Open in Colab

Ungraded Lab: Data Augmentation

In the previous lessons, you saw that having a high training accuracy does not automatically mean having a good predictive model. It can still perform poorly on new data because it has overfit to the training set. In this lab, you will see how to avoid that using *data augmentation*. This increases the amount of training data by modifying the existing training data's properties. For example, in image data, you can apply different preprocessing techniques such as rotate, flip, shear, or zoom on your existing images so you can simulate other data that the model should also learn from. This way, the model would see more variety in the images during training so it will infer better on new, previously unseen data.

Let's see how you can do this in the following sections.

Baseline Performance

You will start with a model that's very effective at learning `Cats vs Dogs` without data augmentation. It's similar to the previous models that you have used. Note that there are four convolutional layers with 32, 64, 128 and 128 convolutions respectively. The code is basically the same from the previous lab so we won't go over the details step by step since you've already seen it before.

You will train only for 20 epochs to save time but feel free to increase this if you want.

```
# Download the dataset
!wget https://storage.googleapis.com/tensorflow-1-public/course2/cats_and_dogs_filtered.zip
```

```
--2023-07-04 10:01:17--  https://storage.googleapis.com/tensorflow-1-public/course2/cats_and_dogs_filtered.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 142.251.10.128, 142.251.12.128, 172.217.194.128, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|142.251.10.128|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 68606236 (65M) [application/zip]
Saving to: ‘cats_and_dogs_filtered.zip’

cats_and_dogs_filte 100%[===================>]  65.43M  17.5MB/s    in 4.8s

2023-07-04 10:01:23 (13.7 MB/s) - ‘cats_and_dogs_filtered.zip’ saved [68606236/68606236]
```

```
import os
import zipfile

# Extract the archive
zip_ref = zipfile.ZipFile("./cats_and_dogs_filtered.zip", 'r')
zip_ref.extractall("tmp/")
zip_ref.close()


# Assign training and validation set directories
base_dir = 'tmp/cats_and_dogs_filtered'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')

# Directory with training cat pictures
train_cats_dir = os.path.join(train_dir, 'cats')

# Directory with training dog pictures
train_dogs_dir = os.path.join(train_dir, 'dogs')

# Directory with validation cat pictures
validation_cats_dir = os.path.join(validation_dir, 'cats')

# Directory with validation dog pictures
validation_dogs_dir = os.path.join(validation_dir, 'dogs')
```

You will place the model creation inside a function so you can easily initialize a new one when you use data augmentation later in this notebook.

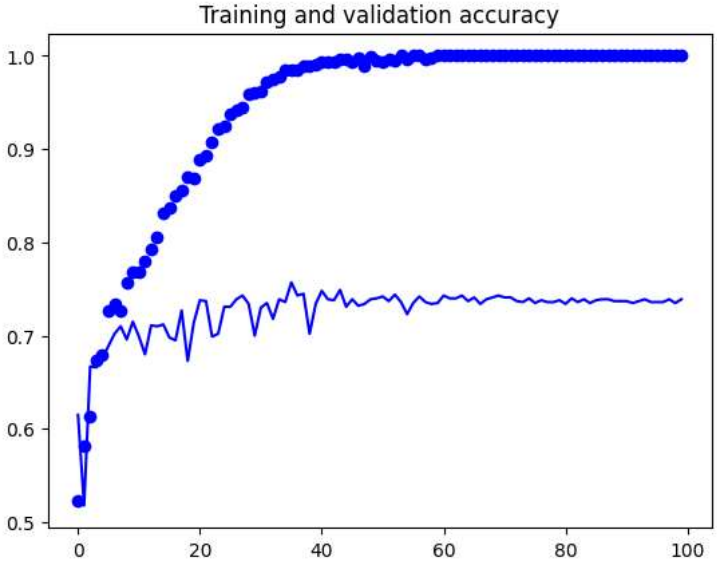
```
import tensorflow as tf
from tensorflow.keras.optimizers import RMSprop
```

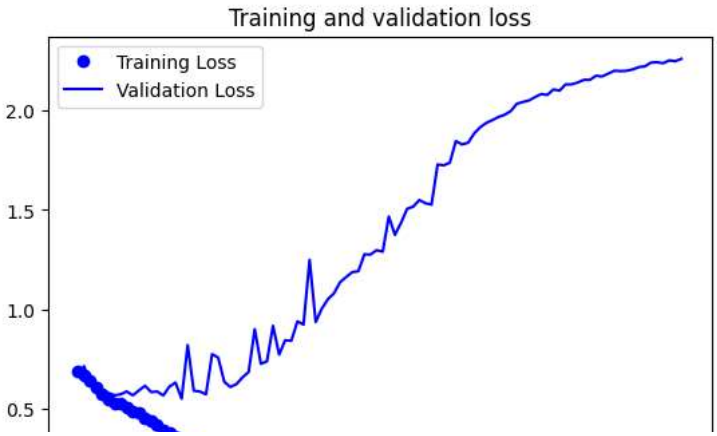
```
def create_model():
  '''Creates a CNN with 4 convolutional layers'''
  model = tf.keras.models.Sequential([
      tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(150, 150, 3)),
      tf.keras.layers.MaxPooling2D(2, 2),
      tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
      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.Flatten(),
      tf.keras.layers.Dense(512, activation='relu'),
      tf.keras.layers.Dense(1, activation='sigmoid')
  ])

  model.compile(loss='binary_crossentropy',
                optimizer=RMSprop(learning_rate=1e-4),
                metrics=['accuracy'])

  return model


from tensorflow.keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

# Flow training images in batches of 20 using train_datagen generator
train_generator = train_datagen.flow_from_directory(
        train_dir,  # This is the source directory for training images
        target_size=(150, 150),  # All images will be resized to 150x150
        batch_size=20,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

# Flow validation images in batches of 20 using test_datagen generator
validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')

    Found 2000 images belonging to 2 classes.
    Found 1000 images belonging to 2 classes.


# Constant for epochs
EPOCHS = 100

# Create a new model
model = create_model()

# Train the model
history = model.fit(
      train_generator,
      steps_per_epoch=100,  # 2000 images = batch_size * steps
      epochs=EPOCHS,
      validation_data=validation_generator,
      validation_steps=50,  # 1000 images = batch_size * steps
      verbose=2)
```

```
Epoch 80/100
100/100 - 9s - loss: 1.4927e-05 - accuracy: 1.0000 - val_loss: 2.0996 - val_accuracy: 0.7380 - 9s/epoch - 92ms/step
Epoch 81/100
100/100 - 9s - loss: 1.3932e-05 - accuracy: 1.0000 - val_loss: 2.1308 - val_accuracy: 0.7340 - 9s/epoch - 94ms/step
Epoch 82/100
100/100 - 10s - loss: 1.3414e-05 - accuracy: 1.0000 - val_loss: 2.1310 - val_accuracy: 0.7400 - 10s/epoch - 104ms/step
Epoch 83/100
100/100 - 10s - loss: 1.2979e-05 - accuracy: 1.0000 - val_loss: 2.1407 - val_accuracy: 0.7360 - 10s/epoch - 101ms/step
Epoch 84/100
100/100 - 10s - loss: 1.2422e-05 - accuracy: 1.0000 - val_loss: 2.1535 - val_accuracy: 0.7390 - 10s/epoch - 102ms/step
Epoch 85/100
100/100 - 11s - loss: 1.1653e-05 - accuracy: 1.0000 - val_loss: 2.1538 - val_accuracy: 0.7350 - 11s/epoch - 113ms/step
Epoch 86/100
100/100 - 9s - loss: 1.1241e-05 - accuracy: 1.0000 - val_loss: 2.1742 - val_accuracy: 0.7380 - 9s/epoch - 90ms/step
Epoch 87/100
100/100 - 9s - loss: 1.0577e-05 - accuracy: 1.0000 - val_loss: 2.1709 - val_accuracy: 0.7390 - 9s/epoch - 93ms/step
Epoch 88/100
100/100 - 10s - loss: 1.0193e-05 - accuracy: 1.0000 - val_loss: 2.1849 - val_accuracy: 0.7390 - 10s/epoch - 102ms/step
Epoch 89/100
100/100 - 10s - loss: 9.8248e-06 - accuracy: 1.0000 - val_loss: 2.1995 - val_accuracy: 0.7370 - 10s/epoch - 104ms/step
Epoch 90/100
100/100 - 10s - loss: 9.7127e-06 - accuracy: 1.0000 - val_loss: 2.1976 - val_accuracy: 0.7370 - 10s/epoch - 105ms/step
Epoch 91/100
100/100 - 11s - loss: 9.2394e-06 - accuracy: 1.0000 - val_loss: 2.1989 - val_accuracy: 0.7370 - 11s/epoch - 114ms/step
Epoch 92/100
100/100 - 11s - loss: 8.8391e-06 - accuracy: 1.0000 - val_loss: 2.2060 - val_accuracy: 0.7350 - 11s/epoch - 114ms/step
Epoch 93/100
100/100 - 9s - loss: 8.5630e-06 - accuracy: 1.0000 - val_loss: 2.2176 - val_accuracy: 0.7370 - 9s/epoch - 92ms/step
Epoch 94/100
100/100 - 10s - loss: 8.3089e-06 - accuracy: 1.0000 - val_loss: 2.2213 - val_accuracy: 0.7390 - 10s/epoch - 98ms/step
Epoch 95/100
100/100 - 10s - loss: 8.0410e-06 - accuracy: 1.0000 - val_loss: 2.2403 - val_accuracy: 0.7360 - 10s/epoch - 104ms/step
Epoch 96/100
100/100 - 10s - loss: 7.8332e-06 - accuracy: 1.0000 - val_loss: 2.2421 - val_accuracy: 0.7360 - 10s/epoch - 101ms/step
Epoch 97/100
100/100 - 10s - loss: 7.5681e-06 - accuracy: 1.0000 - val_loss: 2.2372 - val_accuracy: 0.7360 - 10s/epoch - 103ms/step
Epoch 98/100
100/100 - 9s - loss: 7.4241e-06 - accuracy: 1.0000 - val_loss: 2.2515 - val_accuracy: 0.7390 - 9s/epoch - 93ms/step
Epoch 99/100
100/100 - 10s - loss: 7.1218e-06 - accuracy: 1.0000 - val_loss: 2.2472 - val_accuracy: 0.7350 - 10s/epoch - 98ms/step
Epoch 100/100
100/100 - 10s - loss: 6.9822e-06 - accuracy: 1.0000 - val_loss: 2.2591 - val_accuracy: 0.7390 - 10s/epoch - 103ms/step
```

You will then visualize the loss and accuracy with respect to the training and validation set. You will again use a convenience function so it can be reused later. This function accepts a History object which contains the results of the `fit()` method you ran above.

```
import matplotlib.pyplot as plt

def plot_loss_acc(history):
  '''Plots the training and validation loss and accuracy from a history object'''
  acc = history.history['accuracy']
  val_acc = history.history['val_accuracy']
  loss = history.history['loss']
  val_loss = history.history['val_loss']

  epochs = range(len(acc))

  plt.plot(epochs, acc, 'bo', label='Training accuracy')
  plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
  plt.title('Training and validation accuracy')

  plt.figure()

  plt.plot(epochs, loss, 'bo', label='Training Loss')
  plt.plot(epochs, val_loss, 'b', label='Validation Loss')
  plt.title('Training and validation loss')
  plt.legend()

  plt.show()


# Plot training results
plot_loss_acc(history)
```





From the results above, you'll see the training accuracy is more than 90%, and the validation accuracy is in the 70%-80% range. This is a great example of *overfitting* -- which in short means that it can do very well with images it has seen before, but not so well with images it hasn't.

Data augmentation

One simple method to avoid overfitting is to augment the images a bit. If you think about it, most pictures of a cat are very similar -- the ears are at the top, then the eyes, then the mouth etc. Things like the distance between the eyes and ears will always be quite similar too.

What if you tweak with the images a bit -- rotate the image, squash it, etc. That's what image augementation is all about. And there's an API that makes it easy!

Take a look at the ImageDataGenerator which you have been using to rescale the image. There are other properties on it that you can use to augment the image.

```
# Updated to do image augmentation
train_datagen = ImageDataGenerator(
      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')
```

These are just a few of the options available. Let's quickly go over it:

- `rotation_range` is a value in degrees (0–180) within which to randomly rotate pictures.
- `width_shift` and `height_shift` are ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally.
- `shear_range` is for randomly applying shearing transformations.

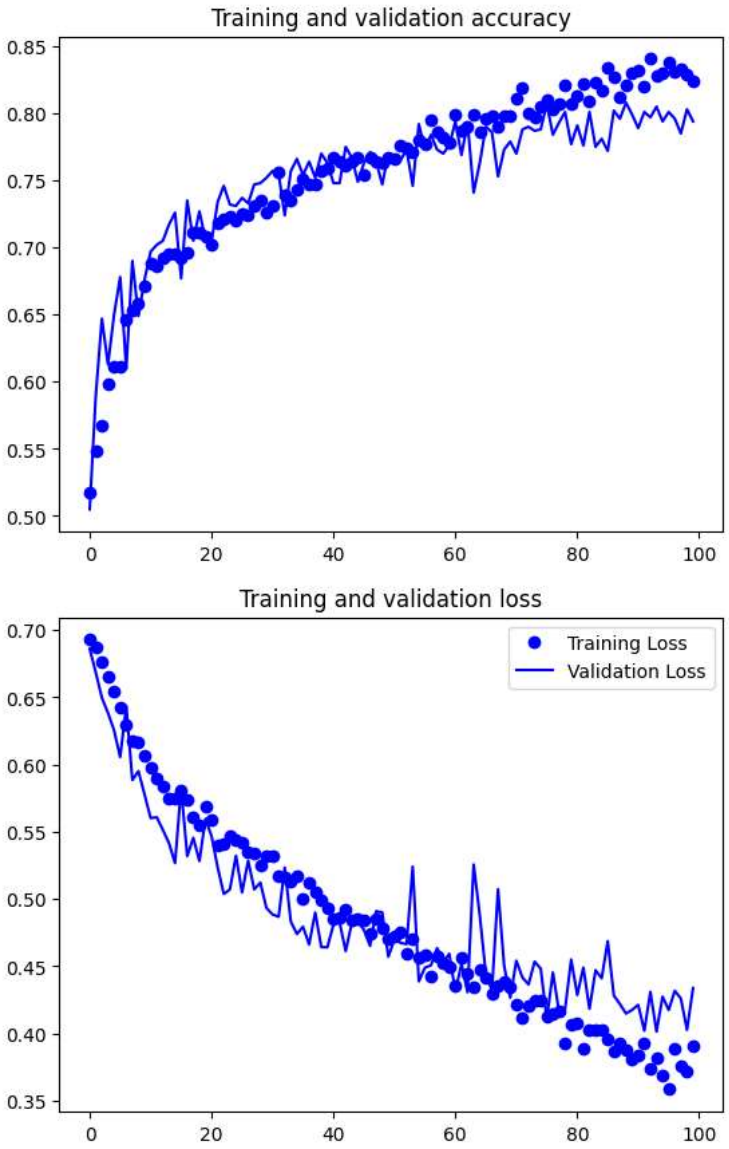
- `zoom_range` is for randomly zooming inside pictures.
- `horizontal_flip` is for randomly flipping half of the images horizontally. This is relevant when there are no assumptions of horizontal assymmetry (e.g. real-world pictures).
- `fill_mode` is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift.

Run the next cells to see the impact on the results. The code is similar to the baseline but the definition of `train_datagen` has been updated to use the parameters described above.

```
# Create new model
model_for_aug = create_model()

# This code has changed. Now instead of the ImageGenerator just rescaling
# the image, we also rotate and do other operations
train_datagen = 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')

test_datagen = ImageDataGenerator(rescale=1./255)

# Flow training images in batches of 20 using train_datagen generator
train_generator = train_datagen.flow_from_directory(
        train_dir,  # This is the source directory for training images
        target_size=(150, 150),  # All images will be resized to 150x150
        batch_size=20,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

# Flow validation images in batches of 20 using test_datagen generator
validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')

# Train the new model
history_with_aug = model_for_aug.fit(
      train_generator,
      steps_per_epoch=100,  # 2000 images = batch_size * steps
      epochs=EPOCHS,
      validation_data=validation_generator,
      validation_steps=50,  # 1000 images = batch_size * steps
      verbose=2)
```

```
Epoch 86/100
100/100 - 20s - loss: 0.3954 - accuracy: 0.8340 - val_loss: 0.4686 - val_accuracy: 0.7720 - 20s/epoch - 201ms/step
Epoch 87/100
100/100 - 21s - loss: 0.3867 - accuracy: 0.8270 - val_loss: 0.4281 - val_accuracy: 0.8020 - 21s/epoch - 210ms/step
Epoch 88/100
100/100 - 23s - loss: 0.3923 - accuracy: 0.8125 - val_loss: 0.4218 - val_accuracy: 0.7960 - 23s/epoch - 225ms/step
Epoch 89/100
100/100 - 20s - loss: 0.3875 - accuracy: 0.8215 - val_loss: 0.4147 - val_accuracy: 0.8080 - 20s/epoch - 203ms/step
Epoch 90/100
100/100 - 19s - loss: 0.3805 - accuracy: 0.8295 - val_loss: 0.4175 - val_accuracy: 0.7990 - 19s/epoch - 195ms/step
Epoch 91/100
100/100 - 21s - loss: 0.3834 - accuracy: 0.8315 - val_loss: 0.4212 - val_accuracy: 0.7890 - 21s/epoch - 206ms/step
Epoch 92/100
100/100 - 19s - loss: 0.3931 - accuracy: 0.8200 - val_loss: 0.4020 - val_accuracy: 0.8010 - 19s/epoch - 188ms/step
Epoch 93/100
100/100 - 20s - loss: 0.3732 - accuracy: 0.8405 - val_loss: 0.4309 - val_accuracy: 0.7970 - 20s/epoch - 200ms/step
Epoch 94/100
100/100 - 21s - loss: 0.3819 - accuracy: 0.8280 - val_loss: 0.4014 - val_accuracy: 0.8050 - 21s/epoch - 210ms/step
Epoch 95/100
100/100 - 20s - loss: 0.3689 - accuracy: 0.8300 - val_loss: 0.4273 - val_accuracy: 0.7940 - 20s/epoch - 200ms/step
Epoch 96/100
100/100 - 21s - loss: 0.3583 - accuracy: 0.8375 - val_loss: 0.4174 - val_accuracy: 0.8010 - 21s/epoch - 211ms/step
Epoch 97/100
100/100 - 20s - loss: 0.3884 - accuracy: 0.8310 - val_loss: 0.4317 - val_accuracy: 0.7960 - 20s/epoch - 204ms/step
Epoch 98/100
100/100 - 21s - loss: 0.3756 - accuracy: 0.8325 - val_loss: 0.4259 - val_accuracy: 0.7850 - 21s/epoch - 213ms/step
Epoch 99/100
100/100 - 19s - loss: 0.3713 - accuracy: 0.8285 - val_loss: 0.4026 - val_accuracy: 0.8030 - 19s/epoch - 194ms/step
Epoch 100/100
100/100 - 20s - loss: 0.3907 - accuracy: 0.8245 - val_loss: 0.4337 - val_accuracy: 0.7940 - 20s/epoch - 201ms/step
```

```
# Plot the results of training with data augmentation
plot_loss_acc(history_with_aug)
```



As you can see, the training accuracy has gone down compared to the baseline. This is expected because (as a result of data augmentation) there are more variety in the images so the model will need more runs to learn from them. The good thing is the validation accuracy is no longer stalling and is more in line with the training results. This means that the model is now performing better on unseen data.

Wrap Up

This exercise showed a simple trick to avoid overfitting. You can improve your baseline results by simply tweaking the same images you have already. The `ImageDataGenerator` class has built-in parameters to do just that. Try to modify the values some more in the `train_datagen` and see what results you get.

Take note that this will not work for all cases. In the next lesson, Laurence will show a scenario where data augmentation will not help improve your validation accuracy.