Introduction To GradCAM (Part 1) - Lecture Notebook

In this lecture notebook we'll be looking at an introduction to Grad-CAM, a powerful technique for interpreting Convolutional Neural Networks. Grad-CAM stands for Gradient-weighted Class Activation Mapping.

CNN's are very flexible models and their great predictive power comes at the cost of losing interpretability (something that is true for all Artificial Neural Networks). Grad-CAM attempts to solve this by giving us a graphical visualisation of parts of an image that are the most relevant for the CNN when predicting a particular class.

Aside from working on some Grad-CAM concepts we'll also look at how we can use Keras to access some concrete information of our model. Let's dive into it!

In [1]:
```
import keras
from keras import backend as K
from util import *
```

```
Using TensorFlow backend.
```

The `load_C3M3_model()` function has been taken care of and its internals are out of the scope of this notebook. But if it intrigues you, you can take a look at it in `util.py`

In [2]:
```
# Load the model we are going to be using
model = load_C3M3_model()
```

```
Got loss weights
Loaded DenseNet
Added layers
Compiled Model
Loaded Weights
```

As you may already know, we can check the architecture of our model using the `summary()` method.

After running the code block below we'll see that this model has a lot of layers. One advantage of Grad-CAM over previous attempts of interpreting CNN's (such as CAM) is that it is architecture agnostic. This means it can be used for CNN's with complex architectures such as this one:

In [3]:
```
# Print all of the model's layers
model.summary()
```

```
_____
Layer (type)                 Output Shape              Param #   Connected to
==========
input_1 (InputLayer)         (None, None, None, 3 0
_____
zero_padding2d_1 (ZeroPadding2D (None, None, None, 3 0         input_1[0][0]
_____
conv1/conv (Conv2D)          (None, None, None, 6 9408        zero_padding2d_1[0][0]
_____
conv1/bn (BatchNormalization)   (None, None, None, 6 256         conv1/conv[0][0]
_____
conv1/relu (Activation)      (None, None, None, 6 0           conv1/bn[0][0]
_____
zero_padding2d_2 (ZeroPadding2D (None, None, None, 6 0         conv1/relu[0][0]
_____
pool1 (MaxPooling2D)         (None, None, None, 6 0           zero_padding2d_2[0][0]
_____
conv2_block1_0_bn (BatchNormali (None, None, None, 6 256         pool1[0][0]
_____
conv2_block1_0_relu (Activation (None, None, None, 6 0           conv2_block1_0_bn[0][0]
```

Keras models include abundant information about the elements that make them up. You can check all of the available methods and attributes of this class by using the `dir()` method:

In [4]:
```
# Printing out methods and attributes for Keras model
print(f"Keras' models have the following methods and attributes: \n\n{dir(model)}")
```

```
Keras' models have the following methods and attributes:

['__call__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute_
_', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce_
_', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_add_inbound_nod
e', '_built', '_check_num_samples', '_check_trainable_weights_consistency', '_collected_trainable_weights', '_container_node
s', '_feed_input_names', '_feed_input_shapes', '_feed_inputs', '_feed_loss_fns', '_feed_output_names', '_feed_output_shapes',
'_feed_outputs', '_feed_sample_weight_modes', '_feed_sample_weights', '_feed_targets', '_fit_loop', '_function_kwargs', '_get_
node_attribute_at_index', '_inbound_nodes', '_internal_input_shapes', '_internal_output_shapes', '_make_predict_function', '_m
ake_test_function', '_make_train_function', '_node_key', '_nodes_by_depth', '_outbound_nodes', '_output_mask_cache', '_output_
shape_cache', '_output_tensor_cache', '_per_input_losses', '_per_input_updates', '_predict_loop', '_standardize_user_data', '_
test_loop', '_updated_config', 'add_loss', 'add_update', 'add_weight', 'assert_input_compatibility', 'build', 'built', 'call',
'compile', 'compute_mask', 'compute_output_shape', 'count_params', 'evaluate', 'evaluate_generator', 'fit', 'fit_generator',
'from_config', 'get_config', 'get_input_at', 'get_input_mask_at', 'get_input_shape_at', 'get_layer', 'get_losses_for', 'get_ou
tput_at', 'get_output_mask_at', 'get_output_shape_at', 'get_updates_for', 'get_weights', 'input', 'input_layers', 'input_layer
s_node_indices', 'input_layers_tensor_indices', 'input_mask', 'input_names', 'input_shape', 'input_spec', 'inputs', 'layers',
'layers_by_depth', 'load_weights', 'loss', 'loss_functions', 'loss_weights', 'losses', 'metrics', 'metrics_names', 'metrics_te
nsors', 'metrics_updates', 'name', 'non_trainable_weights', 'optimizer', 'output', 'output_layers', 'output_layers_node_indice
s', 'output_layers_tensor_indices', 'output_mask', 'output_names', 'output_shape', 'outputs', 'predict', 'predict_function',
'predict_generator', 'predict_on_batch', 'reset_states', 'run_internal_graph', 'sample_weight_mode', 'sample_weight_modes', 's
ample_weights', 'save', 'save_weights', 'set_weights', 'state_updates', 'stateful', 'stateful_metric_functions', 'stateful_met
ric_names', 'summary', 'supports_masking', 'targets', 'test_function', 'test_on_batch', 'to_json', 'to_yaml', 'total_loss', 't
rain_function', 'train_on_batch', 'trainable', 'trainable_weights', 'updates', 'uses_learning_phase', 'weighted_metrics', 'wei
ghts']
```

Wow, this certainly is a lot! These models are indeed very complex.

What we are interested in are the layers of the model which can be easily accessed as an attribute using the dot notation. They are a list of layers, which can be confirmed by checking its type:

In [5]:
```
# Check the type of the model's layers
type(model.layers)
```

Out[5]:
```
list
```

In [6]:
```
# Print 5 first layers along with their names
for i in range(5):
    l = model.layers[i]
    print(f"Layer number {i}: \n{l} \nWith name: {l.name} \n")
```

```
print(f"Layer number {i}: \n{i} \nWith name: {i.name} \n")
```

```
Layer number 0:
<keras.engine.topology.InputLayer object at 0x7f836ab7f0f0>
With name: input_1

Layer number 1:
<keras.layers.convolutional.ZeroPadding2D object at 0x7f836ab7f358>
With name: zero_padding2d_1

Layer number 2:
<keras.layers.convolutional.Conv2D object at 0x7f836ab7f5f8>
With name: conv1/conv

Layer number 3:
<keras.layers.normalization.BatchNormalization object at 0x7f836a8f97f0>
With name: conv1/bn

Layer number 4:
<keras.layers.core.Activation object at 0x7f836a90d8d0>
With name: conv1/relu
```

Let's check how many layers our model has:

In [7]:
```
# Print number of layers in our model
print(f"The model has {len(model.layers)} layers")
```

```
The model has 428 layers
```

Our main goal is interpreting the representations which the neural net is creating for classifying our images. But as you can see this architecture has many layers.

Actually we are really interested in the representations that the convolutional layers produce because these are the layers that (hopefully) recognize concrete elements within the images. We are also interested in the "concatenate" layers because in our model's arquitecture they concatenate convolutional layers.

Let's check how many of those we have:

In [8]:
```
# Number of layers that are of type "Convolutional" or "Concatenate"
len([l for l in model.layers if ("conv" in str(type(l))) or ("Concatenate" in str(type(l)))])
```

Out[8]:
```
180
```

This number is still very big to try to interpret each one of these layers individually.

One characteristic of CNN's is that the earlier layers capture low-level features such as edges in an image while the deeper layers capture high-level concepts such as physical features of a "Cat".

Because of this **Grad-CAM usually focuses on the last layers, as they provide a better picture of what the network is paying attention to when classifying a particular class**. Let's grab the last concatenate layer of our model. Luckily Keras API makes this quite easy:

In [9]:
```
# Save the desired layer in a variable
layer = model.layers[424]

# Print layer
layer
```

Out[9]:
```
<keras.layers.merge.Concatenate at 0x7f82425a0d30>
```

This approach is not the best since we will need to know the exact index of the desired layer. Luckily we can use the `get_layer()` method in conjunction with the layer's name to get the same result.

Remember you can get the name from the information displayed earlier with the `summary()` method.

In [10]:
```
# Save the desired layer in a variable
layer = model.get_layer("conv5_block16_concat")

# Print layer
layer
```

Out[10]:
```
<keras.layers.merge.Concatenate at 0x7f82425a0d30>
```

Let's check what methods and attributes we have available when working with this layer:

In [11]:
```
# Printing out methods and attributes for Keras' layer
print(f"Keras' layers have the following methods and attributes: \n\n{dir(layer)}")
```

```
Keras' layers have the following methods and attributes:

['__call__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute_
_', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce_
_', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_add_inbound_nod
e', '_built', '_compute_elemwise_op_output_shape', '_get_node_attribute_at_index', '_inbound_nodes', '_initial_weights', '_los
ses', '_merge_function', '_node_key', '_non_trainable_weights', '_outbound_nodes', '_per_input_losses', '_per_input_updates',
'_reshape_required', '_trainable_weights', '_updates', 'add_loss', 'add_update', 'add_weight', 'assert_input_compatibility',
'axis', 'build', 'built', 'call', 'compute_mask', 'compute_output_shape', 'count_params', 'from_config', 'get_config', 'get_in
put_at', 'get_input_mask_at', 'get_input_shape_at', 'get_losses_for', 'get_output_at', 'get_output_mask_at', 'get_output_shape
_at', 'get_updates_for', 'get_weights', 'input', 'input_mask', 'input_shape', 'input_spec', 'losses', 'name', 'non_trainable_w
eights', 'output', 'output_mask', 'output_shape', 'set_weights', 'stateful', 'supports_masking', 'trainable', 'trainable_weigh
ts', 'updates', 'weights']
```

Since we want to know the representations which this layer is abstracting from the images we should be interested in the output from this layer. Luckily we have this attribute available:

In [12]:
```
# Print layer's output
layer.output
```

Out[12]:
```
<tf.Tensor 'conv5_block16_concat/concat:0' shape=(?, ?, ?, 1024) dtype=float32>
```

Do you notice something odd? The shape of this tensor is undefined for some dimensions. This is because this tensor is just a placeholder and it doesn't really contain information about the activations that occurred in this layer.

To compute the actual activation values given an input we will need to use a **Keras function**.

This function accepts lists of input and output placeholders and can be used with an actual input to compute the respective output of the layer associated to the placeholder for that given input.

Before jumping onto the Keras function we should rewind a little bit to get the placeholder tensor associated with the input. You can get this from the model's input:

In [13]:
```
# Print model's input tensor placeholder
model.input
```

Out[13]:
```
<tf.Tensor 'input_1:0' shape=(?, ?, ?, 3) dtype=float32>
```

We can see that this is a placeholder as well. Now let's instantiate our Keras function using Keras backend. Please be aware that this **function expects its arguments as lists or tuples**:

In [14]:
```
# Instantiate the function to compute the activations of the last convolutional layer
last_layer_activations_function = K.function([model.input], [layer.output])

# Print the Keras function
last_layer_activations_function
```

Out[14]:
```
<keras.backend.tensorflow_backend.Function at 0x7f8237c26b00>
```

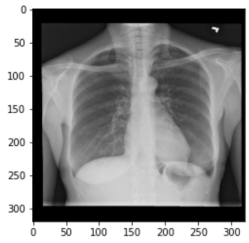
Let's test the functions for computing the last layer activation which we just defined on a particular image. Don't worry about the code to load the image, this has been taken care of for you. You should only care that an image ready to be processed will be saved in the x variable:

In [15]:
```
# Load dataframe that contains information about the dataset of images
df = pd.read_csv("nih_new/train-small.csv")

# Path to the actual image
im_path = 'nih_new/images-small/00000599_000.png'

# Load the image and save it to a variable
x = load_image(im_path, df, preprocess=False)

# Display the image
plt.imshow(x, cmap = 'gray')
plt.show()
```



We should normalize this image before going forward, this has also been taken care of:

In [16]:
```
# Calculate mean and standard deviation of a batch of images
mean, std = get_mean_std_per_batch(df)

# Normalize image
x = load_image_normalize(im_path, mean, std)
```

Now we have everything we need to compute the actual values of the last layer activations. In this case we should also **provide the input as a list or tuple**:

In [17]:
```
# Run the function on the image and save it in a variable
actual_activations = last_layer_activations_function([x])
```

An important intermediary step is to trim the batch dimension which can be done like this. This is necessary because we are applying Grad-CAM to a single image rather than to a batch of images:

In [18]:
```
# Remove batch dimension
actual_activations = actual_activations[0][0, :]
```

In [19]:
```
# Print shape of the activation array
print(f"Activations of last convolutional layer have shape: {actual_activations.shape}")

# Print activation array
actual_activations
```

```
Activations of last convolutional layer have shape: (10, 10, 1024)
```

Out[19]:
```
array([[[-0.2320239 ,  0.10742174, -0.10199872, ...,  0.19553502,
         -0.07577617,  0.21938165],
        [-0.3581124 , -0.49486023, -0.9013574 , ...,  0.28403032,
          -0.08808289,  0.3313018 ],
        [-0.26620686, -0.26146019, -0.9023777 , ...,  0.23974887,
          -0.07797471,  0.27408412],
        ...,
        [-0.37140965, -0.2965785 , -0.8896916 , ...,  0.13394466,
          -0.09150694,  0.17123136],
        [-0.25438887, -0.04947353, -0.51711893, ...,  0.28351784,
          -0.11772951,  0.32467234],
        [-0.23696655,  0.23388954, -0.08584651, ...,  0.14771259,
          -0.07197714,  0.21666431]],

       [[-0.3829313 ,  0.04334363, -0.4917316 , ...,  0.25094673,
          -0.11979439,  0.31754446],
        [-0.17019589, -0.4069005 , -0.26657537, ...,  0.43158036,
          -0.15095033,  0.4968133 ],
        [-0.478095  , -0.46936148, -0.42575347, ...,  0.2595103 ,
          -0.11638939,  0.3337812 ],
        ...,
        [-0.3411888 , -0.64699554, -0.570067  , ...,  0.15747964,
          -0.10875914,  0.1905012 ],
        [-0.53918445, -0.98736066, -0.55306005, ...,  0.32248756,
          -0.13720976,  0.32840085],
        [-0.55647004, -0.20790154,  0.00274089, ...,  0.18345954,
          -0.06889173,  0.21599565]],

       [[-0.46985203, -0.02180868, -0.43788403, ...,  0.27374545,
          -0.11379318,  0.24475136],
        [-1.2435349 , -1.0664959 , -0.30436754, ...,  0.4557925 ,
          -0.15071635,  0.35745603],
        [-0.87904364, -0.6215905 , -0.53652793, ...,  0.29657882,
          -0.09610435,  0.20594177],
        ...,
        [-1.3720858 ,  0.14263596,  0.23295045, ...,  0.06258567,
          -0.059364  ,  0.09076966],
        [-0.87398493, -0.04158668,  1.1696348 , ...,  0.1895878 ,
          -0.06298938,  0.15241994],
        [-0.735913  , -0.19385603,  0.9930296 , ...,  0.08033518,
          -0.02865089,  0.08128747]],

       ...,

       [[-0.5925746 , -0.00360541,  0.1453853 , ...,  0.36645842,
          -0.1544069 ,  0.48882958],
        [-0.3741458 ,  0.71299934, -0.03095169, ...,  0.41347706,
```

```
  -0.12635663,  0.7302184 ],
 [-2.0391881 , -1.2035868 , -0.78836703, ...,  0.19828713,
   0.17152329,  0.8838612 ],
 ...,
 [-2.0202935 , -0.81993145, -0.60169584, ...,  0.00893948,
   0.11304875,  1.4244986 ],
 [-1.627526  ,  0.10089205, -0.56618744, ...,  0.16328251,
  -0.07842733,  1.5084379 ],
 [-0.8808274 ,  0.07495729,  0.32282066, ..., -0.12680294,
   0.1021073 ,  1.0552356 ]],

[[-0.6009467 , -0.02361962, -0.4968306 , ...,  0.29006228,
  -0.15762725,  0.4679813 ],
 [-0.5199242 ,  0.1362013 , -0.45482838, ...,  0.44957054,
  -0.2384877 ,  0.6552417 ],
 [-0.44069824,  0.60020083, -1.5285702 , ...,  0.33973527,
  -0.15118313,  0.529661  ],
 ...,
 [-0.77362245, -0.09528783, -1.0183448 , ...,  0.0740128 ,
  -0.08222361,  0.7616997 ],
 [-0.26163074,  0.4153058 , -0.9229106 , ...,  0.2915144 ,
  -0.22311631,  0.8954488 ],
 [-0.49979827, -0.11817195, -0.59712785, ...,  0.04106264,
  -0.08934524,  0.7061914 ]],

[[-0.8855818 ,  0.21714076,  0.48969206, ...,  0.2639115 ,
  -0.11158875,  0.247319  ],
 [-0.9829037 , -0.16871876,  0.3662469 , ...,  0.3483049 ,
  -0.1248283 ,  0.3412089 ],
 [-0.64528453, -0.00385643, -0.2709455 , ...,  0.27218765,
  -0.09711117,  0.28448832],
 ...,
 [-0.5103816 ,  0.70147526, -0.06775373, ...,  0.16358954,
  -0.08347262,  0.24817598],
 [-0.5810188 ,  0.30591297,  0.22518802, ...,  0.2523684 ,
  -0.11769285,  0.30577222],
 [-0.60597765,  0.22969413,  0.5396775 , ...,  0.17829269,
  -0.05893847,  0.2545595 ]]], dtype=float32)
```

Looks like everything worked out nicely! This is all for this lecture notebook (Grad-CAM Part 1). In Part 2 we will see how to calculate the gradients of the model's output with respect to the activations in this layer. This is the "Grad" part of Grad-CAM.

Congratulations on finishing this lecture notebook! Hopefully you will now have a better understanding of how to leverage Keras's API power for computing activations in specific layers. Keep it up!