Detailed VGG16 Transfer Learning Code with Comments (Practical 6)

```
# -------------------- Cell 1: Imports --------------------
import tensorflow_datasets as tfds          # utility to download/prep standard datasets (tf_flowers used here)
import tensorflow as tf                     # main TensorFlow library (includes Keras)
from tensorflow.keras.utils import to_categorical  # helper to convert integer labels to one-hot vectors

# -------------------- Cell 2: Download / load dataset --------------------
# tf_flowers is a dataset of flower images; tfds.load handles download + preparation.
# We request two splits: first 70% for train, next 30% for test.
# batch_size=-1 asks tfds to return the entire split as a single tensor/array (not an iterator).
# as_supervised=True makes the dataset return (image, label) pairs.
(train_ds, train_labels), (test_ds, test_labels) = tfds.load(
    "tf_flowers",
     split=["train[:70%]", "train[:30%]"],  # create train/test split from the collected examples
     batch_size=-1,                          # load the whole split into memory as a single batch (careful with RAM)
     as_supervised=True                      # return (image, label) pairs instead of dicts
)

# NOTE: When batch_size=-1, tfds returns numpy arrays (or tf.Tensor) containing all images and all labels.
# This is convenient for small datasets, but for large datasets you should use tf.data pipelines.

# -------------------- Cell 3: Inspect a sample image shape --------------------
# Check the shape of the first loaded image to know the current resolution.
# Example output in your run: TensorShape([442, 1024, 3]) which shows images are large and not uniform.
train_ds[0].shape  # prints shape of the first image (height, width, channels)

# -------------------- Cell 4: Resize images --------------------
# Pre-trained CNNs (like VGG16) expect fixed-size inputs (e.g., 150x150 or 224x224).
# Resize all images to (150, 150). tf.image.resize can operate on a tensor of images.
train_ds = tf.image.resize(train_ds, (150, 150))   # resize all training images to 150x150x3
test_ds = tf.image.resize(test_ds, (150, 150))    # resize all test images to 150x150x3

# Re-check shape to confirm resize (expected: TensorShape([150, 150, 3]) per image)
train_ds[0].shape

# -------------------- Cell 5: Convert labels to one-hot (categorical) --------------------
# Pre-trained classifiers often use categorical_crossentropy which expects one-hot labels.
# to_categorical converts integer labels (0..num_classes-1) into one-hot vectors.
# tf_flowers has 5 classes so use num_classes=5
train_labels = to_categorical(train_labels, num_classes=5)
test_labels = to_categorical(test_labels, num_classes=5)
```

```
# -------------------- Cell 6: Load pre-trained VGG16 base and preprocessing util --------------------
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.applications.vgg16 import preprocess_input

# Load VGG16 without its top (fully connected) layers because we'll add our own classification head.
# weights="imagenet" loads pre-trained weights learned on ImageNet.
# include_top=False excludes the original Dense layers; we only keep convolutional feature extractor.
# input_shape is set to the shape of our resized images (150,150,3).
base_model = VGG16(weights="imagenet", include_top=False, input_shape=train_ds[0].shape)

# NOTE: download of weights occurs the first time; subsequent runs will use cached weights.

# -------------------- Cell 7: Freeze base model (transfer learning) --------------------
# Freezing means we do not update the parameters of the pre-trained convolutional layers during training.
# This reduces training time and avoids destroying useful pre-trained features.
base_model.trainable = False

# -------------------- Cell 8: Preprocess inputs before feeding into VGG16 --------------------
# Pre-trained models expect inputs preprocessed in the same way as during their ImageNet training.
# preprocess_input performs necessary scaling and color channel transformations specific to VGG16.
train_ds = preprocess_input(train_ds)
test_ds = preprocess_input(test_ds)

# -------------------- Cell 9: Inspect base model summary (optional, informative) --------------------
# .summary() prints each layer's output shape and parameter count so you can understand capacity.
base_model.summary()

# -------------------- Cell 10: Build the top (classification) layers and full model --------------------
from tensorflow.keras import layers, models  # layer and model constructors

# Define additional layers we add on top of the frozen VGG16 feature extractor:
flatten_layer = layers.Flatten()                        # flatten conv feature maps to a vector
dense_layer_1 = layers.Dense(50, activation='relu')   # first Dense layer with ReLU activation
dense_layer_2 = layers.Dense(20, activation='relu')   # second Dense layer with ReLU activation
prediction_layer = layers.Dense(5, activation='softmax')   # final softmax layer for 5-class classification

# Create a Sequential model stacking: VGG16 base -> flatten -> dense1 -> dense2 ->
```

```
prediction
model = models.Sequential([
    base_model,         # frozen convolutional base
    flatten_layer,      # flatten feature maps to feed dense layers
    dense_layer_1,      # learn new task-specific features
    dense_layer_2,      # intermediate dense layer
    prediction_layer    # output probabilities over 5 classes
])

# -------------------- Cell 11: Compile the model --------------------
# Choose optimizer, loss and metrics:
# - optimizer='adam' : adaptive optimizer, good default for many tasks
# - loss='categorical_crossentropy' : appropriate for one-hot encoded multi-class labels
# - metrics=['accuracy'] : report accuracy during training
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy'],
)

# -------------------- Cell 12: Train the model --------------------
# model.fit trains the model using the provided inputs and labels.
# - train_ds and train_labels are numpy arrays/tensors (we loaded entire splits into memory).
# - epochs=10 : number of passes over the training set
# - validation_split=0.2 : hold out 20% of training data for validation (Note: this splits arrays internally)
# - batch_size=32 : size of mini-batches used for gradient updates
history = model.fit(train_ds, train_labels, epochs=10, validation_split=0.2, batch_size=32)

# NOTES:
# - Because base_model.trainable = False, only the new Dense layers are trained (few parameters).
# - Training time per epoch may still be long because images are large and computation is heavy.

# -------------------- Cell 13: Evaluate on test data --------------------
# Evaluate returns the loss and metrics specified in compile() on held-out test set.
los, accurac = model.evaluate(test_ds, test_labels)
print("Loss: ", los, "Accuracy: ", accurac)  # print numeric results

# -------------------- Cell 14: Plot accuracy curve --------------------
import matplotlib.pyplot as plt

# Plot training accuracy across epochs
plt.plot(history.history['accuracy'])     # training accuracy recorded per epoch
plt.title('ACCURACY')                     # plot title
plt.ylabel('accuracy')                    # y-axis label
plt.xlabel('epoch')                       # x-axis label
plt.legend(['train'], loc='upper left')   # legend for the single plotted line
plt.show()                                # display the plot
```