BST 261: Data Science II
Lecture 6
Convolutional Neural Networks (CNNs): Data Augmentation and Pretrained networks

Heather Mattie
Harvard T.H. Chan School of Public Health
Spring 2 2021

Recipe of the Day!

[Rosemary crackers]

Pretrained Networks

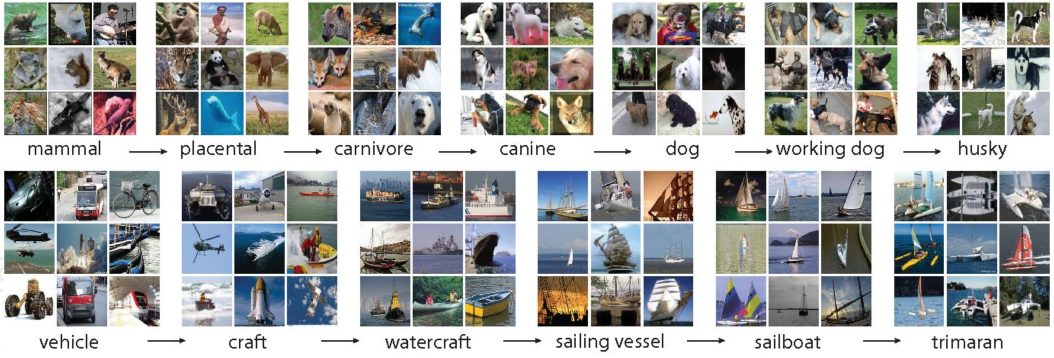
Pretrained Networks

- Another way around having a small number of training examples to learn from is using networks that have been trained on other, bigger datasets similar to the type of data you have
- A **pretrained network** is a saved network that was previously trained on a large dataset
- If the dataset used to train the network is large enough and big enough, the features learned by the pretrained network can act as a generic model to use as a base for your network
- This saves an enormous amount of computing time
- Pretrained networks can be used for **feature extraction** and **fine-tuning**

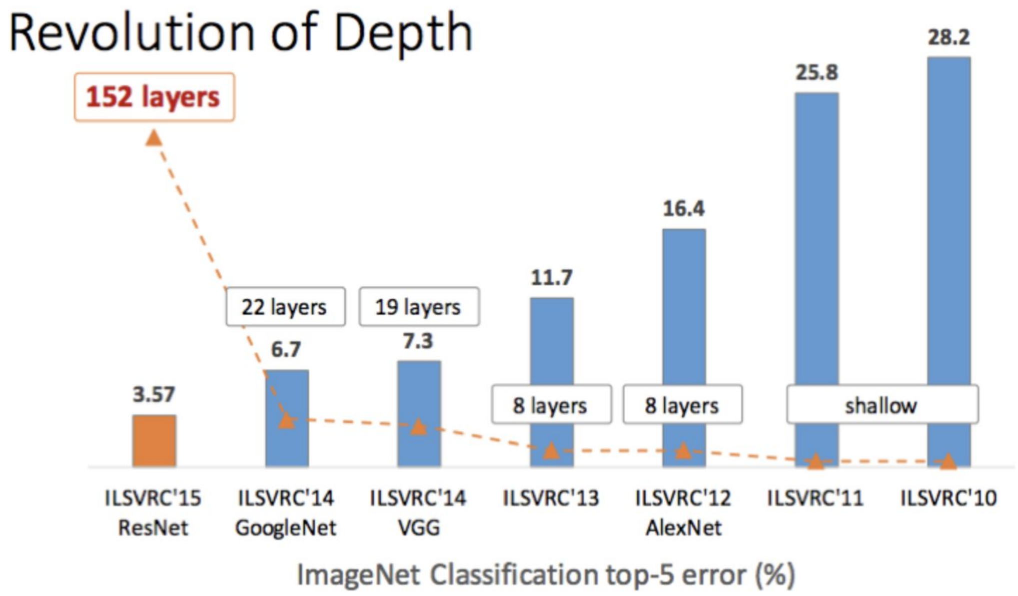
Pretrained Networks

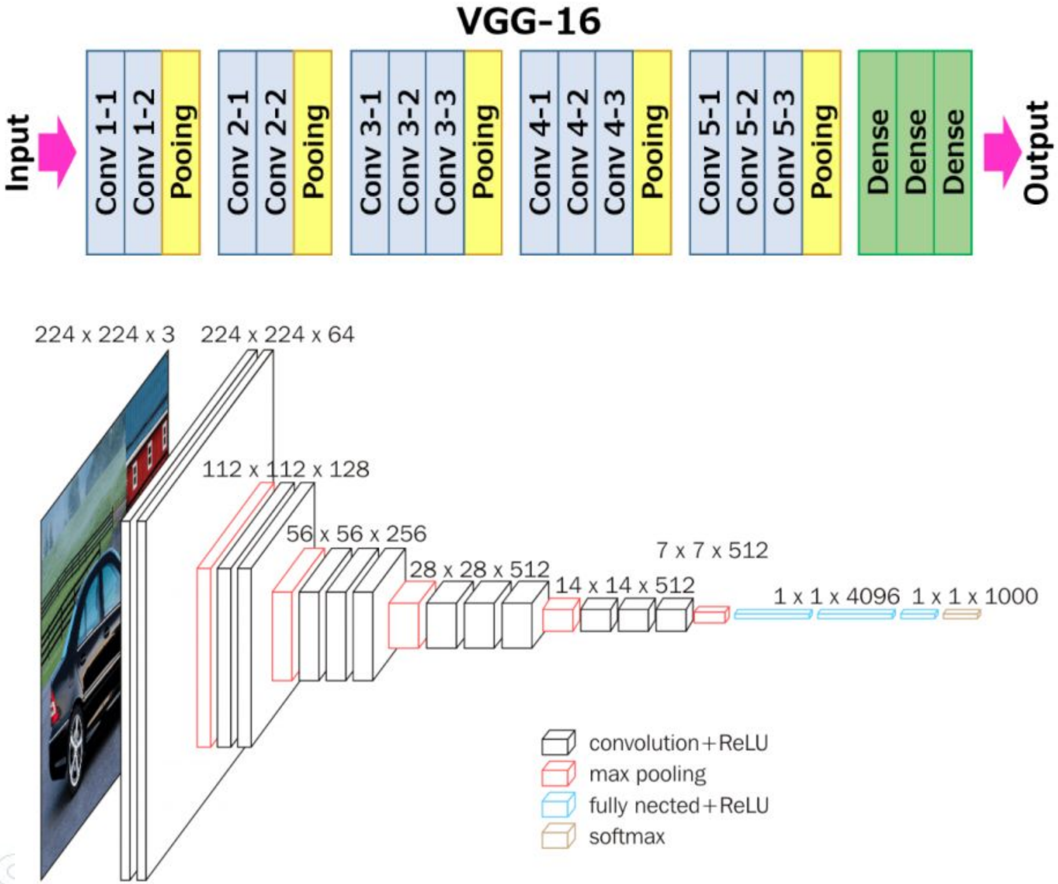
- Commonly used pretrained networks include
  - VGG16
  - ResNet
  - Inception
  - Inception-ResNet
  - Xception

- Commonly used dataset used to train a network is the [ImageNet dataset]()
  - 1.4 million labeled images
  - 1,000 different classes
  - Mostly animals and everyday objects



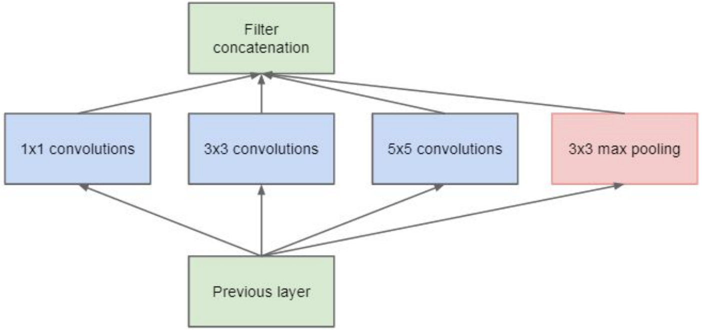
Pretrained Networks



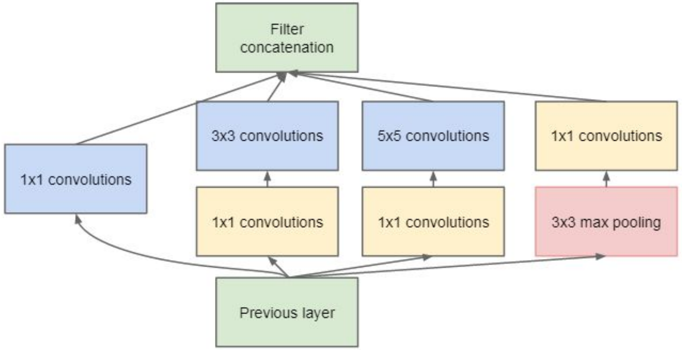


https://neurohive.io/en/popular-networks/vgg16/

Inception Models



(a) Inception module, naïve version

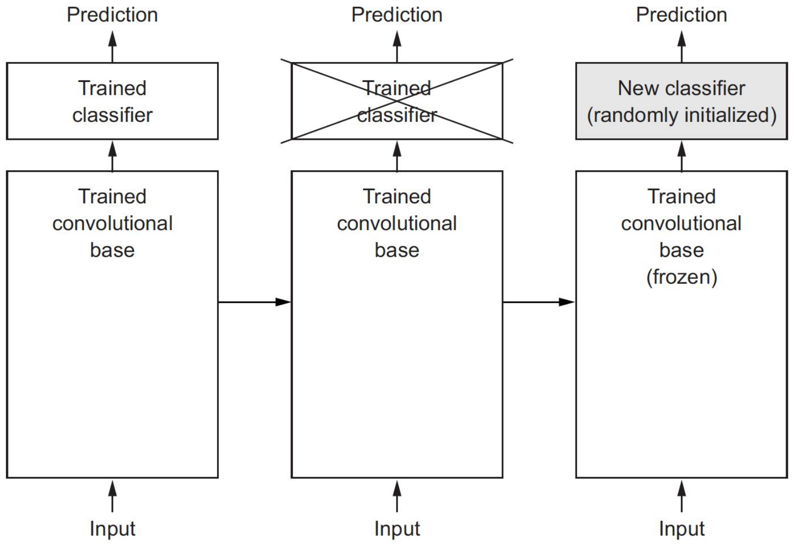


(b) Inception module with dimension reductions

https://becominghuman.ai/understanding-and-coding-inception-module-in-keras-eb56e9056b4b

Feature Extraction

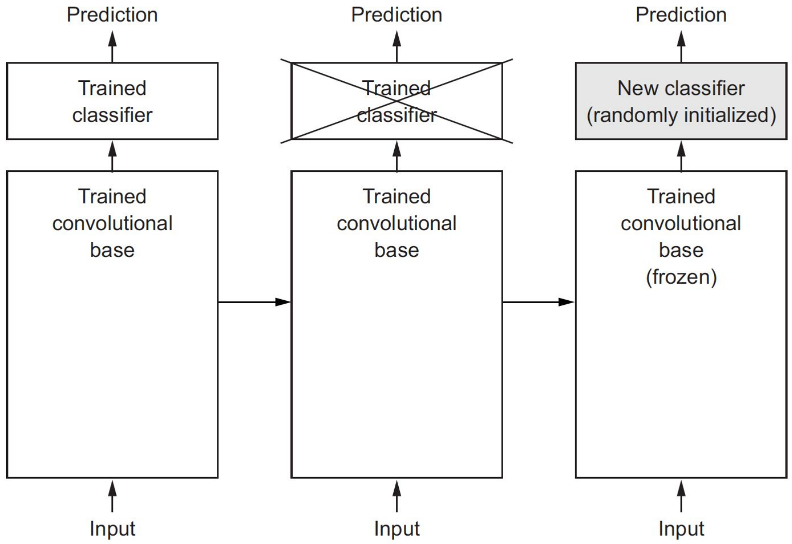
- Consists of using the representations learned by a previous network to extract features from new samples
- These features are then run through a new classifier that is trained from scratch, and predictions are made



Feature Extraction

- For CNNs, the part of the pretrained network you use is called the **convolutional base**, which contains a series of convolution and pooling layers

- For feature extraction, you keep the convolutional base of the pretrained network, remove the dense / trained classifier layers, and append new dense and classifier layers to the convolutional base



Feature Extraction

- We could also reuse the densely connected classifier as well, but this is not advised
- Representations learned by the convolutional base are likely to be more generic and thus more reusable
- The representations learned by the classifier will be specific to the set of classes the model was trained on
- They will also no longer contain information about where objects are located in the input image
  - This makes them especially useless when the object's location is important

Feature Extraction

- The level of generality depends on the depth of the layer in the model
  - Early layers extract local, highly generic features, i.e. edges, colors, textures
  - Later layers extract more abstract concepts i.e. “cat ear” or “dog eye”

- If your new dataset is very different from the dataset that was used to train the model, you should use only the first few layers for feature extraction rather than the entire base

Pretrained Networks in Keras

- Xception
- Inception V3
- ResNet50
- VGG16
- VGG19
- MobileNet

Instantiating the VGG16 Base

```python
#from keras.applications import VGG16

conv_base = tf.keras.applications.VGG16(weights='imagenet',
                                        include_top=False,
                                        input_shape=(150, 150, 3))
```

```python
conv_base.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 150, 150, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 150, 150, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 150, 150, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 75, 75, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 75, 75, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 75, 75, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 37, 37, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 37, 37, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 37, 37, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 37, 37, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 18, 18, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 18, 18, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 18, 18, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 18, 18, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 9, 9, 512)         0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 4, 4, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

Instantiating the VGG16 Base

```python
#from keras.applications import VGG16

conv_base = tf.keras.applications.VGG16(weights='imagenet',
                                        include_top=False,
                                        input_shape=(150, 150, 3))
```

The final layer is a pooling layer and the final output shape from this base is (4, 4, 512). We need this information when adding layers to the base. This output shape will be the input shape for the densely connected layer we'll add to the base.

```
conv_base.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 150, 150, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 150, 150, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 150, 150, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 75, 75, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 75, 75, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 75, 75, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 37, 37, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 37, 37, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 37, 37, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 37, 37, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 18, 18, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 18, 18, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 18, 18, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 18, 18, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 9, 9, 512)         0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 4, 4, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

Using a Pretrained Network

- The final output has shape (4, 4, 512)
- You have 2 options:
  1. **Feature extraction without augmented data**: you can run the convolutional base over the dataset, record its output to a numpy array, and then use these values as input to a densely connected classifier
     - This is fast and cheap to run
     - It won't allow you to use augmented data
  2. **Feature extraction with augmented data**: you can extend the convolutional base by adding dense layers on top and running the whole model on the input data
     - This allows data augmentation
     - This is very computationally expensive

Feature Extraction <u>without</u> Augmented Data

```python
import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(rescale=1./255)
batch_size = 20

def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 512))
    labels = np.zeros(shape=(sample_count))
    generator = datagen.flow_from_directory(
        directory,
        target_size=(150, 150),
        batch_size=batch_size,
        class_mode='binary')
    i = 0
    for inputs_batch, labels_batch in generator:
        features_batch = conv_base.predict(inputs_batch)
        features[i * batch_size : (i + 1) * batch_size] = features_batch
        labels[i * batch_size : (i + 1) * batch_size] = labels_batch
        i += 1
        if i * batch_size >= sample_count:
            # Note that since generators yield data indefinitely in a loop,
            # we must `break` after every image has been seen once.
            break
    return features, labels

train_features, train_labels = extract_features(train_dir, 1609)
validation_features, validation_labels = extract_features(validation_dir, 426)
test_features, test_labels = extract_features(test_dir, 392)
```

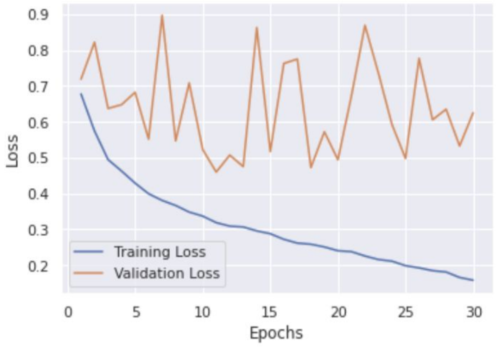
[Colab notebook]

We need to reshape the outputs so we can feed them into a dense layer - recall that dense layers take in vectors.

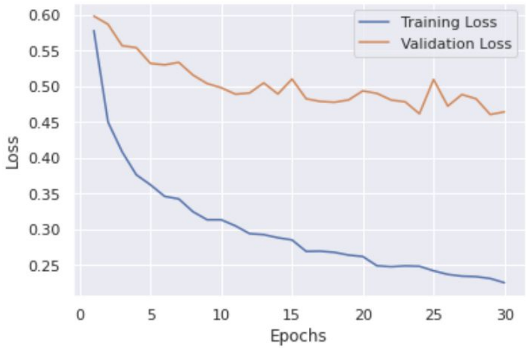
```python
train_features = np.reshape(train_features, (1609, 4 * 4 * 512))
validation_features = np.reshape(validation_features, (426, 4 * 4 * 512))
test_features = np.reshape(test_features, (392, 4 * 4 * 512))
```

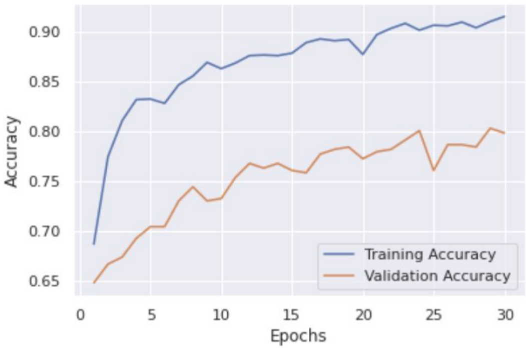
```python
model = keras.Sequential([
  layers.Dense(256, activation='relu', input_dim=4 * 4 * 512),
  layers.Dropout(0.5),
  layers.Dense(1, activation='sigmoid')
])

model.compile(optimizer=tf.keras.optimizers.RMSprop(lr=2e-5),
              loss='binary_crossentropy',
              metrics=['accuracy'])

history = model.fit(train_features, train_labels,
                    epochs=30,
                    batch_size=20,
                    validation_data=(validation_features, validation_labels))
```

Original CNN made from scratch



CNN using pretrained base





Feature Extraction
with Augmented Data

[Colab notebook]

```python
model = tf.keras.models.Sequential([
  conv_base,
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(256, activation='relu'),
  tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```python
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 4, 4, 512)         14714688
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
dense_2 (Dense)              (None, 256)               2097408
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 257
=================================================================
Total params: 16,812,353
Trainable params: 16,812,353
Non-trainable params: 0
```

We can add the base just like a layer to our network

```
model = tf.keras.models.Sequential([
    conv_base,
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 4, 4, 512)         14714688
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
dense_2 (Dense)              (None, 256)               2097408
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 257
=================================================================
Total params: 16,812,353
Trainable params: 16,812,353
Non-trainable params: 0
```

```python
print('This is the number of trainable weights '
      'before freezing the conv base:', len(model.trainable_weights))
```

```
This is the number of trainable weights before freezing the conv base: 30
```

```python
conv_base.trainable = False
```

```python
print('This is the number of trainable weights '
      'after freezing the conv base:', len(model.trainable_weights))
```

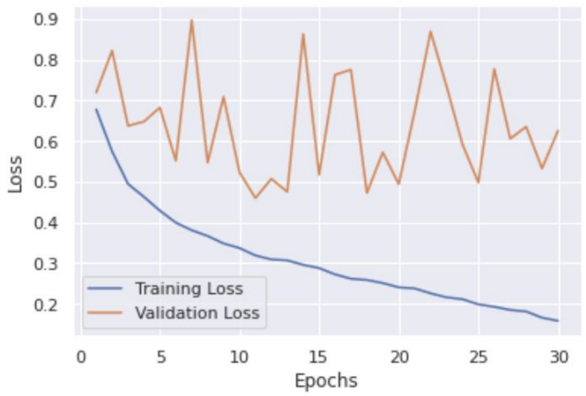
```
This is the number of trainable weights after freezing the conv base: 4
```

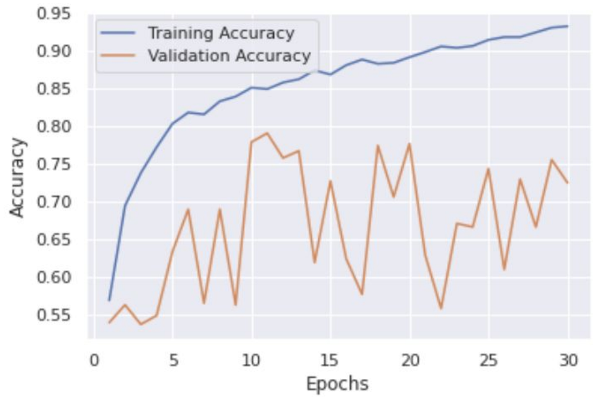
```python
from keras.preprocessing.image import ImageDataGenerator

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')

# Note that the validation data should not be augmented!
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        # This is the target directory
        train_dir,
        # All images will be resized to 150x150
        target_size=(150, 150),
        batch_size=20,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')

model.compile(optimizer=tf.keras.optimizers.RMSprop(lr=2e-5),
              loss='binary_crossentropy',
              metrics=['accuracy'])


history = model.fit(
      train_generator,
      steps_per_epoch=81,
      epochs=30,
      validation_data=validation_generator,
      validation_steps=22)
```

Note: do not run this code without access to a GPU.

Back to Colab notebook

Original CNN made from scratch with data augmentation





CNN using pretrained base with data augmentation