Sean Yue (A01228440  
Nov. 29/22

**Assignment 3 Write-up: Evaluating Image Classifier Models with Saliency Maps**

**Introduction:**

Convolutional Neural Networks (ConvNets or CNNs) have gained massive popularity in recent years, mostly being used to analyze visual imagery. Their applications are wide-ranging, from simple binary classifiers to complex models that are used in hospitals to identify tumors. Despite being so widely used, CNNs are still somewhat of a black box: it is not always clear “why” a model makes the prediction that it does. Without this information, how can researchers and developers be sure their models will work in the real world?

One way we can analyze how an image classifier model makes its predictions is by using tools like TruLens and Keras-vis. These analytical tools can help generate attention heatmaps for input images, showing us just where the model is looking when it makes a prediction. Additionally, tools like TruLens provides additional insights such as being able to see the most important feature map in each layer, giving us important information for evaluating our models.

**Method A: Trulens**

To setup your system to run a project utilizing these tools, run the following import code in a Jupyter Notebook or Google Colab cell.

1. # Here are all the imports that we'll need. Please comment out if you have them installed (not neccessary just for peace of mind).
2. **import** sys
3. !{sys.executable} **-**m pip install trulens
4. !{sys.executable} **-**m pip install torchvision
5. !{sys.executable} **-**m pip install matplotlib
6. !{sys.executable} **-**m pip install pandas
7. !{sys.executable} **-**m pip install scipy
8. !{sys.executable} **-**m pip install tensorflow
9. !{sys.executable} **-**m pip install torchsummary
10. !{sys.executable} **-**m pip install torch
11. !{sys.executable} **-**m pip install IPython
12. !{sys.executable} **-**m pip install numpy **--**upgrade
13. !{sys.executable} **-**m pip install Pillow

Next run the following import statement to import all of the relevant libraries into your Jupyter Notebook

1. # Import all the libraries we'll need
3. **import** json
4. **import** PIL
5. **import** requests
6. **import** scipy.misc
7. **import** torch
8. **import** torch.nn as nn
9. **import** torchvision
10. **import** matplotlib.pyplot as plt
11. **import** matplotlib.cm as cm
12. **import** numpy as np
13. **import** pandas as pd
14. **import** tensorflow as tf
15. **import** torchvision.utils as utils
16. **import** torchvision.models as models
17. **import** torchvision.transforms as transforms
18. **from** IPython.display **import** Image, display
19. **from** tensorflow **import** keras
20. **from** torchsummary **import** summary
21. **from** torch.autograd **import** Variable
22. **from** trulens.nn.attribution **import** InputAttribution
23. **from** trulens.nn.attribution **import** IntegratedGradients
24. **from** trulens.visualizations **import** ChannelMaskVisualizer
25. **from** trulens.visualizations **import** MaskVisualizer
26. **from** trulens.visualizations **import** HeatmapVisualizer
27. **from** trulens.nn.models **import** get\_model\_wrapper
28. **from** trulens.nn.attribution **import** InternalInfluence
29. **from** trulens.nn.distributions **import** PointDoi
30. **from** trulens.nn.quantities **import** ClassQoI, InternalChannelQoI, MaxClassQoI
31. **from** trulens.nn.slices **import** Cut, InputCut, OutputCut, Slice

This next step will display the figures inline in your notebook cell.

1. # Display matplotlib figures in the cell
2. **%**matplotlib inline

Now that the environment is setup, we’ll be importing a pre-trained VGG16 model from pyTorch. Just as a quick introduction, VGG16 is a CNN developed by the Visual Geometry Group from Oxford University. This model won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2014, and is widely considered one of the best image classifier models to currently exist. It is able to classify over 1000 objects with ~93% accuracy. The 16 in VGG16 stands for 16 weighted layers (13 Convolution layers and 3 Dense layers).

This next block of code imports the pre-trained model and creates a wrapped model with TruLens, which we will use later for analysis. It will also print out a quick summary of the model, showing layers and output shapes.

1. # Create a Pytorch VGG16 model
2. pytorch\_model **=** models.vgg16(pretrained**=**True)
3. device **=** 'cpu'
5. # Produce a wrapped model from the pytorch model for Trulens.
6. wrapped\_model **=** get\_model\_wrapper(pytorch\_model, input\_shape**=**(3,224,224), device**=**device)
7. print(summary(pytorch\_model, (3, 224, 224)))

In the following block of code, we’ll use the Pillow library to open an image from a URL on your device and then make transformations on that image to an appropriate size in order to pass it to the VGG16 model. In Jupyter, it will also show a Matplotlib diagram of the original image.

1. url **=** "beaglebike.jpg"
2. #url = "beagle\_blackbike.jpg"
3. #url = "cat\_on\_car.jpg"
4. #url = "bear\_in\_kitchen.png"
6. with PIL.Image.open(url) as img:
7. x **=** np.array(img.resize((224,224), PIL.Image.ANTIALIAS))[np.newaxis]
9. # Pre-process uploaded image
10. normalize **=** transforms.Compose([
11. transforms.ToTensor(), # convert to [0, 1]
13. # these are the means and stddevs of ImageNet (calculated from their image library)
14. transforms.Normalize(
15. mean**=**[0.485, 0.456, 0.406],
16. std**=**[0.229, 0.224, 0.225])])
17. x\_pp **=** np.array(normalize(x[0])).transpose(1, 2, 0)[np.newaxis]
19. # Transpose to [\*, C, H, W] for PyTorch convention.
20. x **=** x.transpose(0, 3, 1, 2)
21. x\_pp **=** x\_pp.transpose(0, 3, 1, 2)
22. plt.imshow(img)

Now that the model and input image are ready, we can make predictions. Before we pass the image through the model, we use the following code to collect some useful information along the way. This code will collect all the convolution layers.

1. # we will save the conv layer weights in this list
2. model\_weights **=**[]
4. #we will save the 13 conv layers in this list
5. conv\_layers **=** []
7. # get all the model children as list
8. model\_children1 **=** list(pytorch\_model.children())
9. model\_children **=** model\_children1[0]
11. #counter to keep count of the conv layers
12. counter **=** 0
14. #append all the conv layers and their weights to the list
15. **for** i **in** range(len(model\_children)):
16. **if** type(model\_children[i]) **==** nn.Conv2d:
17. counter**+=**1
18. model\_weights.append(model\_children[i].weight)
19. conv\_layers.append(model\_children[i])
20. **elif** type(model\_children[i]) **==** nn.Sequential:
21. **for** j **in** range(len(model\_children[i])):
22. **for** child **in** model\_children[i][j].children():
23. **if** type(child) **==** nn.Conv2d:
24. counter**+=**1
25. model\_weights.append(child.weight)
26. conv\_layers.append(child)
27. print(f"Total convolution layers: {counter}")

Then we’ll run the model, make transformations on the input image. This will print out the image shape before being passed to our model.

1. # run the model with cuda if available, else use cpu
2. device **=** torch.device("cuda" **if** torch.cuda.is\_available() **else** "cpu")
3. x\_model **=** pytorch\_model.to(device)
5. # show how transformations change image shape
6. transform **=** transforms.Compose([
7. transforms.Resize((224, 224)),
8. transforms.ToTensor(),
9. transforms.Normalize(mean**=**[0.485, 0.456, 0.406],
10. std**=**[0.229, 0.224, 0.225])])
11. image **=** transform(img)
12. print(f"Image shape before: {image.shape}")
13. image **=** image.unsqueeze(0)
14. print(f"Image shape after: {image.shape}")
15. image **=** image.to(device)

Now, just for a quick double-check, we can print out the names of all of the convolution layers we have collected with the code below.

1. outputs **=** []
2. names **=** []
3. **for** layer **in** conv\_layers[0:]:
4. image **=** layer(image)
5. outputs.append(image)
6. names.append(str(layer))
8. joined **=** list(zip(names,outputs))
9. **for** i **in** joined:
10. print(f"Name: {i[0]}; Shape:{i[1].shape}")

The code below will use the data we set up earlier to show how the image is processed by VGG16 at each convolution layer. You can see how edges are being collected, and how the Gaussian blur is applied.

1. processed **=** []
2. **for** feature\_map **in** outputs:
3. feature\_map **=** feature\_map.squeeze(0)
4. gray\_scale **=** torch.sum(feature\_map,0)
5. gray\_scale **=** gray\_scale **/** feature\_map.shape[0]
6. processed.append(gray\_scale.data.cpu().numpy())
8. fig **=** plt.figure(figsize**=**(30, 50))
9. **for** i **in** range(len(processed)):
10. a **=** fig.add\_subplot(5, 4, i**+**1)
11. imgplot **=** plt.imshow(processed[i])
12. a.axis("off")
13. a.set\_title(names[i].split('(')[0], fontsize**=**30)
14. plt.savefig(str('feature\_maps.jpg'), bbox\_inches**=**'tight')

Now let’s use the model to print out prediction scores for all of the labels in the Imagenet classification library.

1. # Make predictions using pytorch model and write them to output
2. with torch.no\_grad():
3. output **=** pytorch\_model(torch.from\_numpy(x\_pp).to('cpu')).cpu().numpy()
4. df **=** pd.DataFrame(output)
5. df **=** df.transpose()
6. df

The data provided by the code is no use if we don’t put a name to the label index. This code below will use a JSON which I will upload, to match the index to the predicted object.

1. # Open classifications file
2. with open('imagenet\_class\_index.json') as file:
3. classes\_index **=** json.load(file)

6. # change indexes to labels
7. labels\_from\_index **=** [classes\_index[str(x)][1] **for** x **in** range(len(classes\_index))]
8. top\_labels **=** [
9. (j, labels\_from\_index[j], output[0][j])
10. **for** j **in** np.argsort(output[0])[::**-**1][:10]]
12. # Top 10 predictions of image by pytorch VGG16
13. df **=** pd.DataFrame(top\_labels, columns**=**["Index", "Name", "Score"])
14. df

Now let’s use the TruLens library to analyze the predictions. We’ll start with the MaskVisualizer which highlights pixels which the model paid the most attention to when making the predictions. This visualizer takes in a blur and threshold value. The former helps users see a more generalized area of interest, the latter selects for the areas of most interest.

1. **from** trulens.nn.attribution **import** InputAttribution
2. **from** trulens.nn.attribution **import** IntegratedGradients
3. **from** trulens.visualizations **import** MaskVisualizer
5. # Saliency maps in Trulens are implemented by the InputAttribution class
6. # It takes a ModelWrapper object and from the attributions class, we can get several visualizations
8. # The first is this MaskVisualizer, which takes a blur and threshold argument and uses those to overlay
9. # a mask over the image revealing the top threshold precentage of pixels by attribution
11. infl **=** InputAttribution(wrapped\_model)
12. attrs\_input **=** infl.attributions(x\_pp)
14. masked\_image **=** MaskVisualizer(blur**=**10, threshold**=**0.95)(attrs\_input, x)

A similar insight can be made with the IntegradeGradients work flow from TruLens, which computes the gradient of the model’s prediction output to its input features.

1. # Another cool visualization feature from Trulens is the IntegratedGradients workflow
2. # It obtains attributions through Integrated Gradients which computes the gradient of the
3. # model’s prediction output to its input features
5. infl **=** IntegratedGradients(wrapped\_model, resolution**=**10)
6. attrs\_input **=** infl.attributions(x\_pp)
7. masked\_image **=** MaskVisualizer(blur**=**10, threshold**=**0.95)(attrs\_input, x)

And one final interesting visualization is the HeatmapVisualizer which shows hotspots for the model. They are generated with the code below.

1. # One final important feature is the heatmap visualization which most clearly shows
2. # you where the model is looking within an image
4. infl **=** IntegratedGradients(wrapped\_model, resolution**=**10)
5. attrs\_input **=** infl.attributions(x\_pp)
6. masked\_image **=** HeatmapVisualizer(blur**=**10)(attrs\_input, x)

The code below shows TruLens’ most important feature which is to be able to calculate attributions for individual neurons within a model. TruLens can tell us which feature maps in each layer are critical for model decision making.

1. **from** trulens.nn.attribution **import** InternalInfluence
2. **from** trulens.nn.distributions **import** PointDoi
3. **from** trulens.nn.quantities **import** ClassQoI, InternalChannelQoI, MaxClassQoI
4. **from** trulens.nn.slices **import** Cut, InputCut, OutputCut, Slice
6. # The most important feature of Trulens however is being able to calculate attributions for individual neurons
7. # within your model. ie. Trulens can tell you which Feature Maps in each layer are critical in model decision making
9. # Trulens does this with the InternalInfluence object which takes a Trulens ModelWrapper
10. # it requires a Slice, QoI (quantity of interest), and DoI (distribution of interest)
12. # Slice defines a layer to use for internal attributions
13. # the QoI defines the model behavior we want to explain using attributions, it is the output of some layer
14. # the DoI specifies points surrounding each record for faithful attribution calculations
16. # Define the influence measure.
17. infl **=** InternalInfluence(
18. wrapped\_model,
19. Slice(Cut('features\_28'), OutputCut()),
20. MaxClassQoI(),
21. PointDoi())
23. attrs\_internal **=** infl.attributions(x\_pp).sum(axis**=**(2,3))

This following code will show us the top feature map in features\_28, model at that layer.

1. # Define the influence measure.
2. infl **=** InternalInfluence(wrapped\_model, 'features\_28', 'max', 'point')
4. # Get the attributions for the internal neurons at layer 'features\_28'. Because
5. # layer 'features\_28' contains 2D feature maps, we take the sum over the width
6. # and height of the feature maps to obtain a single attribution for each feature
7. # map.
8. attrs\_internal **=** infl.attributions(x\_pp).sum(axis**=**(2,3))
9. print(f"Number of neurons in layer features\_28: {len(attrs\_internal[0])}")
11. top\_feature\_map **=** int(attrs\_internal[0].argmax())
13. print('Top feature map:', top\_feature\_map)

And this code will show the image with an attention mask for that layer:

1. # Since feature maps represent learned features we cant easily interepret them
2. # But Trulens can use a second set of attributions to find input features that are most important
3. # in defining THIS particular feature map
5. masked\_image **=** ChannelMaskVisualizer(
6. wrapped\_model,
7. 'features\_28',
8. top\_feature\_map,
9. blur**=**10,
10. threshold**=**0.95)(x, x\_pp)
11. plt.axis('off')
12. plt.imshow(masked\_image[0].transpose((1,2,0)))

**Method B: Keras-vis**

The steps below will show how to implement the same heatmap using Keras-vis. We’ll start by importing the model and importing preprocessing and decoding functions from Keras.

1. # Create the same model with Keras
2. model\_builder **=** keras.applications.vgg16.VGG16
3. img\_size **=** (224, 224)
4. preprocess\_input **=** keras.applications.vgg16.preprocess\_input
5. decode\_predictions **=** keras.applications.vgg16.decode\_predictions
6. last\_conv\_layer\_name **=** "block5\_conv3"
8. # Make model
9. keras\_model **=** model\_builder(weights**=**"imagenet")
10. print(keras\_model.summary())

Let’s generate the prediction scores using Keras’ version of VGG16

1. # The Heatmap process in Keras-Vis
2. # Trulens basically expands on the work done by keras-vis already, here is a saliency map example generated by keras-vis
4. # Remove last layer's softmax
5. keras\_model.layers[**-**1].activation **=** None
7. img **=** keras.preprocessing.image.load\_img(url, target\_size**=**img\_size)
8. plt.imshow(img)
10. # `array` is a float32 Numpy array of shape (299, 299, 3)
11. array **=** keras.preprocessing.image.img\_to\_array(img)
13. # add a dimension to transform our array into a "batch"
14. # of size (1, 299, 299, 3)
15. img\_array **=** np.expand\_dims(array, axis**=**0)
17. # Print what the top predicted class is
18. preds **=** keras\_model.predict(img\_array)
19. predicted\_string **=** decode\_predictions(preds, top**=**10)[0]
20. data\_preds **=** pd.DataFrame(predicted\_string, columns**=**["ID", "Prediciton", "Score"])
21. data\_preds

Next, we’ll pass the image to the model and generate a saliency map at the final convolution layer. This will give us a general idea of where the model was focusing on in processing the image.

1. # First, we create a model that maps the input image to the activations
2. # of the last conv layer as well as the output predictions
3. grad\_model **=** tf.keras.models.Model(
4. [keras\_model.inputs], [keras\_model.get\_layer(last\_conv\_layer\_name).output, keras\_model.output]
5. )
7. pred\_index**=**None
9. # Then, we compute the gradient of the top predicted class for our input image
10. # with respect to the activations of the last conv layer
11. with tf.GradientTape() as tape:
12. last\_conv\_layer\_output, preds **=** grad\_model(img\_array)
13. **if** pred\_index **is** None:
14. pred\_index **=** tf.argmax(preds[0])
15. class\_channel **=** preds[:, pred\_index]
17. # This is the gradient of the output neuron (top predicted or chosen)
18. # with regard to the output feature map of the last conv layer
19. grads **=** tape.gradient(class\_channel, last\_conv\_layer\_output)
21. # This is a vector where each entry is the mean intensity of the gradient
22. # over a specific feature map channel
23. pooled\_grads **=** tf.reduce\_mean(grads, axis**=**(0, 1, 2))
25. # We multiply each channel in the feature map array
26. # by "how important this channel is" with regard to the top predicted class
27. # then sum all the channels to obtain the heatmap class activation
28. last\_conv\_layer\_output **=** last\_conv\_layer\_output[0]
29. heatmap **=** last\_conv\_layer\_output @ pooled\_grads[..., tf.newaxis]
30. heatmap **=** tf.squeeze(heatmap)
32. # For visualization purpose, we will also normalize the heatmap between 0 & 1
33. heatmap **=** tf.maximum(heatmap, 0) **/** tf.math.reduce\_max(heatmap)
35. heatmap **=** heatmap.numpy()
36. plt.matshow(heatmap)

Finally, we’ll overlay that image back on to the original and display it to the Jupyter Notebook.

1. img **=** keras.preprocessing.image.load\_img(url)
2. img **=** keras.preprocessing.image.img\_to\_array(img)
4. # Rescale heatmap to a range 0-255
5. heatmap **=** np.uint8(255 **\*** heatmap)
7. # Use jet colormap to colorize heatmap
8. jet **=** cm.get\_cmap("jet")
10. # Use RGB values of the colormap
11. jet\_colors **=** jet(np.arange(256))[:, :3]
12. jet\_heatmap **=** jet\_colors[heatmap]
14. # Create an image with RGB colorized heatmap
15. jet\_heatmap **=** keras.preprocessing.image.array\_to\_img(jet\_heatmap)
16. jet\_heatmap **=** jet\_heatmap.resize((img.shape[1], img.shape[0]))
17. jet\_heatmap **=** keras.preprocessing.image.img\_to\_array(jet\_heatmap)
19. # Superimpose the heatmap on original image
20. superimposed\_img **=** jet\_heatmap **\*** 0.4 **+** img
21. superimposed\_img **=** keras.preprocessing.image.array\_to\_img(superimposed\_img)
23. plt.imshow(superimposed\_img)