Homework #2

CSEP 590B: Explainable AI
Prof. Su-In Lee
Due: 5/18/22 11:59 PM

1 Feature attributions and metrics (10 points)

(a) [4 points] Briefly describe **Grad** × **Input**, **SmoothGrad**, and **Integrated Gradients**, and how each method uses model gradients.

(b) [3 points] In contrast to the other gradient-based methods, **GradCAM** does not compute gradients with respect to the input image. Describe how **GradCAM** calculates gradients differently and how they are used to explain the original image.

(c) [3 points] Two of the main types of XAI evaluation metrics are ground truth comparisons and ablation metrics. Which of the two is more suitable to testing if an explanation identifies important features for a specific model?

Preliminaries

The remaining questions focus on feature attribution methods for computer vision models. As a first step, please install the following packages into your local Python environment, preferably using a recent version of Python 3:

```
pip install shap
pip install torch
pip install torchvision
pip install matplotlib
```

We'll use images available in the SHAP package, which can be loaded as follows:

```
import shap

display_images = shap.datasets.imagenet50()[0].astype('uint8')  # shape = (50, 224, 224, 3)
```

We'll use a ResNet-18 model pre-trained on the ImageNet dataset, which you can download using PyTorch:

```
from torchvision import models

model = models.resnet18(pretrained=True)
model = model.eval()  # turns off training mode for batch norm
```

Deep neural networks require input data in a particular format, and you can use the following pre-processing code:

```
import torchvision.transforms as transforms

# Image pre-processing, expects single image of size (224, 224, 3)
model_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
])
```

Putting this together, you can run the model on all 50 images as follows:

```
# Apply pre-processing and make predictions
images = torch.stack([model_transforms(img) for img in display_images])
pred = model(images).softmax(dim=1)
```

2 Occlusion (30 points)

Occlusion is a removal-based method that measures the impact of removing features from the input. The name comes from from the fact that pixels are typically occluded by setting them to zero (black). Here, we will implement **Occlusion** from scratch.

(a) [5 points] Plot the first three images in the dataset. Generate the plot before applying any pre-processing steps, and you can use the following starter code:

```
import matplotlib.pyplot as plt

# Generate plot
plt.figure()
plt.imshow(display_image)
plt.show()
```

(b) [5 points] Run the first three images through the model and find the predicted classes. Check here to see what these classes mean. **Hint:** PyTorch supports many of the same operations as numpy, including argmax.

(c) [10 points] Write a function with the following signature to generate feature importance values:

```
def occlusion(imgs, model, target_labels, baseline, superpixel_size=8):
    '''
    Args:
      imgs: torch.Tensor of pre-processed images, size = (batch, 3, 224, 224)
      model: PyTorch classifier
      target_labels: torch.Tensor of classes for each image, size = (batch,)
      baseline: baseline value for occluded features
      superpixel_size: width/height of superpixels

    Returns:
      importance: occlusion scores, size = (batch, 224, 224)
    '''
    pass
```

For the baseline, you can use the following baseline image to replace with zeros:

```
# Generate black image, then apply pre-processing
baseline = model_transforms(np.zeros((224, 224, 3), np.uint8))
```

Test your function by running it on the first three images. Use the predicted labels for each image, the zeros baseline, and superpixels of size 8×8. The following starter code shows how to display important regions in red and unimportant regions in blue:

```
# Generate plot
plt.figure()
m = single_importance.abs().max()
plt.imshow(single_importance, vmin=-m, vmax=m, cmap='seismic')  # specify min/max value
plt.show()
```

(d) [5 points] For the first of the three images, compare the results when using superpixels of size 4×4, 8×8 and 16×16. Plot the results side-by-side.

(e) [5 points] As an alternative to replacing with zeros, we can use a blurred version of the image as a baseline. For the first of the three images, compare the occlusion results when using several blurring strengths, using 8×8 superpixels. Plot the results together, including the different blurred versions of the image. **Hint:** several Python packages offer functions to blur images, see here for example.

3 Gradient-based explanations (30 points)

Several explanation methods are based on gradients with respect to the input image. These are often faster than removal-based approaches, and they are widely used to explain deep models. In this problem, we'll implement several gradient-based methods from scratch.

(a) [5 points] Implement the **Vanilla Gradients** method. It should be a function with the following signature:

```
def vanilla_gradients(imgs, model, target_labels):
    '''
    Args:
      imgs: torch.Tensor of pre-processed images, size = (batch, 3, 224, 224)
      model: PyTorch classifier
      target_labels: torch.Tensor of classes for each image, size = (batch,)

    Returns:
      saliency: tensor of saliency values, shape = (batch, 224, 224)
    '''
    pass
```

In your function, take the absolute value and sum across the channels before returning the result. Plot the output for the first three images in the dataset. **Hint:** use the following starter code to compute input gradients:

```
def calculate_gradients(imgs, model, target_labels):
    '''
    Args:
      imgs: torch.Tensor of pre-processed images, size = (batch, 3, 224, 224)
      model: PyTorch classifier
      target_labels: torch.Tensor of classes for each image, size = (batch,)

    Returns:
      gradients: gradients for the target class, shape = (batch, 3, 224, 224)
    '''
    # Prepare for model.
    imgs = imgs.clone()
    imgs.requires_grad = True

    # Make prediction.
    output = model(imgs).softmax(dim=1)

    # Sum outputs for target classes.
    mask = torch.zeros(output.shape)
    for i, target in enumerate(target_labels):
        mask[i, target] = 1
    backprop_output = torch.sum(output * mask)

    # Calculate gradients.
    model.zero_grad()
    backprop_output.backward()

    # Convert gradients to numpy.
    gradients = imgs.grad.detach()
    return gradients
```

The following starter code shows how to properly scale the saliency map in a plot:

```
# Generate plot
plt.figure()
plt.imshow(vanilla, vmin=0, vmax=vanilla.max())  # specify min/max value
plt.show()
```

(b) [10 points] Implement **SmoothGrad**, which adds Gaussian noise to the input and averages the gradients. It should be a function with the following signature:

```
def smoothgrad(imgs, model, target_labels, samples=50, sigma=0.1):
    '''
    Args:
```

```
    imgs: torch.Tensor of pre-processed images, size = (batch, 3, 224, 224)
    model: PyTorch classifier
    target_labels: torch.Tensor of classes for each image, size = (batch,)
    samples: number of random noise samples
    sigma: scale for random noise

  Returns:
    saliency: tensor of saliency values, shape = (batch, 224, 224)
  '''
  pass
```

In your function, you can take the absolute value either before or after averaging (whichever looks better), and sum across channels before returning the result. Plot the output for the first three images in the dataset.

(c) [10 points] Implement **Integrated Gradients**, which averages the gradient along a path and multiplies it by the input minus the baseline. It should be a function with the following signature:

```
def integrated_gradients(imgs, model, target_labels, baseline, steps=50):
  '''
  Args:
    imgs: torch.Tensor of pre-processed images, size = (batch, 3, 224, 224)
    model: PyTorch classifier
    target_labels: torch.Tensor of classes for each image, size = (batch,)
    baseline: baseline value for held-out features
    steps: number of steps along path to baseline

  Returns:
    saliency: tensor of saliency values, shape = (batch, 224, 224)
  '''
  pass
```

In your function, for simplicity, take the absolute value and sum across the channels. Plot the results for the first three images in the dataset, using the same baseline provided for Problem 2(c).

(d) [5 points] Run your function from part (c) again with the first three images, this time using the following baseline value:

```
# Generate white image, then apply pre-processing
baseline = model_transforms(255 * np.ones((224, 224, 3), np.uint8))
```

4 Ablation metrics (30 points)

Ablation metrics are those that evaluate explanations by probing a model with perturbed inputs. They typically work by removing different feature subsets and verifying whether the predictions change as the importance values suggest they should. In this problem we'll implement two popular ablation metrics, **Insertion** and **Deletion**. We'll use these metrics to test two methods implemented in the previous problems.

(a) [5 points] First, prepare the explanations that we'll evaluate using the metrics. Use the first ten images from the dataset and generate feature importance values using **Occlusion** (with 8×8 superpixels) and **SmoothGrad** (using 50 samples and a noise level of your choice). In addition, include a **Random** baseline: generate a saliency map for each image that is Gaussian noise. Plot the saliency maps side-by-side for one of the images.

(b) [5 points] To reduce the number of predictions required for the metrics, we'll ensure that all explanations correspond to 8×8 superpixels. The **Occlusion** (8×8) explanation does this already, but **SmoothGrad** and **Random** do not. Reduce their granularity by summing the importance within each non-overlapping 8×8 superpixel. Plot the new saliency maps for the same image. **Hint:** consider using PyTorch's AvgPool2d operation.

(c) [5 points] Write a function to generate an array of prediction probabilities as features are inserted in order of most to least important (**Insertion**). Rather than inserting individual pixels, insert 8×8 superpixels. The function should have the following signature:

```
def insertion(img, model, importance, target_label, baseline):
    '''
    Args:
      img: image to ablate, size = (1, 3, 244, 244)
      model: PyTorch classifier
      importance: feature importance values, size = (1, 28, 28)
      target_label: index of target class
      baseline: baseline value for held-out features

    Returns:
      curve: array of prediction probabilities after each step
      num_feats: array of number of features after each step
    '''
    pass
```

As a baseline value, use the zeros baseline from Problem 2(c). Plot the curve for a single image and just the **Occlusion** explanation, and calculate the area under the curve using the trapezoidal rule.

(d) [5 points] Write a function to generate an array of prediction probabilities as features are deleted in order of most to least important (**Deletion**). Delete 8×8 superpixels rather than individual pixels, similar to part (c). The function should have the following signature:

```
def deletion(img, model, importance, target_label, baseline):
    '''
    Args:
      img: image to ablate, size = (1, 3, 244, 244)
      model: PyTorch classifier
      importance: feature importance values, size = (1, 28, 28)
      target_label: index of target class
      baseline: baseline value for held-out features

    Returns:
      curve: array of prediction probabilities after each step
      num_feats: array of number of features after each step
    '''
    pass
```

Again, plot the curve for a single image and just the **Occlusion** explanation, and calculate the area under the curve using the trapezoidal rule.

(e) [5 points] Generate insertion curves for ten images and for all three explanation methods (**Occlusion**, **SmoothGrad**, **Random**). Plot the average curve for each explanation method, where the y-axis represents the average prediction for the target class at a given number of features (x-axis). Calculate the average area under the insertion curve for each method.

(f) [5 points] Repeat part (e), but this time with deletion curves.