```
try:
  import google.colab
  IN_COLAB = True
except:
  IN_COLAB = False

import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import tensorflow as tf
import keras
from keras.datasets import fashion_mnist, cifar10
from keras.layers import Dense, Flatten, Normalization, Dropout, Conv2D, MaxPooling2D, RandomFlip, RandomRota
from keras.models import Sequential
from keras.losses import SparseCategoricalCrossentropy, CategoricalCrossentropy
from keras.callbacks import EarlyStopping
from keras.utils import np_utils
from keras import utils
import os
from keras.preprocessing.image import ImageDataGenerator

import matplotlib as mpl
import matplotlib.pyplot as plt
import datetime
import PIL
if IN_COLAB:
    !pip install --ignore-installed Pillow==9.0.0
    !pip install -U git+https://github.com/keisen/tf-keras-vis.git@4a90becb02ed3d44825300fcb807dd58157787ba
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting Pillow==9.0.0
  Downloading Pillow-9.0.0-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (4.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4.3/4.3 MB 23.5 MB/s eta 0:00:00
Installing collected packages: Pillow
Successfully installed Pillow-9.0.0
WARNING: The following packages were previously imported in this runtime:
  [PIL]
You must restart the runtime in order to use newly installed versions.
```

RESTART RUNTIME

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting git+https://github.com/keisen/tf-keras-vis.git@4a90becb02ed3d44825300fcb807dd58157787ba
  Cloning https://github.com/keisen/tf-keras-vis.git (to revision 4a90becb02ed3d44825300fcb807dd58157787
  Running command git clone --filter=blob:none --quiet https://github.com/keisen/tf-keras-vis.git /tmp/p
  Running command git rev-parse -q --verify 'sha^4a90becb02ed3d44825300fcb807dd58157787ba'
  Running command git fetch -q https://github.com/keisen/tf-keras-vis.git 4a90becb02ed3d44825300fcb807dc
  Running command git checkout -q 4a90becb02ed3d44825300fcb807dd58157787ba
  Resolved https://github.com/keisen/tf-keras-vis.git to commit 4a90becb02ed3d44825300fcb807dd58157787ba
  Preparing metadata (setup.py) ... done
Requirement already satisfied: scipy in /usr/local/lib/python3.9/dist-packages (from tf-keras-vis==0.7.0
Requirement already satisfied: pillow in /usr/local/lib/python3.9/dist-packages (from tf-keras-vis==0.7.
Collecting deprecated
  Downloading Deprecated-1.2.13-py2.py3-none-any.whl (9.6 kB)
Requirement already satisfied: imageio in /usr/local/lib/python3.9/dist-packages (from tf-keras-vis==0.7
Requirement already satisfied: packaging in /usr/local/lib/python3.9/dist-packages (from tf-keras-vis==0
Requirement already satisfied: wrapt<2,>=1.10 in /usr/local/lib/python3.9/dist-packages (from deprecated
Requirement already satisfied: numpy in /usr/local/lib/python3.9/dist-packages (from imageio->tf-keras-v
Building wheels for collected packages: tf-keras-vis
  Building wheel for tf-keras-vis (setup.py) ... done
  Created wheel for tf-keras-vis: filename=tf_keras_vis-0.7.0-py3-none-any.whl size=52254 sha256=f8bccfe
  Stored in directory: /root/.cache/pip/wheels/a9/7f/6c/d00d4e695bd6a01b65d487987251603921de126df6e0c4ac
Successfully built tf-keras-vis
Installing collected packages: deprecated, tf-keras-vis
Successfully installed deprecated-1.2.13 tf-keras-vis-0.7.0
```

Resources

You are subscribed to Colab Pro+. Learn more.
Available: 1496.75 compute units
Usage rate: approximately 13.08 per hour
You have 1 active session. Manage sessions

Python 3 Google Compute Engine backend (GPU)
Showing resources from 2:23 PM to 2:32 PM

System RAM
23.7 / 83.5 GB

GPU RAM
9.4 / 40.0 GB

Disk
28.6 / 166.8 GB

Transfer Learning

Feature Extraction and Classification

One of the key concepts needed with transfer learning is the separating of the feature extraction from the convolutional layers and the classification done in the fully connected layers.

- The convolutional layer finds features in the image. I.e. the output of the end of the convolutional layers is a set of image-y features.
- The fully connected layers take those features and classify the thing.

The idea behind this is that we allow someone (like Google) to train their fancy network on a bunch of fast computers, using millions and millions of images. These classifiers get very good at extracting features from objects.

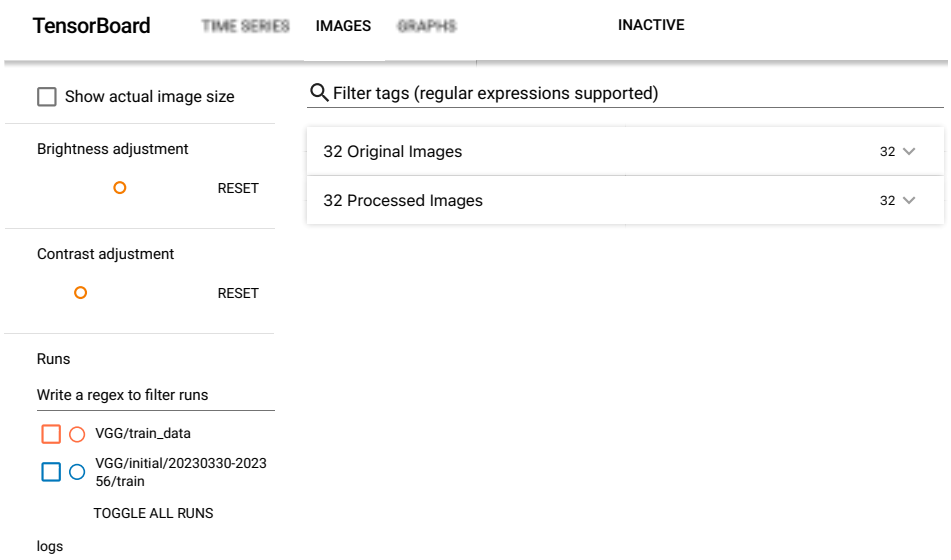
When using these models we take those convolutional layers and slap on our own classifier at the end, so the pretrained convolutional layers extract a bunch of features with their massive amount of training, then we use those features to predict our data!

Tensorboard Up-Front

we'll also launch the tensorboard prior to doing any training. Pay attention to the log locations in each callback, we can nest the logs in folders, then use the names and tensorboard's regex search to monitor each run as it progresses.

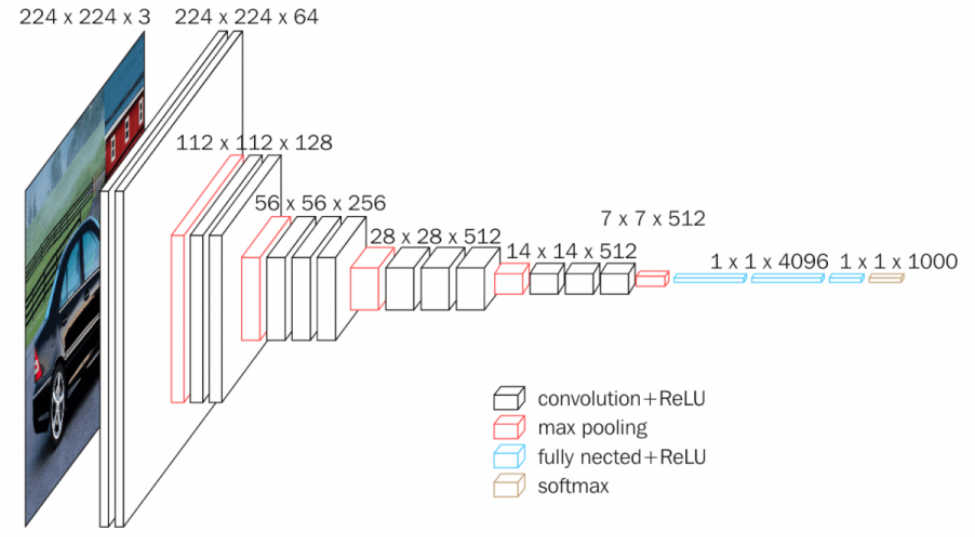
✓ 6s completed at 2:30 PM

```
%load_ext tensorboard
%tensorboard --logdir=logs
```



Download Model

There are several models that are pretrained and available to us to use. VGG16 is one developed to do image recognition, the name stands for "Visual Geometry Group" - a group of researchers at the University of Oxford who developed it, and '16' implies that this architecture has 16 layers. The model got ~93% on the ImageNet test that we mentioned a couple of weeks ago.



Slide Convolutional Layers from Classifier

When downloading the model we specifiy that we don't want the top - that's the classification part. When we remove the top we also allow the model to adapt to the shape of our images, so we specify the input size as well.

```
from keras.applications.vgg16 import VGG16
from keras.layers import Input
from keras.models import Model
from keras.applications.vgg16 import preprocess_input
```

Preprocessing Data

Our VGG 16 model comes with a preprocessing function to prepare the data in a way it is happy with. For this model the color encoding that it was trained on is different, so we should prepare the data properly to get good results.

```
import pathlib
from keras.applications.vgg16 import preprocess_input

dataset_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz"
data_dir = tf.keras.utils.get_file(origin=dataset_url,
                                   fname='flower_photos',
                                   untar=True)
data_dir = pathlib.Path(data_dir)

batch_size = 32
img_height = 224
img_width = 224
img_depth = 3

train_ds_orig = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)

val_ds_orig = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)

class_names = train_ds_orig.class_names
print(class_names)

def preprocess(images, labels):
  return tf.keras.applications.vgg16.preprocess_input(images), labels

train_ds = train_ds_orig.map(preprocess)
val_ds = val_ds_orig.map(preprocess)
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photo
228813984/228813984 [==============================] - 2s 0us/step
Found 3670 files belonging to 5 classes.
Using 2936 files for training.
Found 3670 files belonging to 5 classes.
Using 734 files for validation.
['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
```

Write Some Images for Tensorboard

We'll record some images, both pre and post processing. The VGG model wants images to use a different representation than RGB.

```
file_writer = tf.summary.create_file_writer("logs/VGG/train_data")
train_ds_iterator = train_ds_orig.as_numpy_iterator()
train_proc_iterator = train_ds.as_numpy_iterator()
samp_batch  = train_ds_iterator.next()
proc_batch = train_proc_iterator.next()

with file_writer.as_default():
    images = np.reshape(samp_batch[0].astype("uint8"), (-1, img_height, img_width, img_depth))
    procImages = np.reshape(proc_batch[0].astype("uint8"), (-1, img_height, img_width, img_depth))
    #images = samp_batch
    tf.summary.image("32 Original Images", images, max_outputs=32, step=0)
    tf.summary.image("32 Processed Images", procImages, max_outputs=32, step=0)
```

Add on New Classifier

If we look at the previous summary of the model we can see that the last layer we have is a MaxPool layer. When making our own CNN this is the last layer before we add in the "normal" stuff for making predictions, this is the same. We need to flatten the data, then use dense layers and an output layer to classify the predictions.

We end up with the pretrained parts finding features in images, and the custom part classifying images based on those features. If we think back to the concept of a convolutional network, the convolutional layers do the true heavy lifting in allowing us to do things like classify images, they take in the raw images and transform it into a set of features contained in that image. This ability to turn images into predictive features is the key - important parts of images like edges, corners, contrast, etc... are generic, and our borrowed model is excellent at finding these features in images. Our predicitons are unique, so we tweak the training of our model to make predictions for our data, into our classes - all based on the features that the borrowed model found!

Make Model

We take the model without the top, set the input image size, and then add our own classifier. Loading the model is simple, there are just a few things to specify:

- weights="imagenet" - tells the model to use the weights from its imagenet training. This is what brings the "smarts", so we want it.
- include_top=False - tells the model to not bring over the classifier bits that we wnat to replace.
- input_shape - the model is trained on specific data sizes (224x224x3). We can repurpose it by changing the input size.

We also set the VGG model that we download to be not trainable. We don't want to overwrite all of the training that already exists, coming from the original training. What we want to be trained are the final dense parts we added on to classify our specific scenario. All the weights in the convolutional layers are kept the same, as they have been developed through large amounts of training; the weights in the fully connected layers will be trained, resulting in a model that combines the "sight" of the pretrained model with the context of what we are trying to classify. The VGG bits will just show as though they are one layer in our model, and for training purposes that makes sense. We can also see in the "trainable params" listing in the summary, the large number of weights in that VGG section we are borrowing are not trainable - that's the smart part of the model.

Note: I think the "top" label is a bit misleading, as it isn't really the top, it is the part at the end that shows at the bottom of a summary.

```
## Loading VGG16 model
base_model = VGG16(weights="imagenet", include_top=False, input_shape=(img_height, img_width, img_depth))
base_model.trainable = False ## Not trainable weights

# Add Dense Stuff
flatten_layer = Flatten()
dense_layer_1 = Dense(512, activation='relu', kernel_regularizer='l2', bias_regularizer='l2')
drop_layer_1 = Dropout(.2)
dense_layer_2 = Dense(256, activation='relu', kernel_regularizer='l2', bias_regularizer='l2')
prediction_layer = Dense(5)

model = Sequential([
    base_model,
    flatten_layer,
    dense_layer_1,
    drop_layer_1,
    dense_layer_2,
    prediction_layer
])

model.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_t
58889256/58889256 [==============================] - 0s 0us/step
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 flatten (Flatten)           (None, 25088)             0

 dense (Dense)               (None, 512)               12845568

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 256)               131328

 dense_2 (Dense)             (None, 5)                 1285

=================================================================
Total params: 27,692,869
Trainable params: 12,978,181
Non-trainable params: 14,714,688
_________________________________________________________________
```

Compile and Train

Once the new Frakenstein model is built we finish the training process as we normally would. The only difference is that here the weights of the VGG part of the model are not being adjusted during the backpropagation steps, only the weights in the layers that we added at the end are. For many, if not most, applications, this approach of adapting a pretrained model will give the best real world results. Unless you happen to live in a data centre, you probably lack both the data and the processing capacity to train any model from scratch to be as good as those that we can download.

```
# Model
model.compile(
    optimizer=tf.keras.optimizers.Adam(),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=keras.metrics.SparseCategoricalAccuracy(name="accuracy"),
    run_eagerly=False
)

time_stamp = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
log_dir = "logs/VGG/initial/" + time_stamp
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1, write_graph=True, wr
stopping_callback = EarlyStopping(monitor='val_accuracy', patience=3, restore_best_weights=True, mode="max")
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(filepath="weights/VGG/initial/"+time_stamp+"model.hd
```

```
model.fit(train_ds,
            epochs=epochs,
            verbose=1,
            validation_data=val_ds,
            callbacks=[tensorboard_callback, stopping_callback, checkpoint_callback])
```

```
Epoch 1/10
92/92 [==============================] - 21s 93ms/step - loss: 16.5097 - accuracy: 0.7217 - val_loss: 11
Epoch 2/10
92/92 [==============================] - 6s 68ms/step - loss: 10.0129 - accuracy: 0.8948 - val_loss: 9.5
Epoch 3/10
92/92 [==============================] - 6s 65ms/step - loss: 8.1013 - accuracy: 0.9295 - val_loss: 8.51
Epoch 4/10
92/92 [==============================] - 6s 59ms/step - loss: 7.0846 - accuracy: 0.9384 - val_loss: 7.93
Epoch 5/10
92/92 [==============================] - 5s 57ms/step - loss: 6.3363 - accuracy: 0.9506 - val_loss: 7.34
Epoch 6/10
92/92 [==============================] - 5s 57ms/step - loss: 5.5866 - accuracy: 0.9561 - val_loss: 6.72
<keras.callbacks.History at 0x7fa1200ce5e0>
```

Fine Tune Models

Lastly, we can adapt the entire model to our data. We'll unfreeze the original model, and then train the model again. The key addition here is that we set the learning rate to be extremely low (here it is 2 orders of magnitude smaller than the default) so the model doesn't totally rewrite all of the weights while training, rather it will only change a little bit - fine tuning its predictions to the actual data! Here the oringal convolutional layers are trainable, and the weights will be adjusted during training, but we dial the learning rate way down so that our changes only impact the model a little bit. This is a greater degree of fine tuning than we get when we lock the VGG layers, but it is still mainly relying on the previous training of the VGG model.

The end result is a model that can take advantage of all of the training that the original model received before we downloaded it. That ability of extracting features from images is then reapplied to our data for making predictions based on the features identified in the original model. Finally we take the entire model and just gently train it to be a little more suited to our data. The best of all worlds!

```
#Save a copy of the above model for next test.
copy_model = model

base_model.trainable = True
model.summary()

model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-5),  # Low learning rate
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=keras.metrics.SparseCategoricalAccuracy(name="accuracy")
)

time_stamp = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
log_dir = "logs/VGG/fine_tune/" + time_stamp
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1, write_graph=True, wr
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(filepath="weights/VGG/fine_tune/"+time_stamp+"model.

model.fit(train_ds, epochs=epochs, validation_data=val_ds, verbose=1, callbacks=[tensorboard_callback, stoppi
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 flatten (Flatten)           (None, 25088)             0

 dense (Dense)               (None, 512)               12845568

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 256)               131328

 dense_2 (Dense)             (None, 5)                 1285

=================================================================
Total params: 27,692,869
Trainable params: 27,692,869
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
92/92 [==============================] - 20s 122ms/step - loss: 6.9410 - accuracy: 0.9469 - val_loss: 7.
Epoch 2/10
92/92 [==============================] - 9s 97ms/step - loss: 6.2686 - accuracy: 0.9710 - val_loss: 6.62
Epoch 3/10
92/92 [==============================] - 9s 97ms/step - loss: 5.8448 - accuracy: 0.9857 - val_loss: 6.24
Epoch 4/10
92/92 [==============================] - 8s 87ms/step - loss: 5.5076 - accuracy: 0.9925 - val_loss: 5.96
Epoch 5/10
92/92 [==============================] - 9s 96ms/step - loss: 5.2194 - accuracy: 0.9928 - val_loss: 5.70
Epoch 6/10
92/92 [==============================] - 8s 87ms/step - loss: 4.9769 - accuracy: 0.9918 - val_loss: 5.38
Epoch 7/10
92/92 [==============================] - 9s 96ms/step - loss: 4.7509 - accuracy: 0.9935 - val_loss: 5.16
Epoch 8/10
92/92 [==============================] - 9s 94ms/step - loss: 4.5397 - accuracy: 0.9952 - val_loss: 5.05
Epoch 9/10
92/92 [==============================] - 8s 86ms/step - loss: 4.3537 - accuracy: 0.9976 - val_loss: 4.82
Epoch 10/10
```

```
92/92 [==============================] - 8s 86ms/step - loss: 4.1842 - accuracy: 0.9969 - val_loss: 4.68
<keras.callbacks.History at 0x7fa1002c3f40>
```

Transfer + Fine Tuning Results

Yay, that's probably pretty accurate! In initial testing with 1 epoch, I got results around 80% before the fine tuning, and over 85% after the fine tuning. That's with 1 epoch! Other runs where we allow it to tune more trend to be even better - allowing 5 epochs of training + 5 epochs of fine tuning, my validation accuracy was around 90% and the training accuracy was nearing 100% - we could likely do even better with more aggressive regularization.

This will likely be a great approach for something like image recognition!

Where is the Model Looking?

One of the things that we may wonder is how our models make decisions, or what are the looking at to do so. A tool that can help illustrate that is called a salience map. A salience map shows a visual representation of what parts of the image are impacting the final prediction the most. While the CNN process is largely a black box, this is one way to gain a little insight on what is going on.

Visualize Focus

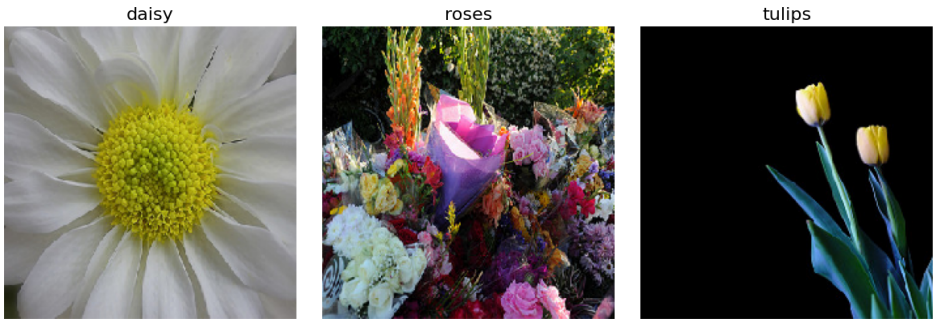
```
if IN_COLAB:
    from tensorflow.keras.preprocessing.image import load_img
    from tf_keras_vis.utils.scores import CategoricalScore

    # Image titles
    image_titles = ['daisy', 'roses', 'tulips']
    score = CategoricalScore([0, 2, 4])

    # Load images and Convert them to a Numpy array
    img1 = load_img('/root/.keras/datasets/flower_photos/daisy/100080576_f52e8ee070_n.jpg', target_size=(224,
    img2 = load_img('/root/.keras/datasets/flower_photos/roses/10894627425_ec76bbc757_n.jpg', target_size=(22
    #img3 = load_img('/root/.keras/datasets/flower_photos/tulips/10128546863_8de70c610d.jpg', target_size=(22
    img3 = load_img("/root/.keras/datasets/flower_photos/tulips/12764617214_12211c6a0c_m.jpg", target_size=(2
    images = np.asarray([np.array(img1), np.array(img2), np.array(img3)])

    # Preparing input data for VGG16
    X = preprocess_input(images)

    # Rendering
    f, ax = plt.subplots(nrows=1, ncols=3, figsize=(12, 4))
    for i, title in enumerate(image_titles):
        ax[i].set_title(title, fontsize=16)
        ax[i].imshow(images[i])
        ax[i].axis('off')
    plt.tight_layout()
    plt.show()
```



Show Saliency

The bright spots in the image are the areas that the model is focusing on to make its prediction. The darker areas are not as important. We can think of this as a rough approximation of feature importance from something like a tree, only in 2D.

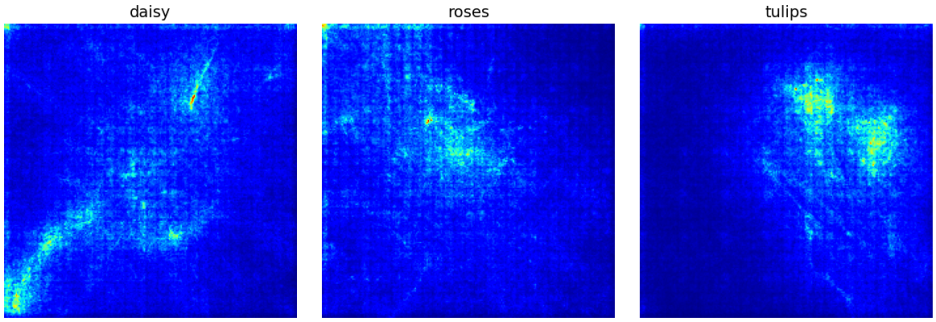
Using a salience map in detail to tune our models goes beyond the scope of what we are going to do, but it does allow us to get at least some insight. The most direct thing that we can do is that we can figure out which parts of images are relied on for the model to do its job, this can help us to understand what the model is looking for. For something like image recognition, this could lead you to think about how the images are processed - for example, it is very common to snip out parts of larger images, usually the center, for use in a predictive model. This could show us evidence of if we are capturing the important parts or if we should modify that image prep process. I we were considering the padding decision, this could also give us an idea of if the edges matter or not for the model's predicitons. If the most important parts of the image are the "thing", then we are likely doing well, if they most important parts are background or the periphery, then we may been to change some things.

```
if IN_COLAB:
    from tf_keras_vis.utils.model_modifiers import ReplaceToLinear
    replace2linear = ReplaceToLinear()
    from tensorflow.keras import backend as K
    from tf_keras_vis.saliency import Saliency
    # from tf_keras_vis.utils import normalize

    # Create Saliency object.
    saliency = Saliency(model, model_modifier=replace2linear, clone=True)

    # Generate saliency map
    saliency_map = saliency(score, X)
    # Generate saliency map with smoothing that reduce noise by adding noise
    saliency_map = saliency(score,
                            X,
                            smooth_samples=20, # The number of calculating gradients iterations.
                            smooth_noise=0.20) # noise spread level.

    # Render
    f, ax = plt.subplots(nrows=1, ncols=3, figsize=(12, 4))
    for i, title in enumerate(image_titles):
        ax[i].set_title(title, fontsize=14)
        ax[i].imshow(saliency_map[i], cmap='jet')
        ax[i].axis('off')
    plt.tight_layout()
    plt.show()
```



More Drastic Retraining

If we are extra ambitious we can also potentially slice the model even deeper, and take smaller portions to mix with our own models. The farther "into" the model you slice, the more of the original training will be removed and the more the model will learn from our training data. If done, this is a balancing act - we want to keep all of the smarts that the model has gotten from the original training, while getting the benefits of adaptation to our data.

This is something that is hard to just eyeball - to splice parts of models together and create something that is actually superior likely requries a lot of experimentation, a solid understanding of the model's problem you're addressing, and some domain knowledge. For something like this adaptation of the VGG model, we'd probably start with some idea of what the model was weak at, build an understanding of what types of features it was extracting along the way, and insert our own layers where we think it would be most beneficial.

```
## Loading VGG16 model
base_model = VGG16(weights="imagenet", include_top=False, input_shape=(img_height, img_width, img_depth))
#base_model.trainable = False ## Not trainable weights
base_model.summary()
```

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0

block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792

block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928

block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0

block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856

block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584

block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0

block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168

block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080

block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080

block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
```

```
 block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

Freeze the First 12 Layers

We will set the first 12 layers to be frozen, and leave the rest open to be trained.

```
for layer in base_model.layers[:12]:
    layer.trainable = False
base_model.summary()
```

```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 224, 224, 3)]     0

 block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792

 block1_conv2 (Conv2D)       (None, 224, 224, 64)      36928

 block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0

 block2_conv1 (Conv2D)       (None, 112, 112, 128)     73856

 block2_conv2 (Conv2D)       (None, 112, 112, 128)     147584

 block2_pool (MaxPooling2D)  (None, 56, 56, 128)       0

 block3_conv1 (Conv2D)       (None, 56, 56, 256)       295168

 block3_conv2 (Conv2D)       (None, 56, 56, 256)       590080

 block3_conv3 (Conv2D)       (None, 56, 56, 256)       590080

 block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0

 block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

=================================================================
Total params: 14,714,688
Trainable params: 11,799,040
Non-trainable params: 2,915,648
_________________________________________________________________
```

More Retraining

Now we have larger portions of the model that can be trained. We will be losing some of the pretrained knowldge, replacing it with the training coming from our data. If we look at the trainable params above, there are a bunch that are trainable and a bunch that aren't.

We are playing with fire here! Taking away more and more of the "smart" model will be risky for actual performance, we are pretty likely to make things worse as we go father and farther into removing the old training.

```
# Add Dense Stuff
flatten_layer = Flatten()
dense_layer_1 = Dense(512, activation='relu', kernel_regularizer='l2', bias_regularizer='l2')
dense_layer_2 = Dense(256, activation='relu', kernel_regularizer='l2', bias_regularizer='l2')
prediction_layer = Dense(5)

model = Sequential([
    base_model,
```

```
    flatten_layer,
    dense_layer_1,
    dense_layer_2,
    prediction_layer
])

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 flatten_1 (Flatten)         (None, 25088)             0

 dense_3 (Dense)             (None, 512)               12845568

 dense_4 (Dense)             (None, 256)               131328

 dense_5 (Dense)             (None, 5)                 1285

=================================================================
Total params: 27,692,869
Trainable params: 24,777,221
Non-trainable params: 2,915,648
_________________________________________________________________
```

```
# Model
model.compile(
    optimizer=tf.keras.optimizers.Adam(),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=keras.metrics.SparseCategoricalAccuracy(name="accuracy")
)

time_stamp = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
log_dir = "logs/VGG/drastic/" + time_stamp
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1, write_graph=True, wr
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(filepath="weights/VGG/drastic/"+time_stamp+"model.hd

model.fit(train_ds,
            epochs=epochs,
            verbose=1,
            validation_data=val_ds,
            callbacks=[tensorboard_callback, stopping_callback, checkpoint_callback])
```

```
Epoch 1/10
92/92 [==============================] - 10s 68ms/step - loss: 18.7599 - accuracy: 0.2354 - val_loss: 7.
Epoch 2/10
92/92 [==============================] - 5s 57ms/step - loss: 6.9268 - accuracy: 0.2446 - val_loss: 5.9
Epoch 3/10
92/92 [==============================] - 5s 54ms/step - loss: 5.4806 - accuracy: 0.2459 - val_loss: 5.04
Epoch 4/10
92/92 [==============================] - 5s 53ms/step - loss: 4.7425 - accuracy: 0.2459 - val_loss: 4.47
<keras.callbacks.History at 0x7fa08cf744f0>
```

Results

We likely see worse results when retraining more of the model, that's to be expected. In general, replacing the classifier and possibly some low learning rate fine tuning is the best solution for most cases like this.

Exercise - ResNet50

This is another pretrained network, containing 50 layers. We can use this one similarly to the last. Try to use transfer learning along with some of your added layers to predict.

```
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input, decode_predictions

def preprocess50(images, labels):
  return tf.keras.applications.resnet50.preprocess_input(images), labels

train_ds = train_ds_orig.map(preprocess50)
val_ds = val_ds_orig.map(preprocess50)


# Make Model
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(img_height, img_width, img_depth))
base_model.trainable = False ## Not trainable weights

# Add Dense Stuff
flatten_layer = Flatten()
dense_layer_1 = Dense(512, activation='relu', kernel_regularizer='l2', bias_regularizer='l2')
dense_layer_2 = Dense(256, activation='relu', kernel_regularizer='l2', bias_regularizer='l2')
dense_layer_3 = Dense(96, activation='relu', kernel_regularizer='l2', bias_regularizer='l2')
prediction_layer = Dense(5)

model = Sequential([
    base_model,
    flatten_layer,
    dense_layer_1,
```

```
    dense_layer_2,
    dense_layer_3,
    prediction_layer
])

model.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weigh
94765736/94765736 [==============================] - 1s 0us/step
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50 (Functional)       (None, 7, 7, 2048)        23587712

 flatten_2 (Flatten)         (None, 100352)            0

 dense_6 (Dense)             (None, 512)               51380736

 dense_7 (Dense)             (None, 256)               131328

 dense_8 (Dense)             (None, 96)                24672

 dense_9 (Dense)             (None, 5)                 485

=================================================================
Total params: 75,124,933
Trainable params: 51,537,221
Non-trainable params: 23,587,712
_________________________________________________________________
```

Train New Classifier

Train model with new classifier.

```
# Model
model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
            optimizer="adam",
            metrics=keras.metrics.SparseCategoricalAccuracy(name="accuracy"))

log_dir = "logs/50/initial/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(filepath="weights/50/initial/"+time_stamp+"model.hdf

model.fit(train_ds,
            epochs=epochs,
            verbose=1,
            validation_data=val_ds,
            callbacks=[tensorboard_callback, stopping_callback, checkpoint_callback])
```

```
Epoch 1/10
92/92 [==============================] - 16s 119ms/step - loss: 14.8141 - accuracy: 0.7752 - val_loss: 8
Epoch 2/10
92/92 [==============================] - 9s 93ms/step - loss: 4.9635 - accuracy: 0.9441 - val_loss: 3.68
Epoch 3/10
92/92 [==============================] - 7s 74ms/step - loss: 2.8526 - accuracy: 0.9493 - val_loss: 2.79
Epoch 4/10
92/92 [==============================] - 7s 74ms/step - loss: 2.4835 - accuracy: 0.9370 - val_loss: 2.65
Epoch 5/10
92/92 [==============================] - 7s 76ms/step - loss: 2.4803 - accuracy: 0.9315 - val_loss: 2.57
<keras.callbacks.History at 0x7fa08c0d0070>
```

Attempt Retraining Entire Model to Fine Tune

We can attempt to unlock the model and retrain in fine tuning.

```
base_model.trainable = True
model.summary()

model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-6),  # Low learning rate
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=keras.metrics.SparseCategoricalAccuracy(name="accuracy")
)

log_dir = "logs/50/fine_tune/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(filepath="weights/50/fine_tune/"+time_stamp+"model.h

model.fit(train_ds, epochs=epochs, validation_data=val_ds, verbose=1, callbacks=[tensorboard_callback, stoppi
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50 (Functional)       (None, 7, 7, 2048)        23587712

 flatten_2 (Flatten)         (None, 100352)            0

 dense_6 (Dense)             (None, 512)               51380736

 dense_7 (Dense)             (None, 256)               131328

 dense_8 (Dense)             (None, 96)                24672
```

```
 dense_9 (Dense)             (None, 5)                 485

=================================================================
Total params: 75,124,933
Trainable params: 75,071,813
Non-trainable params: 53,120
_________________________________________________________________
Epoch 1/10
92/92 [==============================] - 54s 162ms/step - loss: 3.6938 - accuracy: 0.8576 - val_loss: 3.
Epoch 2/10
92/92 [==============================] - 11s 113ms/step - loss: 3.4835 - accuracy: 0.9200 - val_loss: 3.
Epoch 3/10
92/92 [==============================] - 11s 114ms/step - loss: 3.3801 - accuracy: 0.9438 - val_loss: 3.
Epoch 4/10
92/92 [==============================] - 11s 121ms/step - loss: 3.3310 - accuracy: 0.9489 - val_loss: 3.
<keras.callbacks.History at 0x7fa06a4f6370>
```

Transfer Learning Conclusion

Transfer learning is common, especially when working with things like images. Pretrained models that have seen millions upon millions of images get very good at "understanding" what is in an image, or extracting important features from those images. This basic ability to "see" image data is interchangeable between different types of image tasks that we may want to do. For image data, natural language, audio, video, it is likely that one of these large models will be more capable of extracting features from the data than we could ever hope to do from scratch. Since the basics of "seeing a thing" or "reading a sentence" is the same no matter the specific application, that ability to process the data that our pretrained models have can be repurposed to our specific ends.

We can see lots of scenarios in the real world where people are adapting image recognition models trained by Google to do things like recognize objects in their home security system, or language models like the GPT family being adapted to better understand domain specific language. We'll likely see more of this, as the benefits of training on massive amounts of data are hard, if not impossible, to replicate.