val_accuracy: 0.7170
Epoch 23/100
20/20 [==============================] - 20s 1s/step - loss: 0.5670 - accuracy:
0.7075 - val_loss: 0.5606 - val_accuracy: 0.7140
Epoch 24/100
20/20 [==============================] - 20s 995ms/step - loss: 0.5906 -
accuracy: 0.6760 - val_loss: 0.6053 - val_accuracy: 0.6940
Epoch 25/100
20/20 [==============================] - 20s 994ms/step - loss: 0.5495 -
accuracy: 0.7325 - val_loss: 0.5208 - val_accuracy: 0.7430
Epoch 26/100
20/20 [==============================] - 20s 1s/step - loss: 0.5553 - accuracy:
0.7150 - val_loss: 0.5096 - val_accuracy: 0.7490
Epoch 27/100
20/20 [==============================] - 20s 1s/step - loss: 0.5401 - accuracy:
0.7325 - val_loss: 0.5525 - val_accuracy: 0.7300
Epoch 28/100
20/20 [==============================] - 20s 1s/step - loss: 0.5503 - accuracy:
0.7195 - val_loss: 0.5311 - val_accuracy: 0.7400
Epoch 29/100
20/20 [==============================] - 20s 1s/step - loss: 0.5625 - accuracy:
0.6995 - val_loss: 0.5234 - val_accuracy: 0.7290
Epoch 30/100
20/20 [==============================] - 20s 998ms/step - loss: 0.5299 -
accuracy: 0.7340 - val_loss: 0.5102 - val_accuracy: 0.7430
Epoch 31/100
20/20 [==============================] - 20s 1s/step - loss: 0.5238 - accuracy:
0.7395 - val_loss: 0.4947 - val_accuracy: 0.7520
Epoch 32/100
20/20 [==============================] - 20s 1s/step - loss: 0.5088 - accuracy:
0.7520 - val_loss: 0.5031 - val_accuracy: 0.7490
Epoch 33/100
20/20 [==============================] - 20s 1s/step - loss: 0.5197 - accuracy:
0.7385 - val_loss: 0.4907 - val_accuracy: 0.7540
Epoch 34/100
20/20 [==============================] - 20s 987ms/step - loss: 0.5021 -
accuracy: 0.7520 - val_loss: 0.4793 - val_accuracy: 0.7640
Epoch 35/100
20/20 [==============================] - 20s 993ms/step - loss: 0.4994 -
```

```
accuracy: 0.7570 - val_loss: 0.4842 - val_accuracy: 0.7530
Epoch 36/100
20/20 [==============================] - 20s 995ms/step - loss: 0.5115 -
accuracy: 0.7470 - val_loss: 0.5004 - val_accuracy: 0.7460
Epoch 37/100
20/20 [==============================] - 20s 981ms/step - loss: 0.5177 -
accuracy: 0.7355 - val_loss: 0.5016 - val_accuracy: 0.7580
Epoch 38/100
20/20 [==============================] - 20s 994ms/step - loss: 0.4931 -
accuracy: 0.7685 - val_loss: 0.4851 - val_accuracy: 0.7580
Epoch 39/100
20/20 [==============================] - 20s 987ms/step - loss: 0.5031 -
accuracy: 0.7445 - val_loss: 0.4873 - val_accuracy: 0.7640
Epoch 40/100
20/20 [==============================] - 20s 992ms/step - loss: 0.4680 -
accuracy: 0.7725 - val_loss: 0.4652 - val_accuracy: 0.7690
Epoch 41/100
20/20 [==============================] - 20s 989ms/step - loss: 0.4737 -
accuracy: 0.7690 - val_loss: 0.4817 - val_accuracy: 0.7650
Epoch 42/100
20/20 [==============================] - 20s 991ms/step - loss: 0.4979 -
accuracy: 0.7585 - val_loss: 0.4839 - val_accuracy: 0.7640
Epoch 43/100
20/20 [==============================] - 20s 993ms/step - loss: 0.4698 -
accuracy: 0.7705 - val_loss: 0.4738 - val_accuracy: 0.7660
Epoch 44/100
20/20 [==============================] - 20s 991ms/step - loss: 0.4933 -
accuracy: 0.7635 - val_loss: 0.4646 - val_accuracy: 0.7720
Epoch 45/100
20/20 [==============================] - 20s 984ms/step - loss: 0.4609 -
accuracy: 0.7775 - val_loss: 0.4522 - val_accuracy: 0.7880
Epoch 46/100
20/20 [==============================] - 20s 990ms/step - loss: 0.4633 -
accuracy: 0.7915 - val_loss: 0.4544 - val_accuracy: 0.7860
Epoch 47/100
20/20 [==============================] - 20s 1s/step - loss: 0.4586 - accuracy:
0.7860 - val_loss: 0.4551 - val_accuracy: 0.7940
Epoch 48/100
20/20 [==============================] - 20s 1s/step - loss: 0.4578 - accuracy:
0.7775 - val_loss: 0.5032 - val_accuracy: 0.7580
Epoch 49/100
20/20 [==============================] - 20s 1s/step - loss: 0.4474 - accuracy:
0.7930 - val_loss: 0.4796 - val_accuracy: 0.7690
Epoch 50/100
20/20 [==============================] - 20s 1s/step - loss: 0.4538 - accuracy:
0.7855 - val_loss: 0.4441 - val_accuracy: 0.7890
Epoch 51/100
20/20 [==============================] - 20s 987ms/step - loss: 0.4483 -
```

```
accuracy: 0.7920 - val_loss: 0.4404 - val_accuracy: 0.7980
Epoch 52/100
20/20 [==============================] - 20s 992ms/step - loss: 0.4341 -
accuracy: 0.7970 - val_loss: 0.4887 - val_accuracy: 0.7560
Epoch 53/100
20/20 [==============================] - 20s 1s/step - loss: 0.4605 - accuracy:
0.7840 - val_loss: 0.4325 - val_accuracy: 0.7940
Epoch 54/100
20/20 [==============================] - 20s 1s/step - loss: 0.4268 - accuracy:
0.8065 - val_loss: 0.4343 - val_accuracy: 0.7990
Epoch 55/100
20/20 [==============================] - 20s 1s/step - loss: 0.4351 - accuracy:
0.7955 - val_loss: 0.4290 - val_accuracy: 0.8060
Epoch 56/100
20/20 [==============================] - 20s 1s/step - loss: 0.4094 - accuracy:
0.8080 - val_loss: 0.4530 - val_accuracy: 0.7890
Epoch 57/100
20/20 [==============================] - 20s 1s/step - loss: 0.4195 - accuracy:
0.7915 - val_loss: 0.4385 - val_accuracy: 0.7860
Epoch 58/100
20/20 [==============================] - 20s 1s/step - loss: 0.4178 - accuracy:
0.8125 - val_loss: 0.4418 - val_accuracy: 0.8030
Epoch 59/100
20/20 [==============================] - 20s 1000ms/step - loss: 0.3909 -
accuracy: 0.8275 - val_loss: 0.4290 - val_accuracy: 0.8060
Epoch 60/100
20/20 [==============================] - 20s 992ms/step - loss: 0.4260 -
accuracy: 0.8095 - val_loss: 0.4357 - val_accuracy: 0.7840
Epoch 61/100
20/20 [==============================] - 20s 1s/step - loss: 0.3887 - accuracy:
0.8265 - val_loss: 0.4582 - val_accuracy: 0.7760
Epoch 62/100
20/20 [==============================] - 21s 1s/step - loss: 0.4115 - accuracy:
0.8130 - val_loss: 0.4007 - val_accuracy: 0.8120
Epoch 63/100
20/20 [==============================] - 21s 1s/step - loss: 0.3831 - accuracy:
0.8240 - val_loss: 0.4110 - val_accuracy: 0.8040
Epoch 64/100
20/20 [==============================] - 21s 1s/step - loss: 0.3925 - accuracy:
0.8190 - val_loss: 0.4164 - val_accuracy: 0.8130
Epoch 65/100
20/20 [==============================] - 20s 1s/step - loss: 0.4006 - accuracy:
0.8210 - val_loss: 0.4450 - val_accuracy: 0.7980
Epoch 66/100
20/20 [==============================] - 20s 1s/step - loss: 0.4117 - accuracy:
0.8140 - val_loss: 0.4121 - val_accuracy: 0.8070
Epoch 67/100
20/20 [==============================] - 20s 1s/step - loss: 0.3944 - accuracy:
```

```
0.8150 - val_loss: 0.4522 - val_accuracy: 0.7790
Epoch 68/100
20/20 [==============================] - 20s 1s/step - loss: 0.3828 - accuracy:
0.8145 - val_loss: 0.4177 - val_accuracy: 0.8000
Epoch 69/100
20/20 [==============================] - 20s 1s/step - loss: 0.3916 - accuracy:
0.8270 - val_loss: 0.4213 - val_accuracy: 0.7990
Epoch 70/100
20/20 [==============================] - 20s 1s/step - loss: 0.3822 - accuracy:
0.8245 - val_loss: 0.3944 - val_accuracy: 0.8170
Epoch 71/100
20/20 [==============================] - 21s 1s/step - loss: 0.3779 - accuracy:
0.8335 - val_loss: 0.4280 - val_accuracy: 0.7900
Epoch 72/100
20/20 [==============================] - 20s 1s/step - loss: 0.3627 - accuracy:
0.8405 - val_loss: 0.4005 - val_accuracy: 0.8200
Epoch 73/100
20/20 [==============================] - 20s 996ms/step - loss: 0.3443 -
accuracy: 0.8535 - val_loss: 0.4177 - val_accuracy: 0.8050
Epoch 74/100
20/20 [==============================] - 20s 993ms/step - loss: 0.3610 -
accuracy: 0.8405 - val_loss: 0.4732 - val_accuracy: 0.7890
Epoch 75/100
20/20 [==============================] - 20s 1s/step - loss: 0.3461 - accuracy:
0.8390 - val_loss: 0.4239 - val_accuracy: 0.8040
Epoch 76/100
20/20 [==============================] - 21s 1s/step - loss: 0.3659 - accuracy:
0.8285 - val_loss: 0.4355 - val_accuracy: 0.7960
Epoch 77/100
20/20 [==============================] - 20s 1s/step - loss: 0.3363 - accuracy:
0.8560 - val_loss: 0.3990 - val_accuracy: 0.8330
Epoch 78/100
20/20 [==============================] - 21s 1s/step - loss: 0.3594 - accuracy:
0.8360 - val_loss: 0.3984 - val_accuracy: 0.8260
Epoch 79/100
20/20 [==============================] - 20s 1s/step - loss: 0.3474 - accuracy:
0.8500 - val_loss: 0.4035 - val_accuracy: 0.8150
Epoch 80/100
20/20 [==============================] - 20s 992ms/step - loss: 0.3463 -
accuracy: 0.8485 - val_loss: 0.4542 - val_accuracy: 0.7950
Epoch 81/100
20/20 [==============================] - 20s 1s/step - loss: 0.3457 - accuracy:
0.8500 - val_loss: 0.4473 - val_accuracy: 0.7950
Epoch 82/100
20/20 [==============================] - 20s 1s/step - loss: 0.3480 - accuracy:
0.8485 - val_loss: 0.3952 - val_accuracy: 0.8200
Epoch 83/100
20/20 [==============================] - 20s 1s/step - loss: 0.3322 - accuracy:
```

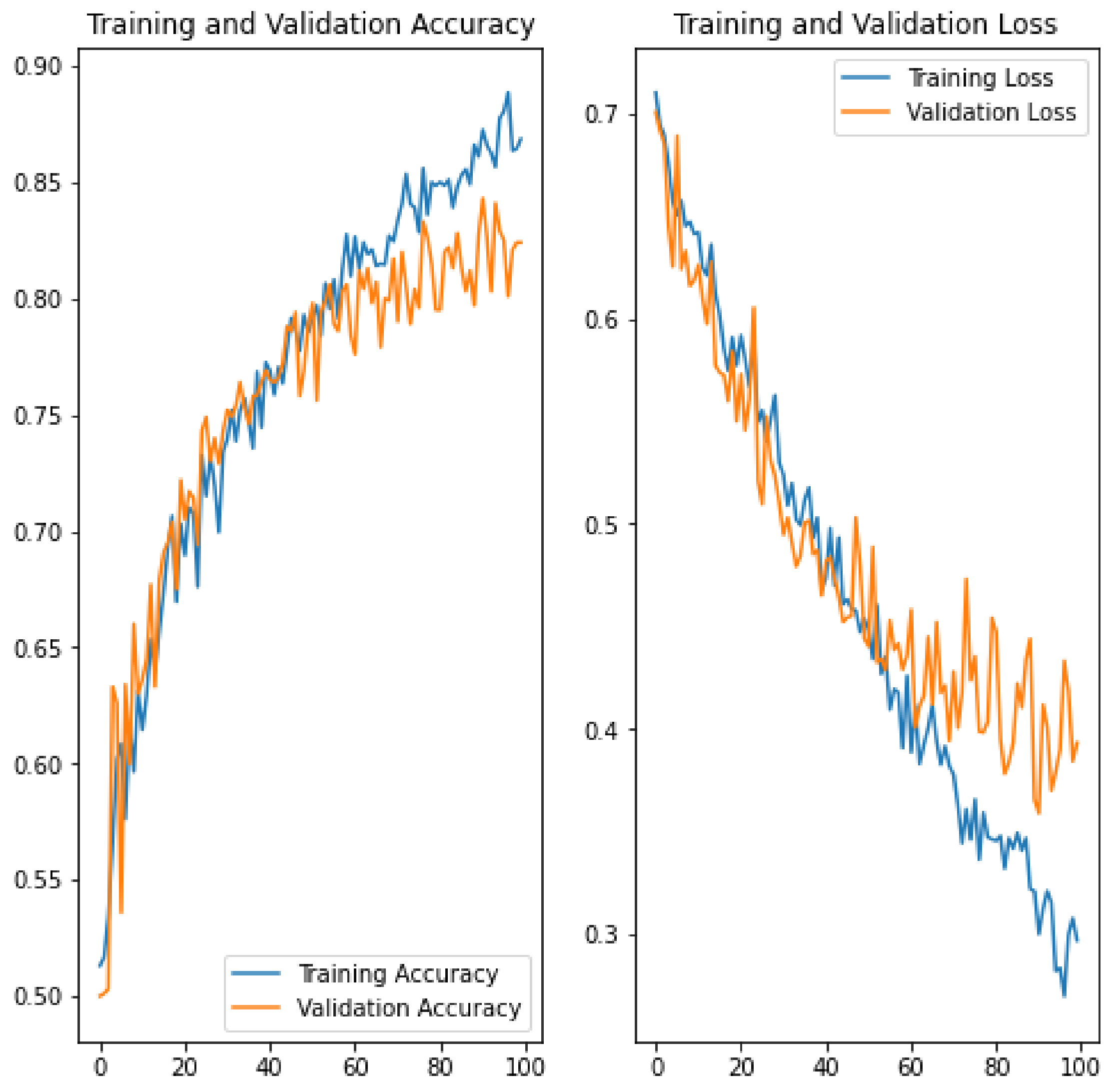
```
0.8510 - val_loss: 0.3783 - val_accuracy: 0.8220
Epoch 84/100
20/20 [==============================] - 20s 1s/step - loss: 0.3468 - accuracy:
0.8390 - val_loss: 0.3839 - val_accuracy: 0.8130
Epoch 85/100
20/20 [==============================] - 20s 1s/step - loss: 0.3419 - accuracy:
0.8480 - val_loss: 0.3934 - val_accuracy: 0.8280
Epoch 86/100
20/20 [==============================] - 20s 1s/step - loss: 0.3494 - accuracy:
0.8530 - val_loss: 0.4222 - val_accuracy: 0.8130
Epoch 87/100
20/20 [==============================] - 20s 1s/step - loss: 0.3408 - accuracy:
0.8555 - val_loss: 0.4105 - val_accuracy: 0.8030
Epoch 88/100
20/20 [==============================] - 20s 1s/step - loss: 0.3470 - accuracy:
0.8490 - val_loss: 0.4339 - val_accuracy: 0.8120
Epoch 89/100
20/20 [==============================] - 20s 992ms/step - loss: 0.3222 -
accuracy: 0.8660 - val_loss: 0.4440 - val_accuracy: 0.7970
Epoch 90/100
20/20 [==============================] - 20s 985ms/step - loss: 0.3210 -
accuracy: 0.8610 - val_loss: 0.3657 - val_accuracy: 0.8270
Epoch 91/100
20/20 [==============================] - 20s 998ms/step - loss: 0.3001 -
accuracy: 0.8725 - val_loss: 0.3590 - val_accuracy: 0.8430
Epoch 92/100
20/20 [==============================] - 20s 1s/step - loss: 0.3125 - accuracy:
0.8660 - val_loss: 0.4119 - val_accuracy: 0.8260
Epoch 93/100
20/20 [==============================] - 20s 1s/step - loss: 0.3212 - accuracy:
0.8625 - val_loss: 0.4010 - val_accuracy: 0.8030
Epoch 94/100
20/20 [==============================] - 20s 1s/step - loss: 0.3157 - accuracy:
0.8565 - val_loss: 0.3699 - val_accuracy: 0.8410
Epoch 95/100
20/20 [==============================] - 20s 1s/step - loss: 0.2822 - accuracy:
0.8775 - val_loss: 0.3794 - val_accuracy: 0.8290
Epoch 96/100
20/20 [==============================] - 20s 1s/step - loss: 0.2838 - accuracy:
0.8805 - val_loss: 0.3901 - val_accuracy: 0.8250
Epoch 97/100
20/20 [==============================] - 20s 1s/step - loss: 0.2700 - accuracy:
0.8885 - val_loss: 0.4332 - val_accuracy: 0.8010
Epoch 98/100
20/20 [==============================] - 20s 1s/step - loss: 0.2999 - accuracy:
0.8635 - val_loss: 0.4188 - val_accuracy: 0.8210
Epoch 99/100
20/20 [==============================] - 20s 1s/step - loss: 0.3079 - accuracy:
```

```
0.8645 - val_loss: 0.3843 - val_accuracy: 0.8240
Epoch 100/100
20/20 [==============================] - 20s 1s/step - loss: 0.2971 - accuracy:
0.8685 - val_loss: 0.3933 - val_accuracy: 0.8240
```

6.1.4 Visualizing results of the training

We'll now visualize the results we get after training our network.

[25]:
```
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()
```


Training and Validation Accuracy
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0
20
40
60
80
100
Training Accuracy
Validation Accuracy
Training and Validation Loss
0.7
0.6
0.5
0.4
0.3
Training Loss
Validation Loss