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`.

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.

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.

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

```
import tensorflow as tf
```

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

```
import os
import numpy as np
import matplotlib.pyplot as plt
```

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.

```
_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, extrac
```

```
Downloading data from https://storage.google
68608000/68606236 [==============================
```

The dataset we have downloaded has following directory structure.

```
cats_and_dogs_filtered
|__ train
    |______ cats: [cat.0.jpg, cat.1.jpg, cat.2.
    |______ dogs: [dog.0.jpg, dog.1.jpg, dog.2.
|__ validation
    |______ cats: [cat.2000.jpg, cat.2001.jpg,
    |______ dogs: [dog.2000.jpg, dog.2001.jpg,
```

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

```
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')
```

```
train_cats_dir = os.path.join(train_dir, 'cats')  # directory with our training cat
train_dogs_dir = os.path.join(train_dir, 'dogs')  # directory with our training dog
validation_cats_dir = os.path.join(validation_dir, 'cats')  # directory with our val
validation_dogs_dir = os.path.join(validation_dir, 'dogs')  # directory with our val
```

Understanding our data

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

```
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


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
```

Setting Model Parameters

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

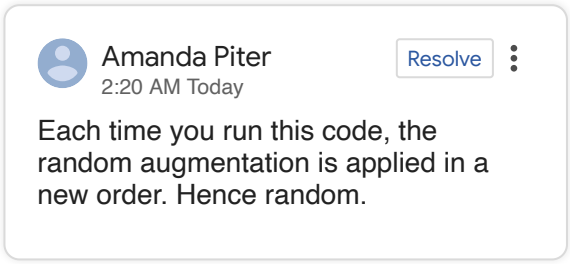
```
BATCH_SIZE = 100
IMG_SHAPE  = 150 # Our training data consists of images with width of 150 pixels and
```

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.

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.

```
# This function will plot images in the form of a grid with 1 row and 5 columns whe
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()
```

Flipping the image horizontally

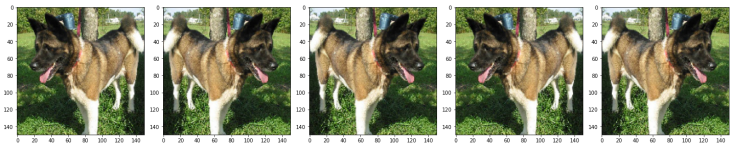
We can begin by randomly applying horizontal flip augmentation to our dataset and seeing how individual images will look after the transformation. This is achieved by passing `horizontal_flip=True` as an argument to the `ImageDataGenerator` class.

```
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.

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



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.

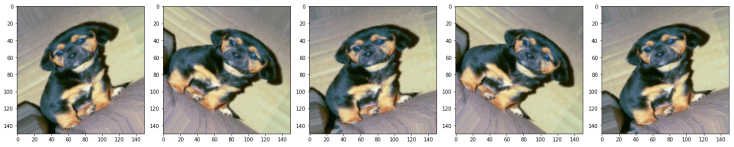
```
image_gen = ImageDataGenerator(rescale=1./255, rotation_range=45)

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.

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



Applying Zoom

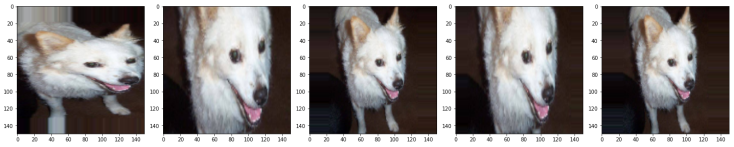
We can also apply Zoom augmentation to our dataset, zooming images up to 50% randomly.

```
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.

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



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.

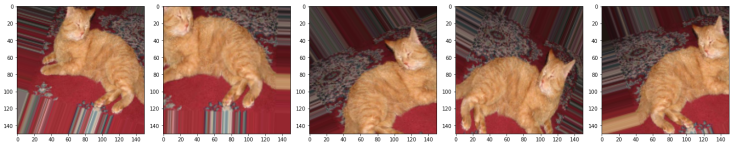
Here, we have applied rescale, rotation of 45 degrees, width shift, height shift, horizontal flip, and zoom augmentation to our training images.

```
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_SHAI
                                                     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.

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



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.

```
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.
```

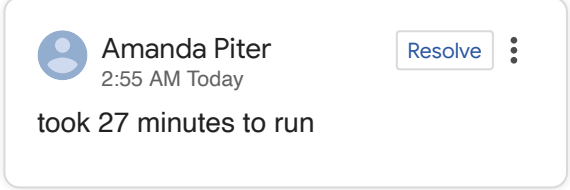
Model Creation

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`.




```
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)
])
```

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.

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

Model Summary

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

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              
=================================================================
conv2d (Conv2D)              (None, 148, 148
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 74, 74,
_________________________________________________________________
```

```
conv2d_1 (Conv2D)            (None, 72, 72,
_________________________________________
max_pooling2d_1 (MaxPooling2 (None, 36, 36,
_________________________________________
conv2d_2 (Conv2D)            (None, 34, 34,
_________________________________________
max_pooling2d_2 (MaxPooling2 (None, 17, 17,
_________________________________________
conv2d_3 (Conv2D)            (None, 15, 15,
_________________________________________
max_pooling2d_3 (MaxPooling2 (None, 7, 7, 12
_________________________________________
dropout (Dropout)            (None, 7, 7, 12
_________________________________________
flatten (Flatten)            (None, 6272)
_________________________________________
dense (Dense)                (None, 512)
_________________________________________
dense_1 (Dense)              (None, 2)
=================================================
Total params: 3,453,634
Trainable params: 3,453,634
Non-trainable params: 0
_________________________________________
```

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` .

```
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)))
)
```

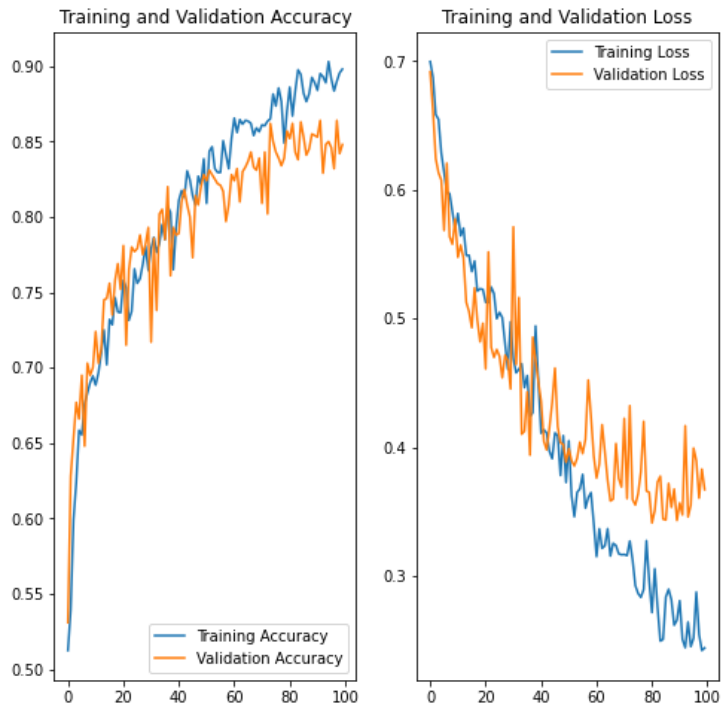
```
20/20 [==============================] - 16s
Epoch 63/100
20/20 [==============================] - 16s
Epoch 64/100
20/20 [==============================] - 16s
Epoch 65/100
20/20 [==============================] - 16s
Epoch 66/100
20/20 [==============================] - 16s
Epoch 67/100
```

```
20/20 [==============================] - 16s
Epoch 68/100
20/20 [==============================] - 16s
Epoch 69/100
20/20 [==============================] - 16s
Epoch 70/100
20/20 [==============================] - 16s
Epoch 71/100
20/20 [==============================] - 16s
Epoch 72/100
20/20 [==============================] - 16s
Epoch 73/100
20/20 [==============================] - 16s
Epoch 74/100
20/20 [==============================] - 16s
Epoch 75/100
20/20 [==============================] - 16s
Epoch 76/100
20/20 [==============================] - 16s
Epoch 77/100
20/20 [==============================] - 16s
Epoch 78/100
20/20 [==============================] - 16s
Epoch 79/100
20/20 [==============================] - 16s
Epoch 80/100
20/20 [==============================] - 16s
Epoch 81/100
20/20 [==============================] - 16s
Epoch 82/100
20/20 [==============================] - 16s
Epoch 83/100
20/20 [==============================] - 16s
Epoch 84/100
20/20 [==============================] - 16s
Epoch 85/100
20/20 [==============================] - 16s
Epoch 86/100
20/20 [==============================] - 16s
Epoch 87/100
20/20 [==============================] - 16s
Epoch 88/100
20/20 [==============================] - 16s
Epoch 89/100
20/20 [==============================] - 16s

Epoch 90/100
20/20 [==============================] - 16s
Epoch 91/100
20/20 [==============================] - 16s
```

Visualizing results of the training

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

```
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()
```



✓ 0s completed at 2:55 AM