Visualisation Methods for Deep Learning

May 13, 2020

Introduction

In this project I will investigate and demonstrate methods for visualising inference drawn from Convolutional Neural Networks (CNNs). Three differing types of visualisation will be implemented: Class Activation Heatmaps (CAMs)[1], Occlusion Sensitivity Maps (OSMs) and Guided Backpropagation[7].

1 Background

CNNs are a blackbox approach to prediction whose decisions processes are difficult to understand; visualisation provides researchers with important model diagnostic insight not available through error-testing alone. Visualisations can show which features or regions of an image are being used in the decision making process of a network, enabling researchers to ensure a network is not incorrectly learning classes, for example distinguishing dogs and wolves by the presence of snow in the background[9] or the US army's fabled attempt to detect camouflaged tanks[10].

Not only can visualisations be used for diagnostics, they can be used as the basis for object-localisation and labelling. Object localisation has important practical uses: medical imaging software uses object-localisation and pixel labelling to highlight areas of images, for example potential lesions, that might be relevant to researchers[4]. Whilst the methods here will focus on diagnostic visualisation, the underlying functionality could be extended to provide object-localisation.

As there are a variety of different types of visualisation available for NNs, which particular method is most appropriate is often dictated by the domain or purpose of the model. For the three methods demonstrated in this project, their function is to display which areas of input images are discriminative in prediction, helping researchers understand and explain the model's decision making process. Here, input areas may be defined on a pixel-level or a wider region represented as a heatmap. Whilst the visualisations considered in this project are input-based, other methods exist that are feature-based (feature activation maps), display convolutional filters or maximise output by regularized optimisation[8].

2 Methods

2.1 Class Activation Maps

CAMs use global average pooling (GAP) layers to produce visualisations that show the discriminative areas of an input image for a given class as heatmaps. They highlight regions of an input image that activate feature maps most strongly for a given class, essentially showing which areas of the input contribute most to the CNN's evidence for the input belonging to the given class.

The method allows visualisation for any convolutional-GAP-fully connected layer series (if necessary the architecture can be adjusted such that a GAP layer follows the target), usually at the top of the network, by projecting back the weights of the fully connected layer for a given class onto the convolutional layer. The GAP layer outputs the spatial average of the feature maps, $F_k = \sum_{x,y} f_k(x, y)$, for each of the k convolutional filters in the target layer; the output of the fully connected layer (i.e. the pre-softmax network output if the layer is at the top of the network), S_c, is the weighted sum of these values:

$$S_c = \sum_k w_c^k F_k = \sum_{x,y} \sum_k w_c^k f_k(x, y) = \sum_{x,y} M_c(x, y) \tag{1}$$

where $M_c(x, y)$ is the CAM for class c. Thus the CAM for class c is the weighted sum of the feature maps in the convolutional layer.

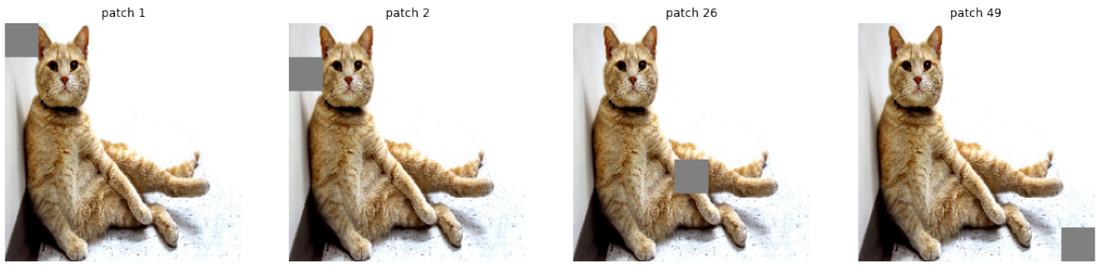
Further research has led to the introduction of Grad-CAM[6] which uses gradients from the target convolutional layer to compute the CAM. This method can be generalised to architectures without GAP layers. As the model used in this project, ResNet-152, includes a GAP layer we do not need to consider Grad-CAM.

2.2 Occlusion Sensitivity

OSMs also show discriminative regions of an input by sequentially occluding patches of the input and calculating the impact on the prediction confidence. The output is a grid of prediction confidences, with each cell value corresponding to the prediction for the image with that cell patched. Figure 1 demonstrates this patching scheme. The approach is simple and can help identify areas of the image that are both increase

and decrease confidence for a given class. The method allows scope to adjust the patching behaviour: if a more detailed, granular output is required one could use smaller patches or greater number of patches with an overlapping stride. In this sense the method is more adjustable than CAMs.

Figure 1: Batch of Patches



2.3 Guided Backpropagation

Guided backpropagation[7] is an example of a saliency mapping scheme and was developed as an alternative to the deconvnet approach developed by Zeiler and Fergus[2]. Saliency maps display which input pixels are important in determination of the output of a given layer and are useful in practice: variants of salient point detection schemes are used in anomaly detection and medical imaging, highlighting important parts of images to practitioners. By asking which pixels affect the output of a given layer most, we are effectively searching for the pixels for which neuron (de-)activation is most sensitive. These are the pixels corresponding to the largest neuron activation partial gradients with respect to the model inputs.

Backpropagation methods work by first making a forward pass up to a given layer for an input image, setting all neurons in the layer except the target to 0, and propagating back to the input to get an image reconstruction. Variants differ in the way they backpropagate through activation units: vanilla backpropagation maintains the forward behaviour of the units by blocking gradients based on the values of the forward pass; deconvnet resets and reverses the units by blocking gradients based on the gradient values as they backpropagate; whereas guided backpropagation combines both by blocking gradients based on the forward pass values and the gradient values. For given layer l and neuron i with activation f_i and gradient $R_i^l = \frac{\partial f^{\text{output}}}{\partial f_i^l}$:

$$\begin{aligned} \text{backpropagation:} \quad & R_i^l = [f_i^l > 0] \cdot R_i^{l+1} \\ \text{deconvnet:} \quad & R_i^l = [R_i^{l+1} > 0] \cdot R_i^{l+1} \\ \text{guided backpropagation:} \quad & R_i^l = [f_i^l > 0] \cdot [R_i^{l+1} > 0] \cdot R_i^{l+1} \end{aligned} \tag{2}$$

The idea of guided backpropagation is to block negative gradients from propagating back to the reconstruction. Negative gradients for intermediate level neurons contribute to decreasing the activation of the higher level neuron we are attempting to visualise and make reconstructed images less distinct.

3 Implementations

There are various packages available for CNN visualisation compatible with a number of deep learning libraries. For example, `tfexplain`[**?**] provides TensorFlow and Keras compatible functionalities, including CAMs, gradient backpropagation based methods (though not guided backpropagation), OSMs and Grad-CAM. Some of the code written for this project is inspired by their implementations.

Each of the three methods are used with a ResNet-152[3] model pre-trained on ImageNet, an online database of images divided into 1000 classes. The pre-built, pre-trained model is available through Keras. Using the pre-trained model allows demonstration of visualisations of accurate decisions without requiring lots of training. The methods are implemented using Keras and TensorFlow. To demonstrate the visualisations, some test images were selected, shown in figure 2.

3.1 CAM

The CAM method is relatively simple to implement. The target convolutional layer is extracted as `conv_layer`, and using this we build a Keras function mapping the model inputs to `conv_layer` outputs and model outputs (the latter is not strictly necessary and is simply for obtaining predictions):

```
K.function([model.input], [conv_layer.output, model.output])
```

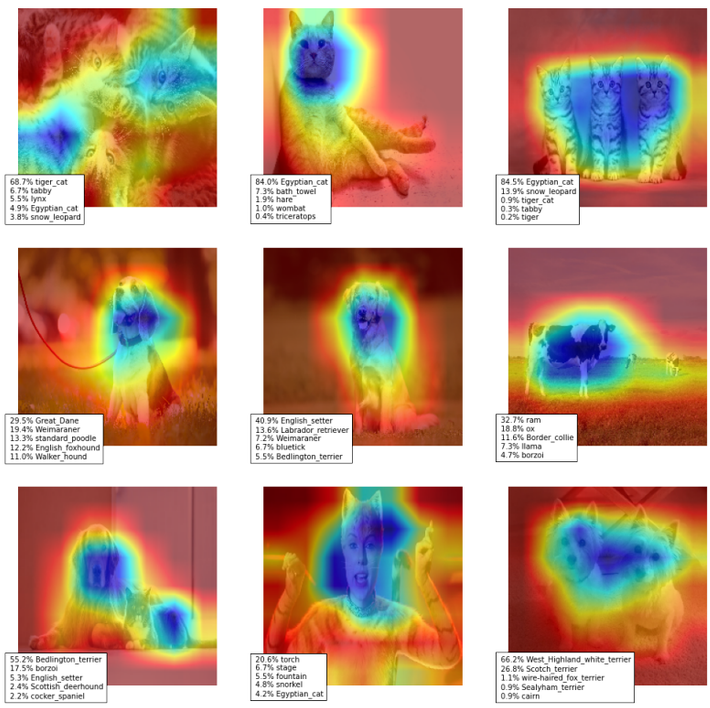
When supplied an input test image, this function returns the convolutional layer output, a tensor with shape `(7, 7, 2048)` corresponding to the 2048 7x7 filters, and model predictions, an array with shape `(1, 1000)`. From the final dense layer the weights `class_weights` are extracted as a tensor with shape `(2048, 1000)`. An empty CAM matrix, `cam`, is instantiated. Finally, we loop through the `class_weights` (indexed by the target class) and populate `cam` as the ponderated sum of the convolutional outputs:

Figure 2: Test Images

```
for i, w in enumerate(class_weights[:, target_class]):
    cam += w * conv_outputs[:, :, i]
```

To display, we superimpose `cam` on the input and plot. The generated plots have high values (blue) in areas that are most discriminative. Figure 3 is the output on the test images. The visualisation accurately highlights areas one would expect to be important in image classification, tracking animal faces where possible. It is worth noting the accuracy is limited by the granularity of the feature maps: the filters of the final convolutional layer in the ResNet-152 model only allow a 7x7 output grid. If more granular visualisation was desired, the architecture would have to be altered to increase the filter dimensions and the model retrained, making this approach relatively inflexible.

Figure 3: CAMs

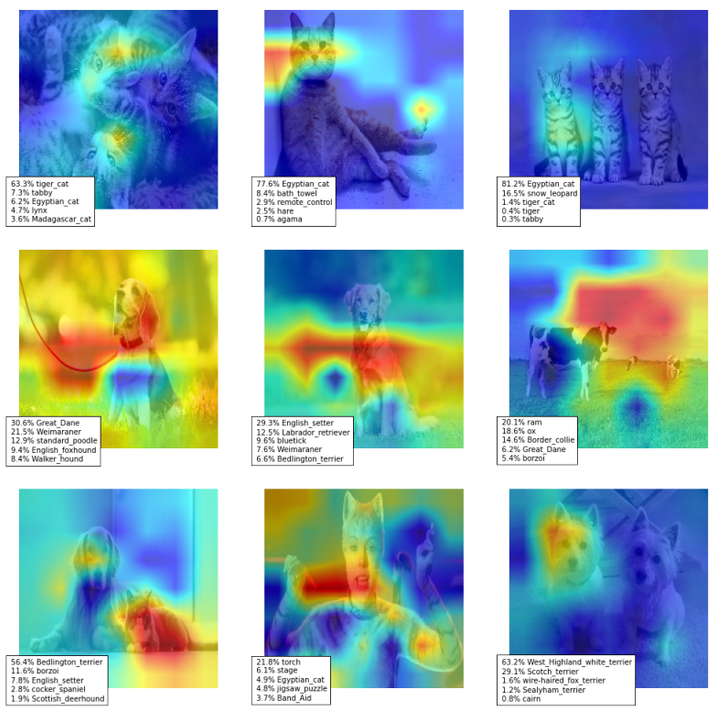


3.2 Occlusion Sensitivity

Occlusion Sensitivity is the most straightforward method to understand and implement. A batch of input images is generated with each identical except for an overlayed grey patch moving sequentially across the image, resulting in a batch of images length `n_patches * n_patches`, as in figure 1.

The batch is fed into the network and the predictions for a given class index are extracted. The map is then produced by reshaping the prediction array into a square array with shape `(n_patches, n_patches)` such that each entry is the prediction confidence of the input image with the corresponding area patched. The generated plots have low values (red) in regions that are most discriminative. Figure 4 is the output on the test images.
The generated visualisations do not accurately map the expected regions, only very approximately tracking the object locations in each image. This is particularly apparent in the test images with the least confident predictions. It is worth noting that in test images with no obvious features for classification, the model will often use the patch itself as a feature. Evidence of this can be seen in Figure 4 where the predictions for the bottom central panel have been updated from torch, stage, fountain, snorkel, Egyptian cat to torch, stage, Egyptian cat, jigsaw puzzle, band-aid.

Figure 4: OSMs



3.3 Guided Backpropagation

Guided Backpropagation is the most complicated method to implement, requiring multiple steps. Before backpropagating, the computational graph must be altered to "guide" the gradients as desired. To achieve this a custom layer, `GuidedBpgGate`, is inserted into the network after every ReLU occurrence. This layer is an identity on the forward pass, with a custom gradient that acts as a ReLU on backpropagation. This layer is defined as follows:

```
@tf.custom_gradient
def guided_bpg_gate(x):
    # op for GuidedBpgGate with custom guided gradients
    def grad(dy):
        return dy * tf.cast(dy > 0., x.dtype)
    return tf.identity(x), grad

class GuidedBpgGate(tf.keras.layers.Layer):
    # Custom layer that does nothing on a forward pass but blocks negative
    # gradients on a backwards pass.
    def __init__(self, **kwargs):
        super(GuidedBpgGate, self).__init__(**kwargs)
    def call(self, x):
        return guided_bpg_gate(x)
```

Inserting the layer into the network is not trivial: in order to update the graph correctly, care must be taken to ensure computation node inputs and outputs are directed appropriately. This is especially complicated in non-sequential models like the ResNet-152 used in this project where multiple nodes might point to the same node and vice versa (though the activations we replace are always single-input single-output). In order to coerce TensorFlow to recompile the graph, the model is saved to file and reloaded.

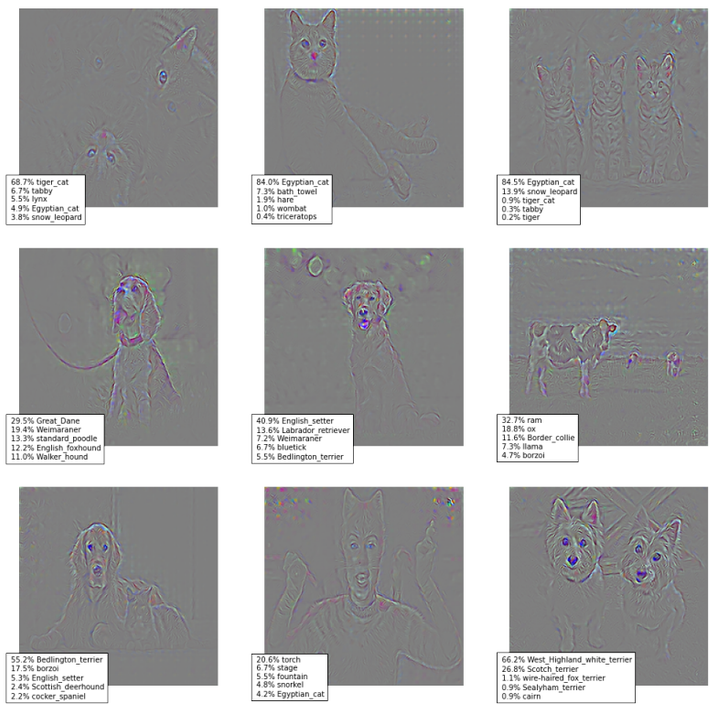
Once the altered model is defined, we backpropagate through the network using TensorFlow 2.x's `GradientTape` context manager. Automatic gradients are stored for calculations performed within its scope. The gradients are thus backpropagated via loss calculation:

```
with tf.GradientTape() as tape:
    tape.watch(inputs)
    preds = model(inputs)
    loss = tf.keras.losses.categorical_crossentropy(labels, preds)
# return back propagated gradients
return tape.gradient(loss, inputs)
```

Once correctly computed, the backpropagated guided gradients can be plotted; figure 5 shows the generated output for the test images.

The visualisations highlight the features of the input that are discriminative for the most likely class: eyes, noses, ears and the like. They show that the model typically learns animal classes by their facial features as a human would. For images where prediction is less certain the model, and thus the reconstructed image, is not able to pick out any distinctive features.

Figure 5: Guided Backpropagation



4 Discussion

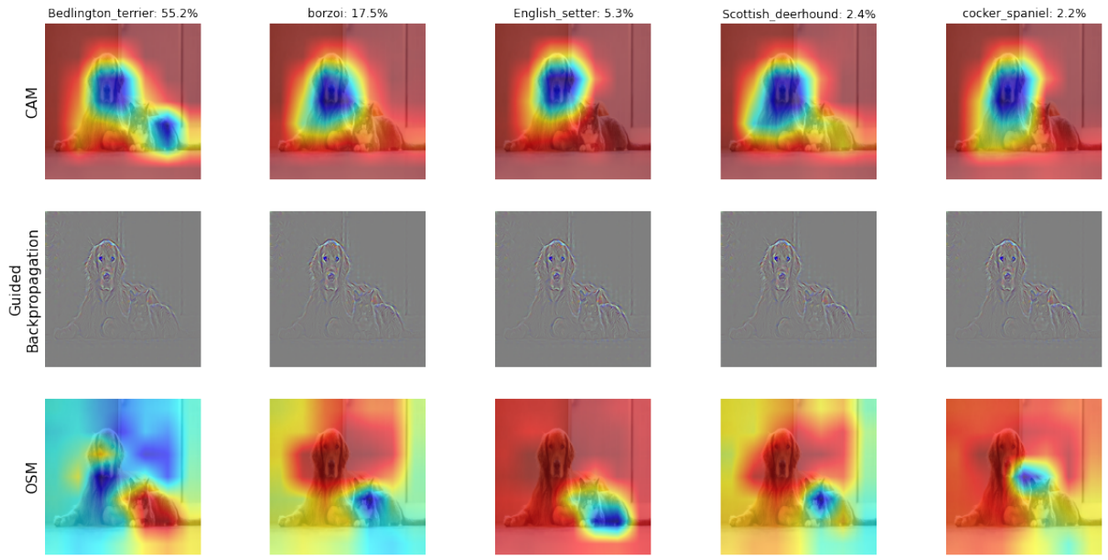
Figure 6 demonstrates some further visual analysis that may prove useful. Plotting the visualisations for a number of different classes helps shed insight on how the model distinguishes between classes. In the figure, the three methods are computed for the test image at each of the 5 top predicted classes. The CAMs show relatively stable visualisations, each highlighting a similar area. Interestingly, in the visualisation for the class with the highest confidence, the region of the image containing the cat is positively discriminative in the prediction. This is not true for the other classes' visualisations and perhaps gives insight into how the model has learnt Bedlington terriers (the dog in the image specifically does not look like a Bedlington terrier).

The OSMs are unstable and it is hard to determine much from them. They do however reaffirm the evidence that the model discriminates positively in favour of the cat's presence for the top predicted class, and negatively for the remaining classes.

The guided backpropagation visualisations are identical across the 5 classes. This is because the class and the backpropagated gradients are independent conditional on the target layer neurons; if the neurons are activated by the same pixels they will backpropagate the same gradients. Other less confident classes will result in different backpropagated gradients.

Overall, three approaches above could prove useful to practitioners, with each having benefits and drawbacks. CAMs and OSMs display similar information to each other and both point to which region of the image is deciding a NN's prediction. CAMs are a more recent approach and are typically more accurate and reliable (even with low confidence predictions), as is evident in this demonstration, though they are not trivial to explain conceptually. OSMs are on the other hand extremely easy to understand and implement. They can also easily be generalised to any type of CNN (including non-image networks). It is also possible to fine tune the granularity of the OSM by selecting the number of patches used. In contrast, CAMs are limited to the dimensions of the given convolutional layer (in the case of the final ResNet152 convolutional layer this is 7x7). A coarser grid limits the specificity of the approach: instead of being able to point to specific features it can only indicate larger regions.

Figure 6: Visualisations for Different Classes



Guided Backpropagation is relatively challenging to implement and has a non-trivial interpretation. It does however provide information not present in heatmaps: in highlighting discriminative pixels, the reconstructed images show in detail exactly which features of the image (e.g. eyes, stripes or ears) are used in the NN's prediction.

In summary, a practitioner interested in examining the performance of a model may choose to use a combination of the methods explored here. Visualisations provide an insightful, powerful model diagnostic beyond what is available through error testing alone, allowing researchers to gain a better understanding of a network's learning.

References

[1] Zhou et al., "Learning Deep Features for Discriminative Localization" *arXiv:1512.04150*, 2015.

[2] Zeiler, Fergus, "Visualizing and Understanding Convolutional Networks", *arXiv:1311.2901*, 2013.

[3] He, Zhang, Ren, Sun, "Deep Residual Learning for Image Recognition", *arXiv:1512.03385*, 2015.

[4] D. K. Iakovidis, S. V. Georgakopoulos, M. Vasilakakis, A. Koulaouzidis and V. P. Plagianakos, "Detecting and Locating Gastrointestinal Anomalies Using Deep Learning and Iterative Cluster Unification," *IEEE Transactions on Medical Imaging, vol. 37, no. 10, pp. 2196-2210*, Oct. 2018.

[5] M. Oquab, L. Bottou, I. Laptev and J. Sivic, "Is object localization for free? - Weakly-supervised learning with convolutional neural networks," *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, 2015, pp. 685-694*, 2015.

[6] Selvaraju, Cogswell, Das, Vedantam, Parikh, Batra, "Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization" *arXiv:1610.02391*, 2016.

[7] Springenberg et al., "Striving for Simplicity: The All Convolutional Net", *arXiv:1412.6806*, 2015.

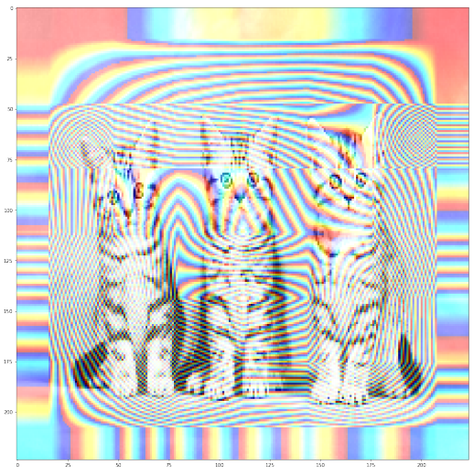
[8] Yosinski et al. "Understanding Neural Networks Through Deep Visualization", *arXiv:1506.06579*, 2015.

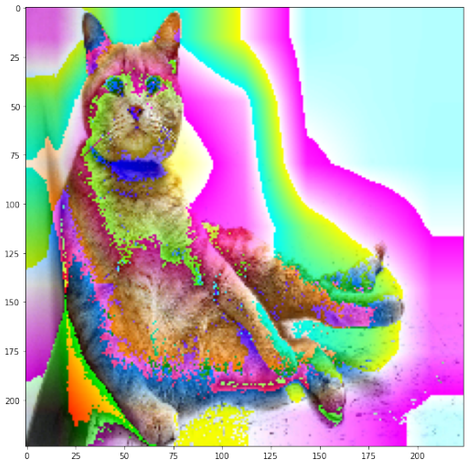
[9] Ribeiro, Singh, Guestrin, "'Why Should I Trust You?': Explaining the Predictions of Any Classifier", *arXiv:1602.04938*, 2016.

[10] *https://www.gwern.net/Tanks*

[11] *https://tf-explain.readthedocs.io/en/latest/#*

[12] *https://github.com/eclique/kerasgradcam/blob/master/grad_cam.py*





Appendix

Included in the appendix are two fun images that were accidentally produced during the process of completing this project. What is shown in these images is entirely unclear.