Building a convolution neural network to classify cat and dog images with Tensorflow and Transfer learning

In this post, we would be building a machine learning-based binary image classification model for classifying images of cats and dogs with Tensorflow and Keras.

Libraries

TensorFlow is a free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks.

Keras is compact, easy to learn, high-level Python library run on top of TensorFlow framework. It is made with focus of understanding deep learning techniques, such as creating layers for neural networks maintaining the concepts of shapes and mathematical details.

```
import os
from keras import utils
```

```
import tensorflow as tf
```

It might take a while to train each model if you are running on your computer's local ram. By using google colab, you can change your runtime to be based on GPU, or even built your on VMs with the google cloud computing engine and connect your colab notebook to it.

```

print(tf.test.gpu_device_name())
```

```
    /device:GPU:0
```

The first step would be loading the datasets. For building a typical machine learning model, we usually need to split the data into train, test, and validation subsets. Train and test are used to optimize the model performance in the process of training, while validation set is used to evaluate the model's performance on unforseen data.

```

# location of data
_URL = 'https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip'

# download the data and extract it
path_to_zip = utils.get_file('cats_and_dogs.zip', origin=_URL, extract=True)

# construct paths
PATH = os.path.join(os.path.dirname(path_to_zip), 'cats_and_dogs_filtered')

train_dir = os.path.join(PATH, 'train')
validation_dir = os.path.join(PATH, 'validation')

# parameters for datasets
BATCH_SIZE = 32
IMG_SIZE = (160, 160)

# construct train and validation datasets
train_dataset = utils.image_dataset_from_directory(train_dir,
                                                   shuffle=True,
                                                   batch_size=BATCH_SIZE,
                                                   image_size=IMG_SIZE)

validation_dataset = utils.image_dataset_from_directory(validation_dir,
                                                        shuffle=True,
                                                        batch_size=BATCH_SIZE,
                                                        image_size=IMG_SIZE)

# construct the test dataset by taking every 5th observation out of the validation dataset
val_batches = tf.data.experimental.cardinality(validation_dataset)
```

```
test_dataset = validation_dataset.take(val_batches // 5)
validation_dataset = validation_dataset.skip(val_batches // 5)
```

```
    Downloading data from https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip
    68606236/68606236 [==============================] - 4s 0us/step
    Found 2000 files belonging to 2 classes.
    Found 1000 files belonging to 2 classes.
```

By running the above section code, we have created TensorFlow Datasets for training, validation, and testing.

We also made use of a special-purpose keras utility called image_dataset_from_directory to construct a Dataset. The arguments for the function are:

- directory: where the images are located.
- shuffle: True/False, when retrieving data from this directory, the order would be randomized if True.
- batch_size: how many data points are gathered from the directory at once
- image_size: specifies the size of the input images

Additionally, the following code would allow you to read-in data quickly when training through prefetching.

Prefetch would make use of the spared CPU power and I/O bandwidth to pre-load the next dataset/batch when the current batch is being processed to reduce latency.

```
AUTOTUNE = tf.data.AUTOTUNE

train_dataset = train_dataset.prefetch(buffer_size=AUTOTUNE)
validation_dataset = validation_dataset.prefetch(buffer_size=AUTOTUNE)
test_dataset = test_dataset.prefetch(buffer_size=AUTOTUNE)
```

Dataset visualization

```
from matplotlib import pyplot as plt
```

```
t_dataset = utils.image_dataset_from_directory(train_dir,
                                               shuffle=True,
                                               batch_size=BATCH_SIZE,
                                               image_size=IMG_SIZE)
t_dataset.class_names
```

```
    Found 2000 files belonging to 2 classes.
    ['cats', 'dogs']
```

In this sect., we would write a function to create a two-row visualization, where the first row contains random images of cat/dog, and the second row contains random images of the other pet.

```
def two_row(batch):
  class_names = ['cats', 'dogs']
  fig, ax = plt.subplots(2, 3, figsize=(10, 10))
  for images, labels in batch:
    cnt_cat = 0
    cnt_dog = 0
    i = 0
    while cnt_cat < 3 or cnt_dog < 3:
      if labels[i] == 1 and cnt_cat < 3:
        ax[0,cnt_cat].imshow(images[i].numpy().astype("uint8"))
        ax[0,cnt_cat].set_title(class_names[labels[i]])
        plt.axis("off")
        cnt_cat += 1
      elif labels[i] == 0 and cnt_dog < 3:
        ax[1,cnt_dog].imshow(images[i].numpy().astype("uint8"))
        ax[1,cnt_dog].set_title(class_names[labels[i]])
        plt.axis("off")
        cnt_dog += 1
      i += 1
```

```
two_row(train_dataset.take(1))
```

Check frequencies

For a baseline model, all inputs would be classified as the most frequent label.

```
labels_iterator= train_dataset.unbatch().map(lambda image, label: label).as_numpy_iterator()
```

```
WARNING:tensorflow:From /usr/local/lib/python3.8/dist-packages/tensorflow/python/autograph/pyct/static_analysis/liveness.py:
Instructions for updating:
Lambda fuctions will be no more assumed to be used in the statement where they are used, or at least in the same block. http
```

```
dogs = 0
cnt = 0
for lbl in labels_iterator:
  dogs += lbl
  cnt += 1
print("dog: ",dogs/cnt)
print("cat: ",(cnt - dogs) / cnt)
```

```
dog:  0.5
cat:  0.5
```

In this case, all inputs would be guessed as either dog or cat, meaning a 50% accuracy of classification for the baseline model.

First model: Sequential model

Create a tf.keras.Sequential model using some of the layers we've discussed in class. In each model, include at least two Conv2D layers, at least two MaxPooling2D layers, at least one Flatten layer, at least one Dense layer, and at least one Dropout layer. Train your model and plot the history of the accuracy on both the training and validation sets. Give your models different names

```
import keras
from keras import layers
```

```
#loading one batch from the dataset
train_dataset.take(1)
```

```
<TakeDataset element_spec=(TensorSpec(shape=(None, 160, 160, 3), dtype=tf.float32, name=None), TensorSpec(shape=(None,),
dtype=tf.int32, name=None))>
```

```
#one batch has a total of 32 images,
#the output is either dog/cat for each image, binary classification
#from above, the shape of each input/img is 160,160,3 (height, width, rgb)
```

```
model1 = keras.Sequential([
    keras.Input(shape=(160,160,3)),
    layers.Conv2D(filters = 32, kernel_size = (5,5), strides=(2,2), activation="relu"),
    layers.Dropout(.1),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(32, (3,3), 1, activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(),
    layers.Dense(32, activation='relu'),
    layers.Dense(16, activation='relu')
])

model1.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d (Conv2D)             (None, 78, 78, 32)        2432      
                                                                 
 dropout (Dropout)           (None, 78, 78, 32)        0         
                                                                 
 max_pooling2d (MaxPooling2D  (None, 39, 39, 32)       0         
 )                                                               
                                                                 
 conv2d_1 (Conv2D)           (None, 37, 37, 32)        9248      
                                                                 
 max_pooling2d_1 (MaxPooling  (None, 18, 18, 32)       0         
 2D)                                                             
                                                                 
 flatten (Flatten)           (None, 10368)             0         
                                                                 
 dense (Dense)               (None, 32)                331808    
                                                                 
 dense_1 (Dense)             (None, 16)                528       
                                                                 
=================================================================
Total params: 344,016
Trainable params: 344,016
Non-trainable params: 0
_________________________________________________________________
```

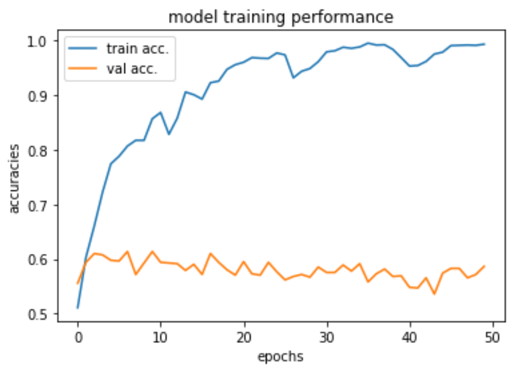
```
optimizer = 'adam'
#sparse_categorical_crossentropy: Used as a loss function
#for multi-class classification model where the output label is assigned integer value (0, 1, 2, 3…).
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
metrics = ['accuracy']
model1.compile(optimizer, loss, metrics)
```

```
def plot_from_history(history):
    """
    Plot model training accuracy and validation accuracy over training epochs from model.fit history.
    """
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('model training performance')
    plt.ylabel('accuracies')
    plt.xlabel('epochs')
    plt.legend(['train acc.', 'val acc.'], loc='upper left')
    plt.show()
```

```
history1 = model1.fit(train_dataset,
                     epochs=50,
                     validation_data=validation_dataset)
```

```
Epoch 29/50
63/63 [==============================] - 3s 45ms/step - loss: 0.1638 - accuracy: 0.9490 - val_loss: 2.8287 - val_accuracy: 0
Epoch 30/50
63/63 [==============================] - 3s 46ms/step - loss: 0.1095 - accuracy: 0.9615 - val_loss: 2.8042 - val_accuracy: 0
Epoch 31/50
63/63 [==============================] - 4s 61ms/step - loss: 0.0559 - accuracy: 0.9795 - val_loss: 3.2268 - val_accuracy: 0
Epoch 32/50
63/63 [==============================] - 3s 46ms/step - loss: 0.0576 - accuracy: 0.9815 - val_loss: 3.0948 - val_accuracy: 0
Epoch 33/50
63/63 [==============================] - 3s 47ms/step - loss: 0.0394 - accuracy: 0.9880 - val_loss: 3.5414 - val_accuracy: 0
Epoch 34/50
63/63 [==============================] - 3s 46ms/step - loss: 0.0518 - accuracy: 0.9860 - val_loss: 3.7371 - val_accuracy: 0
Epoch 35/50
63/63 [==============================] - 4s 63ms/step - loss: 0.0301 - accuracy: 0.9885 - val_loss: 3.6986 - val_accuracy: 0
Epoch 36/50
63/63 [==============================] - 3s 45ms/step - loss: 0.0205 - accuracy: 0.9955 - val_loss: 3.5964 - val_accuracy: 0
Epoch 37/50
63/63 [==============================] - 3s 45ms/step - loss: 0.0242 - accuracy: 0.9920 - val_loss: 3.9564 - val_accuracy: 0
Epoch 38/50
63/63 [==============================] - 3s 50ms/step - loss: 0.0234 - accuracy: 0.9925 - val_loss: 4.1115 - val_accuracy: 0
Epoch 39/50
63/63 [==============================] - 4s 54ms/step - loss: 0.0515 - accuracy: 0.9840 - val_loss: 4.8419 - val_accuracy: 0
Epoch 40/50
63/63 [==============================] - 3s 46ms/step - loss: 0.1226 - accuracy: 0.9690 - val_loss: 3.9227 - val_accuracy: 0
Epoch 41/50
63/63 [==============================] - 3s 45ms/step - loss: 0.1452 - accuracy: 0.9535 - val_loss: 3.4994 - val_accuracy: 0
Epoch 42/50
63/63 [==============================] - 4s 62ms/step - loss: 0.1612 - accuracy: 0.9545 - val_loss: 4.0409 - val_accuracy: 0
Epoch 43/50
63/63 [==============================] - 3s 46ms/step - loss: 0.1259 - accuracy: 0.9620 - val_loss: 3.4836 - val_accuracy: 0
Epoch 44/50
63/63 [==============================] - 3s 45ms/step - loss: 0.0983 - accuracy: 0.9755 - val_loss: 3.7912 - val_accuracy: 0
Epoch 45/50
63/63 [==============================] - 4s 62ms/step - loss: 0.0632 - accuracy: 0.9790 - val_loss: 3.5603 - val_accuracy: 0
Epoch 46/50
63/63 [==============================] - 3s 45ms/step - loss: 0.0329 - accuracy: 0.9910 - val_loss: 4.0083 - val_accuracy: 0
Epoch 47/50
63/63 [==============================] - 3s 46ms/step - loss: 0.0238 - accuracy: 0.9915 - val_loss: 3.8261 - val_accuracy: 0
Epoch 48/50
63/63 [==============================] - 4s 62ms/step - loss: 0.0284 - accuracy: 0.9920 - val_loss: 4.5754 - val_accuracy: 0
Epoch 49/50
63/63 [==============================] - 3s 45ms/step - loss: 0.0395 - accuracy: 0.9915 - val_loss: 5.1191 - val_accuracy: 0
Epoch 50/50
63/63 [==============================] - 3s 46ms/step - loss: 0.0172 - accuracy: 0.9935 - val_loss: 4.9158 - val_accuracy: 0
```

```
plot_from_history(history1)
```



The validation accuracy of my model stabilized **somewhere between 61% and 63%** during training, more than **10% improvement comparing to the baseline.** However, the training accuracy is as high as 98.90%, an evidence for **overfitting**.

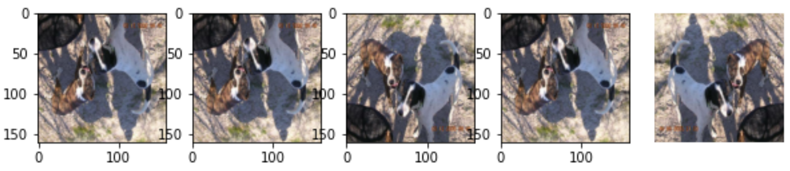
Second model: With data augmentation

Data augmentation refers to the practice of including modified copies of the same image in the training set. For example, a picture of a cat is still a picture of a cat even if we flip it upside down or rotate it 90 degrees. We can include such transformed versions of the image in our training process in order to help our model learn so-called invariant features of our input images.

Random Flip

```
#input: 3D (unbatched) or 4D (batched) tensor with shape:
  #(..., height, width, channels), in "channels_last" format.
#output: same
r_flip = tf.keras.layers.RandomFlip(
    mode="horizontal_and_vertical", seed=2)
```

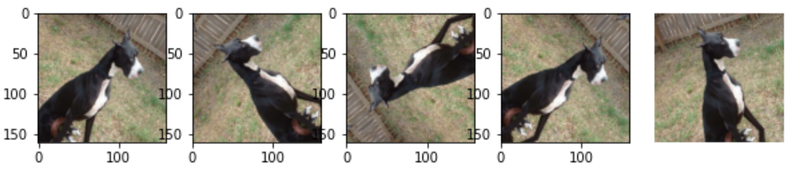
```
batch = train_dataset.take(1)
for img,labl in batch:
  pic = img[0]
fig, ax = plt.subplots(1, 5, figsize=(10, 10))
for i in range(5):
    ax[i].imshow(r_flip(pic).numpy().astype("uint8"))
    #ax[0,i].set_title()
    plt.axis("off")
```



Random Rotate

```
## rotate
r_rotate = tf.keras.layers.RandomRotation(
    #factor (rotation angle range, unit in percent of 2 pi)
    factor = (-1,1),
    fill_mode="reflect",
    interpolation="bilinear",
    seed=2,
    fill_value=0.0
)
```

```
batch = train_dataset.take(1)
for img,labl in batch:
  pic = img[0]
fig, ax = plt.subplots(1, 5, figsize=(10, 10))
for i in range(5):
    ax[i].imshow(r_rotate(pic).numpy().astype("uint8"))
    #ax[0,i].set_title()
    plt.axis("off")
```



```
#adding preprocessing: random flip and random rotate
model2 = keras.Sequential([
    keras.Input(shape=(160,160,3)),
    r_flip,
    r_rotate,
    layers.Conv2D(filters = 32, kernel_size = (5,5), strides=(2,2), activation="relu"),
    layers.Dropout(.1),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(32, (3,3), 1, activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(),
    layers.Dense(32, activation='relu'),
    layers.Dense(16, activation='relu')
])
model2.build()

model2.summary()
```

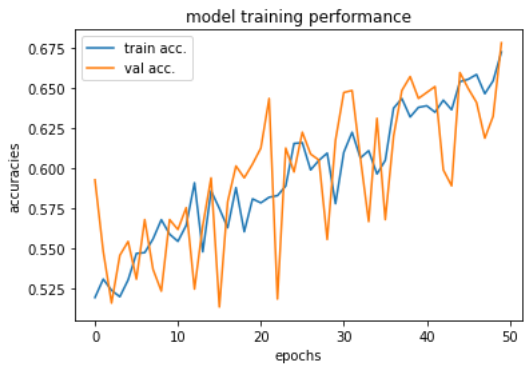
```
WARNING:tensorflow:Using a while_loop for converting RngReadAndSkip cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting StatelessRandomUniformV2 cause there is no registered converter for thi
WARNING:tensorflow:Using a while_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for t
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 random_flip (RandomFlip)    (None, 160, 160, 3)       0         
                                                                 
 random_rotation (RandomRota  (None, 160, 160, 3)      0         
 tion)                                                           
                                                                 
 conv2d_2 (Conv2D)           (None, 78, 78, 32)        2432      
                                                                 
 dropout_1 (Dropout)         (None, 78, 78, 32)        0         
                                                                 
 max_pooling2d_2 (MaxPooling  (None, 39, 39, 32)       0         
 2D)                                                             
                                                                 
 conv2d_3 (Conv2D)           (None, 37, 37, 32)        9248      
                                                                 
 max_pooling2d_3 (MaxPooling  (None, 18, 18, 32)       0         
 2D)                                                             
                                                                 
 flatten_1 (Flatten)         (None, 10368)             0         
                                                                 
 dense_2 (Dense)             (None, 32)                331808    
                                                                 
 dense_3 (Dense)             (None, 16)                528       
                                                                 
=================================================================
Total params: 344,016
Trainable params: 344,016
Non-trainable params: 0
_________________________________________________________________
```

```
model2.compile(optimizer, loss, metrics)
```

```
history2 = model2.fit(train_dataset,
                     epochs=50,
                     validation_data = validation_dataset)
```

```
63/63 [==============================] - 8s 122ms/step - loss: 0.6575 - accuracy: 0.6100 - val_loss: 0.6424 - val_accuracy:
Epoch 32/50
63/63 [==============================] - 8s 122ms/step - loss: 0.6468 - accuracy: 0.6225 - val_loss: 0.6470 - val_accuracy:
Epoch 33/50
63/63 [==============================] - 7s 110ms/step - loss: 0.6653 - accuracy: 0.6065 - val_loss: 0.6698 - val_accuracy:
Epoch 34/50
63/63 [==============================] - 7s 107ms/step - loss: 0.6505 - accuracy: 0.6110 - val_loss: 0.6805 - val_accuracy:
Epoch 35/50
63/63 [==============================] - 8s 123ms/step - loss: 0.6658 - accuracy: 0.5965 - val_loss: 0.6444 - val_accuracy:
Epoch 36/50
63/63 [==============================] - 8s 122ms/step - loss: 0.6527 - accuracy: 0.6050 - val_loss: 0.6802 - val_accuracy:
Epoch 37/50
63/63 [==============================] - 7s 109ms/step - loss: 0.6332 - accuracy: 0.6375 - val_loss: 0.6491 - val_accuracy:
Epoch 38/50
63/63 [==============================] - 8s 124ms/step - loss: 0.6329 - accuracy: 0.6435 - val_loss: 0.6414 - val_accuracy:
Epoch 39/50
63/63 [==============================] - 8s 123ms/step - loss: 0.6315 - accuracy: 0.6320 - val_loss: 0.6312 - val_accuracy:
Epoch 40/50
63/63 [==============================] - 7s 108ms/step - loss: 0.6422 - accuracy: 0.6380 - val_loss: 0.6330 - val_accuracy:
Epoch 41/50
63/63 [==============================] - 8s 123ms/step - loss: 0.6307 - accuracy: 0.6390 - val_loss: 0.6328 - val_accuracy:
Epoch 42/50
63/63 [==============================] - 9s 135ms/step - loss: 0.6471 - accuracy: 0.6350 - val_loss: 0.6307 - val_accuracy:
Epoch 43/50
63/63 [==============================] - 7s 107ms/step - loss: 0.6258 - accuracy: 0.6425 - val_loss: 0.6882 - val_accuracy:
Epoch 44/50
63/63 [==============================] - 8s 121ms/step - loss: 0.6295 - accuracy: 0.6365 - val_loss: 0.6736 - val_accuracy:
Epoch 45/50
63/63 [==============================] - 8s 124ms/step - loss: 0.6247 - accuracy: 0.6540 - val_loss: 0.6194 - val_accuracy:
Epoch 46/50
63/63 [==============================] - 7s 110ms/step - loss: 0.6152 - accuracy: 0.6555 - val_loss: 0.6175 - val_accuracy:
Epoch 47/50
63/63 [==============================] - 8s 122ms/step - loss: 0.6218 - accuracy: 0.6585 - val_loss: 0.6079 - val_accuracy:
Epoch 48/50
63/63 [==============================] - 8s 123ms/step - loss: 0.6216 - accuracy: 0.6465 - val_loss: 0.6766 - val_accuracy:
Epoch 49/50
63/63 [==============================] - 7s 108ms/step - loss: 0.6147 - accuracy: 0.6545 - val_loss: 0.6336 - val_accuracy:
Epoch 50/50
63/63 [==============================] - 8s 121ms/step - loss: 0.6074 - accuracy: 0.6725 - val_loss: 0.6076 - val_accuracy:
```

```
plot_from_history(history2)
```



The model performance with data augmentation included is similar to the previous one with validation accuracy reaching somewhere **between 62% and 67%**, but **without significant overfitting** as the training accuracy is somehwere around 65% also.

Third model: adding more data preprocesing

Since the images are all colored, the original data has pixels with RGB values between 0 and 255. But many models will train faster with RGB values normalized between 0 and 1, or possibly between -1 and 1. These are mathematically identical situations, since we can always just scale the weights. But if we handle the scaling prior to the training process, we can spend more of our training energy handling actual signal in the data and less energy having the weights adjust to the data scale.

The following code will create a preprocessing layer called preprocessor which you can slot into your model pipeline to perform color normalization.

```
i = tf.keras.Input(shape=(160, 160, 3))
x = tf.keras.applications.mobilenet_v2.preprocess_input(i)
preprocessor = tf.keras.Model(inputs = [i], outputs = [x])
```

```
#adding preprocessing: random flip and random rotate
model3 = keras.Sequential([
    preprocessor,
    r_flip,
    r_rotate,
    layers.Conv2D(filters = 32, kernel_size = (5,5), strides=(2,2), activation="relu"),
    layers.Dropout(.1),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(32, (3,3), 1, activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(),
    layers.Dense(32, activation='relu'),
    layers.Dense(16, activation='relu')
])
model3.build()

model3.summary()
```

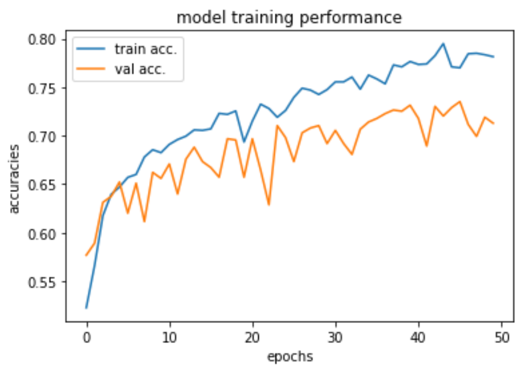
```
WARNING:tensorflow:Using a while_loop for converting RngReadAndSkip cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting StatelessRandomUniformV2 cause there is no registered converter for thi
WARNING:tensorflow:Using a while_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for t
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 model (Functional)          (None, 160, 160, 3)       0         
                                                                 
 random_flip (RandomFlip)    (None, 160, 160, 3)       0         
                                                                 
 random_rotation (RandomRota  (None, 160, 160, 3)      0         
 tion)                                                           
                                                                 
 conv2d_4 (Conv2D)           (None, 78, 78, 32)        2432      
                                                                 
 dropout_2 (Dropout)         (None, 78, 78, 32)        0         
                                                                 
 max_pooling2d_4 (MaxPooling  (None, 39, 39, 32)       0         
 2D)                                                             
                                                                 
 conv2d_5 (Conv2D)           (None, 37, 37, 32)        9248      
                                                                 
 max_pooling2d_5 (MaxPooling  (None, 18, 18, 32)       0         
 2D)                                                             
                                                                 
 flatten_2 (Flatten)         (None, 10368)             0         
                                                                 
 dense_4 (Dense)             (None, 32)                331808    
                                                                 
 dense_5 (Dense)             (None, 16)                528       
                                                                 
=================================================================
Total params: 344,016
Trainable params: 344,016
Non-trainable params: 0
_________________________________________________________________
```

```
model3.compile(optimizer, loss, metrics)
```

```
history3 = model3.fit(train_dataset,
                     epochs=50,
                     validation_data = validation_dataset)
```

```
63/63 [==============================] - 8s 114ms/step - loss: 0.5171 - accuracy: 0.7425 - val_loss: 0.5605 - val_accuracy:
Epoch 30/50
63/63 [==============================] - 8s 122ms/step - loss: 0.5145 - accuracy: 0.7475 - val_loss: 0.5807 - val_accuracy:
Epoch 31/50
63/63 [==============================] - 7s 107ms/step - loss: 0.5017 - accuracy: 0.7555 - val_loss: 0.5617 - val_accuracy:
Epoch 32/50
63/63 [==============================] - 7s 108ms/step - loss: 0.5025 - accuracy: 0.7555 - val_loss: 0.5856 - val_accuracy:
Epoch 33/50
63/63 [==============================] - 8s 123ms/step - loss: 0.4930 - accuracy: 0.7605 - val_loss: 0.5769 - val_accuracy:
Epoch 34/50
63/63 [==============================] - 7s 113ms/step - loss: 0.5087 - accuracy: 0.7480 - val_loss: 0.5584 - val_accuracy:
Epoch 35/50
63/63 [==============================] - 8s 111ms/step - loss: 0.4982 - accuracy: 0.7625 - val_loss: 0.5364 - val_accuracy:
Epoch 36/50
63/63 [==============================] - 8s 123ms/step - loss: 0.4930 - accuracy: 0.7585 - val_loss: 0.5403 - val_accuracy:
Epoch 37/50
63/63 [==============================] - 9s 135ms/step - loss: 0.4948 - accuracy: 0.7535 - val_loss: 0.5478 - val_accuracy:
Epoch 38/50
63/63 [==============================] - 7s 108ms/step - loss: 0.4818 - accuracy: 0.7730 - val_loss: 0.5491 - val_accuracy:
Epoch 39/50
63/63 [==============================] - 8s 122ms/step - loss: 0.4766 - accuracy: 0.7710 - val_loss: 0.5497 - val_accuracy:
Epoch 40/50
63/63 [==============================] - 8s 124ms/step - loss: 0.4724 - accuracy: 0.7765 - val_loss: 0.5513 - val_accuracy:
Epoch 41/50
63/63 [==============================] - 7s 107ms/step - loss: 0.4678 - accuracy: 0.7735 - val_loss: 0.5558 - val_accuracy:
Epoch 42/50
63/63 [==============================] - 8s 122ms/step - loss: 0.4793 - accuracy: 0.7740 - val_loss: 0.5943 - val_accuracy:
Epoch 43/50
63/63 [==============================] - 8s 121ms/step - loss: 0.4646 - accuracy: 0.7825 - val_loss: 0.5540 - val_accuracy:
Epoch 44/50
63/63 [==============================] - 7s 113ms/step - loss: 0.4586 - accuracy: 0.7950 - val_loss: 0.5650 - val_accuracy:
Epoch 45/50
63/63 [==============================] - 7s 106ms/step - loss: 0.4659 - accuracy: 0.7710 - val_loss: 0.5466 - val_accuracy:
Epoch 46/50
63/63 [==============================] - 8s 123ms/step - loss: 0.4700 - accuracy: 0.7700 - val_loss: 0.5695 - val_accuracy:
Epoch 47/50
63/63 [==============================] - 8s 123ms/step - loss: 0.4582 - accuracy: 0.7845 - val_loss: 0.5464 - val_accuracy:
Epoch 48/50
63/63 [==============================] - 7s 108ms/step - loss: 0.4618 - accuracy: 0.7850 - val_loss: 0.5854 - val_accuracy:
Epoch 49/50
63/63 [==============================] - 8s 123ms/step - loss: 0.4439 - accuracy: 0.7835 - val_loss: 0.5682 - val_accuracy:
Epoch 50/50
63/63 [==============================] - 8s 122ms/step - loss: 0.4616 - accuracy: 0.7815 - val_loss: 0.5494 - val_accuracy:
```

```
plot_from_history(history3)
```



Given the additional preprocessing layer that normalizes the RGB values between [-1,1] before the actual training, the validation accuracy stablized somewhere **between 70% and 71% without significant overfitting**.

Fourth Model: With transfer learning

In some cases, someone might already have trained a model that does a related task to your model, and might have learned some relevant patterns. Build incorporating the existing model as a layer in our current model and adding extra layers, we might get better classsification outcomes.

```
IMG_SHAPE = IMG_SIZE + (3,)
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
                                               include_top=False,
                                               weights='imagenet')
base_model.trainable = False
```

```
i = tf.keras.Input(shape=IMG_SHAPE)
x = base_model(i, training = False)
base_model_layer = tf.keras.Model(inputs = [i], outputs = [x])
```

```
    Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_weights_tf_dim_
    9406464/9406464 [==============================] - 0s 0us/step
```

```
#adding preprocessing: random flip and random rotate
model4 = keras.Sequential([
    preprocessor,
    r_flip,
    r_rotate,
    base_model_layer,
    layers.Dropout(.1),
    layers.GlobalMaxPooling2D(),
    layers.Flatten(),
    layers.Dense(32, activation='relu'),
    layers.Dense(2, activation='softmax')
])

model4.build()

model4.summary()
```

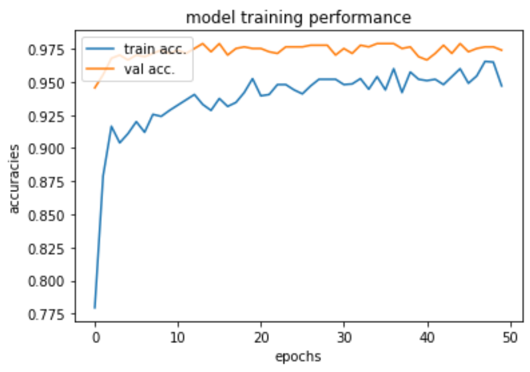
```
    WARNING:tensorflow:Using a while_loop for converting RngReadAndSkip cause there is no registered converter for this op.
    WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
    WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
    WARNING:tensorflow:Using a while_loop for converting StatelessRandomUniformV2 cause there is no registered converter for thi
    WARNING:tensorflow:Using a while_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for t
    Model: "sequential_3"
    _________________________________________________________________
     Layer (type)                Output Shape              Param #
    =================================================================
     model (Functional)          (None, 160, 160, 3)       0

     random_flip (RandomFlip)    (None, 160, 160, 3)       0

     random_rotation (RandomRota  (None, 160, 160, 3)      0
     tion)

     model_1 (Functional)        (None, 5, 5, 1280)        2257984

     dropout_3 (Dropout)         (None, 5, 5, 1280)        0

     global_max_pooling2d (Globa  (None, 1280)             0
     lMaxPooling2D)

     flatten_3 (Flatten)         (None, 1280)              0

     dense_6 (Dense)             (None, 32)                40992

     dense_7 (Dense)             (None, 2)                 66

    =================================================================
    Total params: 2,299,042
    Trainable params: 41,058
    Non-trainable params: 2,257,984
    _________________________________________________________________
```

```
model4.compile(optimizer,loss,metrics)
```

```
history4 = model4.fit(train_dataset,
                     epochs=50,
                     validation_data = validation_dataset)
```

```
63/63 [==============================] - 9s 135ms/step - loss: 0.1317 - accuracy: 0.9520 - val_loss: 0.0592 - val_accuracy:
Epoch 29/50
63/63 [==============================] - 8s 123ms/step - loss: 0.1203 - accuracy: 0.9520 - val_loss: 0.0625 - val_accuracy:
Epoch 30/50
63/63 [==============================] - 9s 139ms/step - loss: 0.1277 - accuracy: 0.9520 - val_loss: 0.0674 - val_accuracy:
Epoch 31/50
63/63 [==============================] - 9s 137ms/step - loss: 0.1293 - accuracy: 0.9480 - val_loss: 0.0573 - val_accuracy:
Epoch 32/50
63/63 [==============================] - 9s 146ms/step - loss: 0.1346 - accuracy: 0.9485 - val_loss: 0.0777 - val_accuracy:
Epoch 33/50
63/63 [==============================] - 8s 124ms/step - loss: 0.1174 - accuracy: 0.9525 - val_loss: 0.0675 - val_accuracy:
Epoch 34/50
63/63 [==============================] - 9s 140ms/step - loss: 0.1305 - accuracy: 0.9445 - val_loss: 0.0658 - val_accuracy:
Epoch 35/50
63/63 [==============================] - 9s 137ms/step - loss: 0.1206 - accuracy: 0.9540 - val_loss: 0.0589 - val_accuracy:
Epoch 36/50
63/63 [==============================] - 8s 123ms/step - loss: 0.1312 - accuracy: 0.9440 - val_loss: 0.0636 - val_accuracy:
Epoch 37/50
63/63 [==============================] - 8s 126ms/step - loss: 0.1067 - accuracy: 0.9600 - val_loss: 0.0603 - val_accuracy:
Epoch 38/50
63/63 [==============================] - 9s 139ms/step - loss: 0.1220 - accuracy: 0.9420 - val_loss: 0.0651 - val_accuracy:
Epoch 39/50
63/63 [==============================] - 9s 143ms/step - loss: 0.1076 - accuracy: 0.9575 - val_loss: 0.0606 - val_accuracy:
Epoch 40/50
63/63 [==============================] - 8s 124ms/step - loss: 0.1120 - accuracy: 0.9520 - val_loss: 0.0827 - val_accuracy:
Epoch 41/50
63/63 [==============================] - 8s 127ms/step - loss: 0.1239 - accuracy: 0.9510 - val_loss: 0.0746 - val_accuracy:
Epoch 42/50
63/63 [==============================] - 9s 138ms/step - loss: 0.1212 - accuracy: 0.9520 - val_loss: 0.0785 - val_accuracy:
Epoch 43/50
63/63 [==============================] - 9s 138ms/step - loss: 0.1337 - accuracy: 0.9480 - val_loss: 0.0631 - val_accuracy:
Epoch 44/50
63/63 [==============================] - 9s 139ms/step - loss: 0.1220 - accuracy: 0.9540 - val_loss: 0.0732 - val_accuracy:
Epoch 45/50
63/63 [==============================] - 8s 124ms/step - loss: 0.1093 - accuracy: 0.9600 - val_loss: 0.0672 - val_accuracy:
Epoch 46/50
63/63 [==============================] - 9s 137ms/step - loss: 0.1203 - accuracy: 0.9490 - val_loss: 0.0737 - val_accuracy:
Epoch 47/50
63/63 [==============================] - 9s 138ms/step - loss: 0.1144 - accuracy: 0.9545 - val_loss: 0.0626 - val_accuracy:
Epoch 48/50
63/63 [==============================] - 9s 136ms/step - loss: 0.0941 - accuracy: 0.9655 - val_loss: 0.0665 - val_accuracy:
Epoch 49/50
63/63 [==============================] - 8s 122ms/step - loss: 0.0993 - accuracy: 0.9650 - val_loss: 0.0639 - val_accuracy:
Epoch 50/50
63/63 [==============================] - 8s 126ms/step - loss: 0.1292 - accuracy: 0.9470 - val_loss: 0.0665 - val_accuracy:
```

```
plot_from_history(history4)
```



By incorporating a pretrained network and transfer the learning to our model, the final validation accuracy we reached is approximately **97.8%** with no evidence of overfitting as the train accuracy is somewhere around 96.7%.

Test on test set

Our best performing model is the last model. The following sections evaluate the model performance on the test dataset.

```
import numpy as np
```

https://keras.io/api/models/model_training_apis/#predictonbatch-method

```
test_image = []
test_label = []
for image, label in test_dataset:
  test_image.append(image.numpy())
  test_label.append(label.numpy())
```

```
test_imge = np.concatenate(test_image)
```

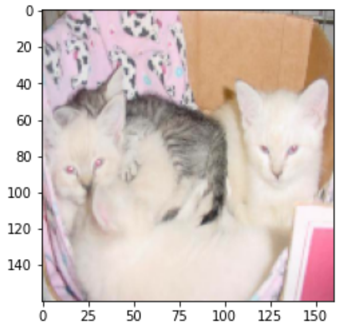
```
test_label = np.concatenate(test_label)
```

```
test_label[0] #dog is 1, cat is 0
```

```
0
```

```
plt.imshow(test_imge[0].astype("uint8"))
```

```
<matplotlib.image.AxesImage at 0x7f821bb02d60>
```



```
prediction = model4.predict(test_imge)
prediction.shape
```

```
6/6 [==============================] - 1s 21ms/step
(192, 2)
```

```
pred = np.argmax(prediction, axis=-1)
pred.shape == test_label.shape
```

```
True
```

```
print("Test accuracy: ",np.sum([pred == test_label])/len(pred))
```

```
Test accuracy:  0.9895833333333334
```

We reached a final classification accuracy of **98.96%** on the test dataset by using the fourth model which incorporated color normalization, random flip/rotate, and learning transfer.