In [2]:
```
# import the necessary packages
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.applications import VGG16
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.preprocessing.image import load_img
from tensorflow.keras.applications import imagenet_utils
from tensorflow.keras.models import Model

import tensorflow as tf

import numpy as np
import imutils
import cv2
```

```
/Users/shrihanshumishra/anaconda3/lib/python3.11/site-packages/pan
das/core/arrays/masked.py:60: UserWarning: Pandas requires version
'1.3.6' or newer of 'bottleneck' (version '1.3.5' currently instal
led).
  from pandas.core import (
```

In [3]:
```
# load the pre-trained CNN from disk
model = VGG16(weights="imagenet")
```

```
2024-10-08 02:22:35.982707: I metal_plugin/src/device/metal_devic
e.cc:1154] Metal device set to: Apple M1
2024-10-08 02:22:35.982736: I metal_plugin/src/device/metal_devic
e.cc:296] systemMemory: 8.00 GB
2024-10-08 02:22:35.982739: I metal_plugin/src/device/metal_devic
e.cc:313] maxCacheSize: 2.67 GB
2024-10-08 02:22:35.982753: I tensorflow/core/common_runtime/plugg
able_device/pluggable_device_factory.cc:305] Could not identify NU
MA node of platform GPU ID 0, defaulting to 0. Your kernel may not
have been built with NUMA support.
2024-10-08 02:22:35.982763: I tensorflow/core/common_runtime/plugg
able_device/pluggable_device_factory.cc:271] Created TensorFlow de
vice (/job:localhost/replica:0/task:0/device:GPU:0 with 0 MB memor
y) -> physical PluggableDevice (device: 0, name: METAL, pci bus i
d: <undefined>)
```

In [4]:
```
# load the original image from disk (in OpenCV format) and then
# resize the image to its target dimensions
orig = cv2.imread("/Users/shrihanshumishra/Downloads/beagle.jpg")
resized = cv2.resize(orig, (224, 224))
```

In [5]:
```
# load the input image from disk (in Keras/TensorFlow format) and
# preprocess it
image = load_img("/Users/shrihanshumishra/Downloads/beagle.jpg", ta
image = img_to_array(image)
image = np.expand_dims(image, axis=0)
image = imagenet_utils.preprocess_input(image)
```

In [6]:
```
# use the network to make predictions on the input imag and find
# the class label index with the largest corresponding probability
model = VGG16(weights="imagenet")
preds = model.predict(image)
i = np.argmax(preds[0])
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 283ms/step

2024-10-08 02:22:41.380489: I tensorflow/core/grappler/optimizers/
custom_graph_optimizer_registry.cc:117] Plugin optimizer for devic
e_type GPU is enabled.
```

In [7]:
```
# decode the ImageNet predictions to obtain the human-readable labe
decoded = imagenet_utils.decode_predictions(preds)
(imagenetID, label, prob) = decoded[0][0]
label = "{}: {:.2f}%".format(label, prob * 100)
print("[INFO] {}".format(label))
```

```
[INFO] beagle: 73.94%
```

In [8]:
```
class GradCAM:
    def __init__(self, model, classIdx, layerName=None):
        # store the model, the class index used to measure the clas
        # activation map, and the layer to be used when visualizing
        # the class activation map
        self.model = model
        self.classIdx = classIdx
        self.layerName = layerName

        # if the layer name is None, attempt to automatically find
        # the target output layer
        if self.layerName is None:
            self.layerName = self.find_target_layer()

    def find_target_layer(self):
        # attempt to find the final convolutional layer in the netw
        # by looping over the layers of the network in reverse orde
        for layer in reversed(self.model.layers):
            # check to see if the layer has a 4D output
            if len(layer.output.shape) == 4:
                return layer.name

        # otherwise, we could not find a 4D layer so the GradCAM
        # algorithm cannot be applied
        raise ValueError("Could not find 4D layer. Cannot apply Gra
```

```
    def compute_heatmap(self, image, eps=1e-8):
        # construct our gradient model by supplying (1) the inputs
        # to our pre-trained model, (2) the output of the (presumab
        # final 4D layer in the network, and (3) the output of the
        # softmax activations from the model
        gradModel = Model(inputs=[self.model.inputs], outputs= [sel

        # record operations for automatic differentiation
        with tf.GradientTape() as tape:
            # cast the image tensor to a float-32 data type, pass t
            # image through the gradient model, and grab the loss
            # associated with the specific class index
            inputs = tf.cast(image, tf.float32)
            (convOutputs, predictions) = gradModel(inputs)
            loss = predictions[:, self.classIdx]

        # use automatic differentiation to compute the gradients
        grads = tape.gradient(loss, convOutputs)

        # compute the guided gradients
        castConvOutputs = tf.cast(convOutputs > 0, "float32")
        castGrads = tf.cast(grads > 0, "float32")
        guidedGrads = castConvOutputs * castGrads * grads

        # the convolution and guided gradients have a batch dimensi
        # (which we don't need) so let's grab the volume itself and
        # discard the batch
        convOutputs = convOutputs[0]
        guidedGrads = guidedGrads[0]

        # compute the average of the gradient values, and using the
        # as weights, compute the ponderation of the filters with
        # respect to the weights
        weights = tf.reduce_mean(guidedGrads, axis=(0, 1))
        cam = tf.reduce_sum(tf.multiply(weights, convOutputs), axis

        # grab the spatial dimensions of the input image and resize
        # the output class activation map to match the input image
        # dimensions
        (w, h) = (image.shape[2], image.shape[1])
        heatmap = cv2.resize(cam.numpy(), (w, h))

        # normalize the heatmap such that all values lie in the ran
        # [0, 1], scale the resulting values to the range [0, 255],
        # and then convert to an unsigned 8-bit integer
        numer = heatmap - np.min(heatmap)
        denom = (heatmap.max() - heatmap.min()) + eps
        heatmap = numer / denom
        heatmap = (heatmap * 255).astype("uint8")

        # return the resulting heatmap to the calling function
        return heatmap
```

```
    def overlay_heatmap(self, heatmap, image, alpha=0.5,
        colormap=cv2.COLORMAP_JET):
        # apply the supplied color map to the heatmap and then
        # overlay the heatmap on the input image
        heatmap = cv2.applyColorMap(heatmap, colormap)
        output = cv2.addWeighted(image, alpha, heatmap, 1 - alpha,

        # return a 2-tuple of the color mapped heatmap and the outp
        # overlaid image
        return (heatmap, output)
```

In [9]:
```
# initialize our gradient class activation map and build the heatma
cam = GradCAM(model, i)
heatmap = cam.compute_heatmap(image)
```

In [10]:
```
# resize the resulting heatmap to the original input image dimensio
# and then overlay heatmap on top of the image
heatmap = cv2.resize(heatmap, (orig.shape[1], orig.shape[0]))
(heatmap, output) = cam.overlay_heatmap(heatmap, orig, alpha=0.5)
```

In [*]:
```
# draw the predicted label on the output image
cv2.rectangle(output, (0, 0), (340, 40), (0, 0, 0), -1)
cv2.putText(output, label, (10, 25), cv2.FONT_HERSHEY_SIMPLEX, 0.8,

# display the original image and resulting heatmap and output image
# to our screen
output = np.vstack([orig, heatmap, output])
output = imutils.resize(output, height=700)
cv2.imshow("Output", output)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: