Captum

A model interpretability library

Captum : Characteristics

MULTIMODAL

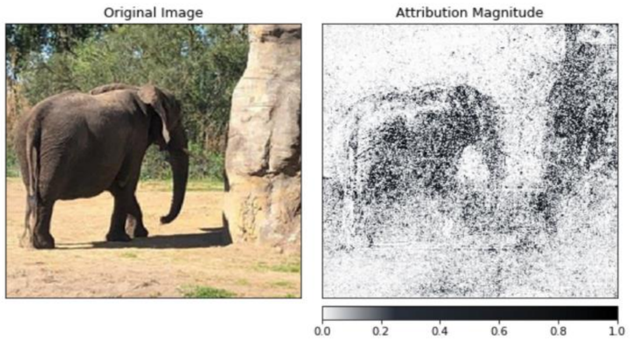
EXTENSIBLE

```
class MyAttribution(Attribution):

    def attribute(self, input, ...):
        attributions = self._compute_attrs(input, ... )
        # <Add any logic necessary for attribution>
        return attributions
```

EASY TO USE

Visualize_image_attr(attr_algo. Attribute(input),…)

Target Label	Predicted Label	Attribution Label	Attribution Score	Word Importance
elephant	elephant (0.55)	elephant	8.21	what is on the picture



What Does The Captum Library Offer?

Attribute algorithms to interpret:

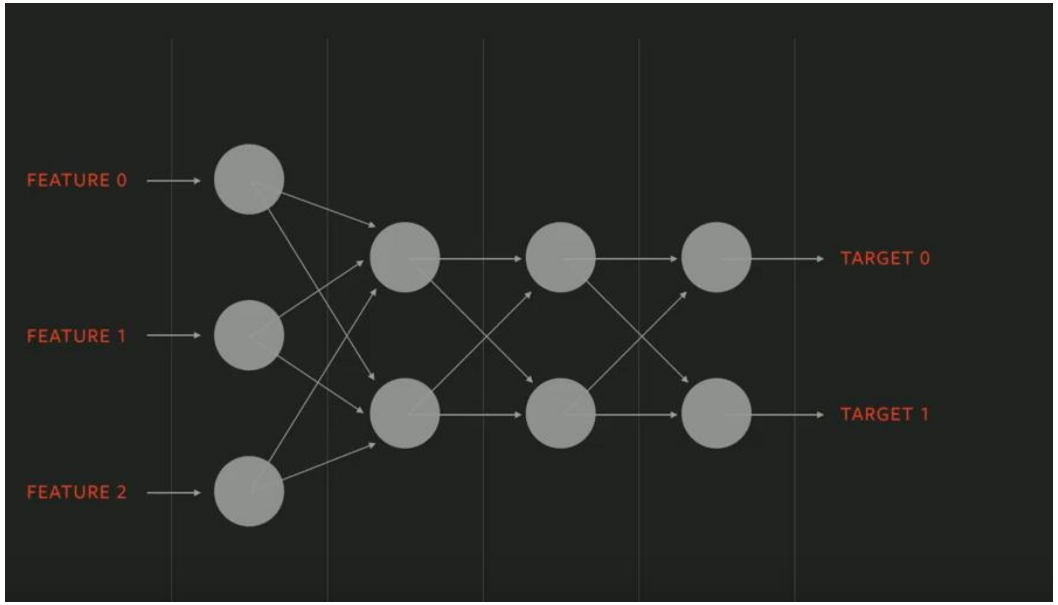
- Output predictions with respect to inputs
- Output predictions with respect to layers
- Neurons with respect to inputs

Possible to extend with Perturbation based Algorithms

Attribution

Toy Model

```python
class ToyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.lin1 = nn.Linear(3, 3)
        self.relu = nn.ReLU()
        self.lin2 = nn.Linear(3, 2)

        # initialize weights and biases
        self.lin1.weight = nn.Parameter(torch.arange(-4.0, 5.0).view(3, 3))
        self.lin1.bias = nn.Parameter(torch.zeros(1,3))
        self.lin2.weight = nn.Parameter(torch.arange(-3.0, 3.0).view(2, 3))
        self.lin2.bias = nn.Parameter(torch.ones(1,2))

    def forward(self, input):
        return self.lin2(self.relu(self.lin1(input)))
```



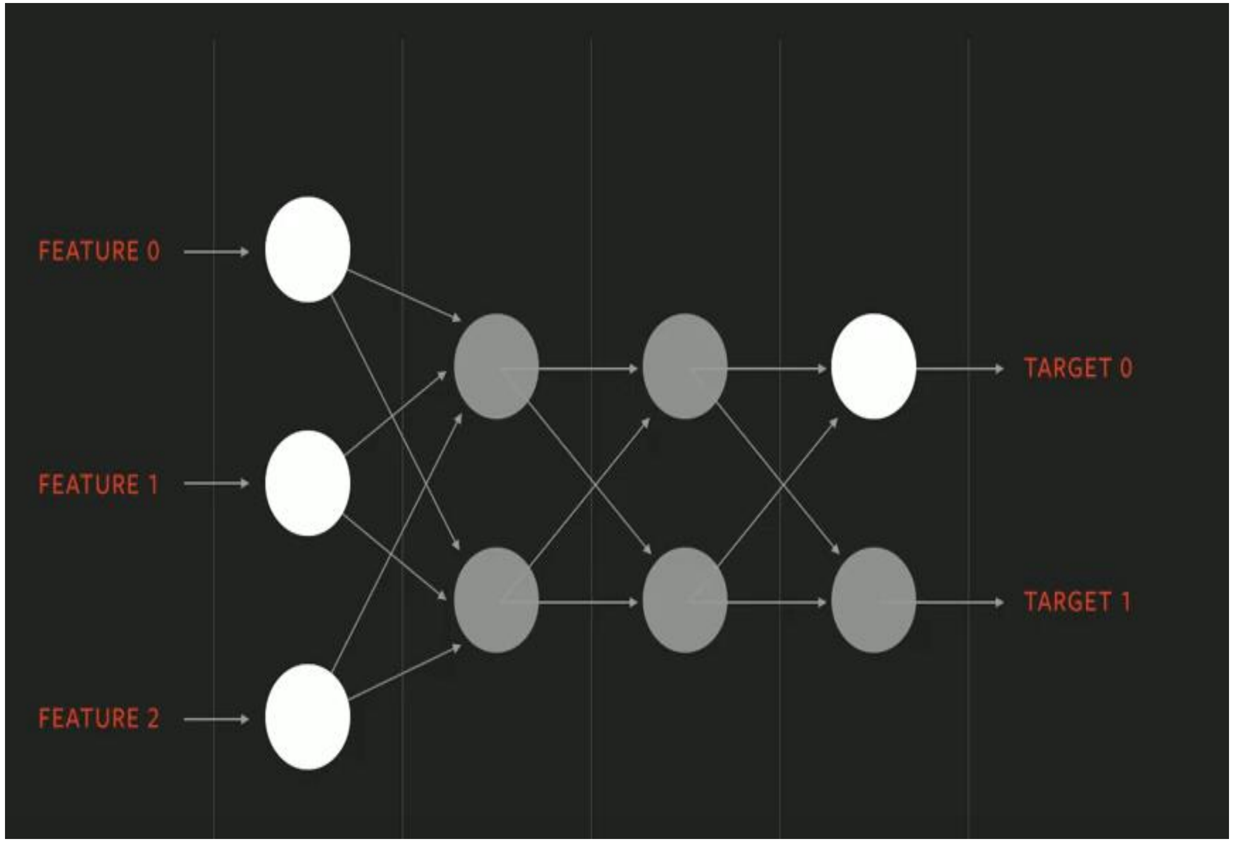
Integrated Gradients: Explanation

```
from captum.attr import IntegratedGradients
ig = IntegratedGradients(model)
input = torch.rand(1, 3)
attributions = ig.attribute(input, target=0)
print('IG Attributions:', attributions)
```

Output

```
IG Attributions: tensor([[ 0.0000, -0.5899, -1.8985]],
```

- Positive attribution score means that the input in that particular position positively contributed to the final prediction and negative means the opposite.
- Zero attribution score means no contribution from that particular feature.



Integrated Gradient: applied on image input

```python
from captum.attr import IntegratedGradients

input = samples[6].unsqueeze(0)
input.requires_grad = True
baseline = torch.zeros(input.shape).cuda()

output = model(input)
pred_label_idx = torch.argmax(output, dim=1)

ig = IntegratedGradients(model)
attributions, delta = ig.attribute(input, baseline,
                                   target=pred_label_idx, n_steps=20,
                                   return_convergence_delta=True)

print('Convergence Delta:', delta)
```

```
Convergence Delta: tensor([0.1521], device='cuda:0', dtype=torch.float64)
```

- Input: an image from Test set
- Baseline: defines the starting point for calculating feature importance

- pred_label_idx: output indices for which gradients are computed

- Calculate attributions and delta.
- The lower the absolute value of the convergence delta the better is the approximation.

Integrated Gradient: Output

```
default_cmap = LinearSegmentedColormap.from_list('custom blue',
                                                 [(0, '#ffffff'),
                                                  (0.25, '#000000'),
                                                  (1, '#000000')], N=256)

_ = viz.visualize_image_attr(
    np.transpose(attributions.squeeze().cpu().detach().numpy(),
    (1,2,0)),

    input.cpu().detach().numpy(), method='heat_map',
                             cmap=default_cmap,
                             show_colorbar=True,
                             sign='positive',
                             outlier_perc=1)
```

captum.attr.visualization.visualize_image_attr:

- Visualizes attribution for a given image by normalizing attribution values of the desired sign (positive, negative, absolute value, or all).
- Displays them using the desired mode in a matplotlib figure.

Original image

Integrated Gradient

Integrated Gradients + Smoothgrad_sq

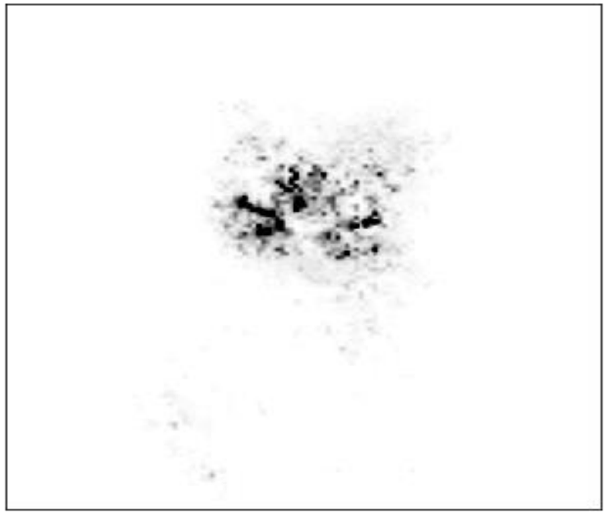
Integrated Gradient + Smoothgrad_sq

```python
from captum.attr import NoiseTunnel

nt = NoiseTunnel(ig)
attributions, delta = nt.attribute(input, nt_type='smoothgrad_sq',
                        stdevs=0.02, n_samples=10,n_steps=10,
                        baselines=baseline, target=pred_label_idx,
                        return_convergence_delta=True)


_ = viz.visualize_image_attr(
    np.transpose(attributions.squeeze().cpu().detach().numpy(),
    (1,2,0)),

    input.cpu().detach().numpy(), method='heat_map',
                            cmap=default_cmap,
                            show_colorbar=True,
                            sign='positive',
                            outlier_perc=1)
```

- NoiseTunnel smoothens the attribution score across 10 (n_samples) noise samples using smoothgrad_sq technique.
- Smoothgrad_sq : represents the mean of the squared sample attribution

Original image

Integrated Gradient

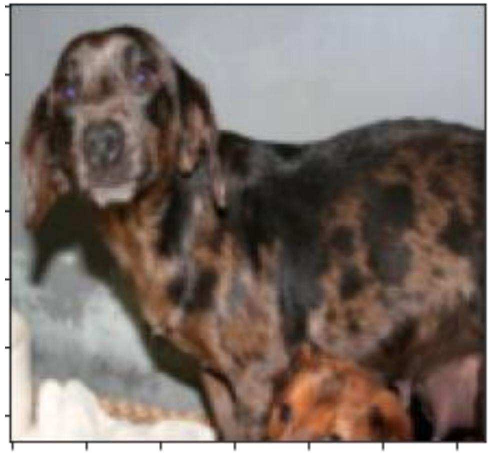


Integrated Gradient + Smoothgrad_sq

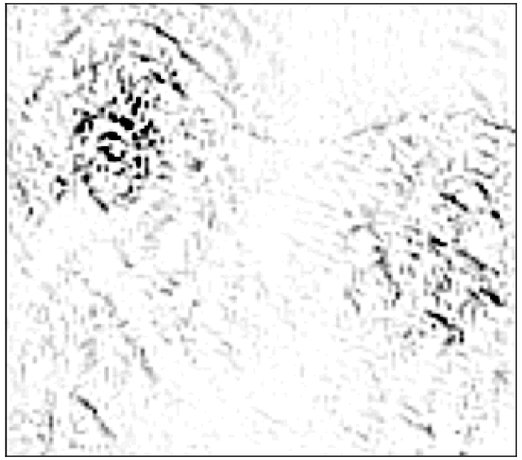
DeepLift : another gradient based algorithm

```
dl = DeepLift(model)
attributions, delta = dl.attribute(input,baseline ,
                        target=pred_label_idx, return_convergence_delta=True)
```

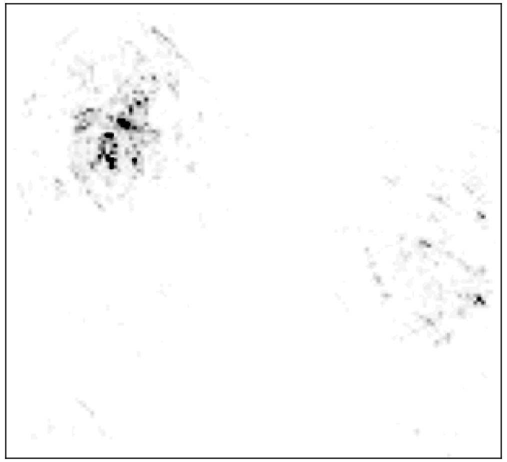
```
nt = NoiseTunnel(dl)
attributions, delta = nt.attribute(input, nt_type='smoothgrad_sq',
                    stdevs=0.02, n_samples=10,baselines=baseline,
                 target=pred_label_idx, return_convergence_delta=True)
```



Original image



DeepLift



DeepLift + Smoothgrad_sq

Layer Attribution

Layer Integrated Gradients

```
Net(
  (conv1): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=512, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=2, bias=True)
)
```

```
from captum.attr import LayerIntegratedGradients


lig = LayerIntegratedGradients(net, net.fc2)

input = test_X[1].view(1,1,50,50)
input.requires_grad = True

net_out = net(input)[0]
predicted_class = torch.argmax(net_out)

attribution = lig.attribute(input, target=predicted_class)
print(attribution.shape)
print(attribution)
```

```
torch.Size([1, 2])
tensor([[0.1460, 0.1554]], dtype=torch.float64)
```

- Resulting attributions shows importance of each neurons in fc2.
- Computes integral of gradients defined by chain rule for output target w.r.t fc2 and fc2 w.r.t to the input features.
- Attributions will always be the same dimensionality as input/output of the given layer.

Neuron Attribution

Neuron DeepLift

```
from captum.attr import NeuronDeepLift

lig = NeuronDeepLift(net, net.fc2)
input = test_X[1].view(1,1,50,50)
input.requires_grad = True

net_out = net(input)[0]
predicted_class = torch.argmax(net_out)

attribution = lig.attribute(input, 1)
print(attribution.shape)
print(attribution)
```

Output:

```
torch.Size([1, 1, 50, 50])
tensor([[[[ 5.0032e-05,  2.0641e-04,  6.3123e-04,  ...,  0.0000e+00,
            0.0000e+00,  0.0000e+00],
          [-2.0437e-04, -1.1215e-04,  3.0435e-04,  ...,  0.0000e+00,
            0.0000e+00,  0.0000e+00],
          [-1.9767e-04, -4.2504e-03, -4.5801e-06,  ...,  0.0000e+00,
            0.0000e+00,  0.0000e+00],
          ...,
          [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  ...,  0.0000e+00,
            0.0000e+00,  0.0000e+00],
          [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  ...,  0.0000e+00,
            0.0000e+00,  0.0000e+00],
          [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  ...,  0.0000e+00,
            0.0000e+00,  0.0000e+00]]]], grad_fn=<MulBackward0>)
```

- Computes attributions using Deeplift's rescale relu for particular neuron (neuron 1 in fc2) w.r.t each input feature.
- Resulting attributions shows importance of each input index
- Attributions will always be the same dimensionality(50 x 50) as the provided inputs

Out of memory Error!

- For Integrated gradients, Conductance, Internal influence or other algorithms, try reducing n_steps argument.

```
ig = IntegratedGradients(model)
attributions, delta = ig.attribute(input, baseline,
                                   target=pred_label_idx, n_steps=20,
                                   return_convergence_delta=True)
```

- For Deeplift, try reducing size of n_samples

```
nt = NoiseTunnel(dl)
attributions, delta = nt.attribute(input, nt_type='smoothgrad_sq',
                    stdevs=0.02, n_samples=10,baselines=baseline,
                 target=pred_label_idx, return_convergence_delta=True)
```

- Reduce batch size and number of epoch. Use torch.cuda.empty_cache() before running algorithm.
- Restart!!

Reference

- https://github.com/pytorch/captum
- https://captum.ai/api/