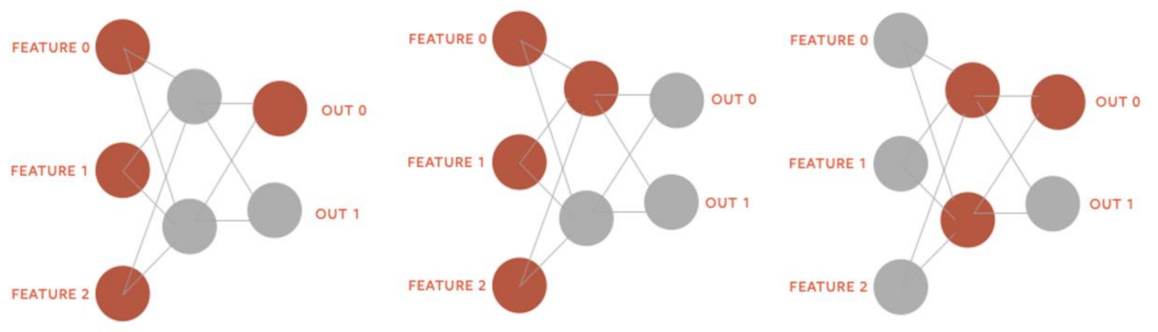
Explainability of Neural models using Captum

- Captum is an open-source model interpretability library developed by Facebook AI.
- It is implemented in Python and integrates well with popular deep learning frameworks such as PyTorch.
- The name "Captum" is derived from the Latin word "captum," which means understanding or grasping.
- It provides a set of tools and techniques for understanding and explaining the behavior of machine learning/deep learning models.
- Helps to gain insights into the inner workings of complex models and make them more transparent and explainable.
- It contains general purpose implementations of integrated gradients, saliency maps, smoothgrad, vargrad and others for PyTorch models.
- These methods aim to answer questions like "Which features or inputs are most influential in the model's decision-making process?" or "How sensitive is the model's output to changes in input variables?"
- It can be used to explore and analyze model behavior, identify biases, validate model decisions, debug and diagnose issues, and build trust in the predictions made by machine learning models.
- The attribution algorithms in Captum can be grouped into three main categories: primary-, neuron- and layer- attributions.



PRIMARY ATTRIBUTION

```
from captum.attr import IntegratedGradients

attr_algo = IntegratedGradients(model)

# Returned attributions have the same shape
# as the inputs
attributions = attr_algo.attribute(model, inputs)
```

NEURON ATTRIBUTION

```
from captum.attr import NeuronGradientShap

attr_neuron_algo = NeuronGradientShap(model,
                                      model.layer1)

# Returned attributions have the same shape
# as the inputs
attributions = attr_neuron_algo.attribute(inputs,
                                          neuron_index=0)
```

LAYER ATTRIBUTION

```
from captum.attr import LayerConductance

attr_layer_algo = LayerConductance(model,
                                   model.layer1,
           attribute_to_layer_input=False)

# Returned attributions have the same shape
# as the output of the first layer
attributions = attr_layer_algo.attribute(inputs)
```

Primary attribution:

Primary attributions, also known as feature or input attributions, focus on understanding the importance or contribution of individual input features to the model's output. They provide insights into how changes in specific features affect the model's predictions. Primary attributions are computed using techniques such as Integrated Gradients, DeepLIFT, or GradientShap, which assign importance scores to each input feature.

For example, if analyzing an image classification model, primary attributions can indicate which pixels or regions in the image are most relevant for the model's decision. This information helps understand the model's attention and identify which features contribute the most to the predicted class.

Neuron attribution:

Neuron attributions aim to explain the behavior of individual neurons within a neural network. They provide insights into the importance of each neuron's activations in influencing the model's output. Neuron attributions can be computed using methods like DeepLIFT or Layer Conductance.

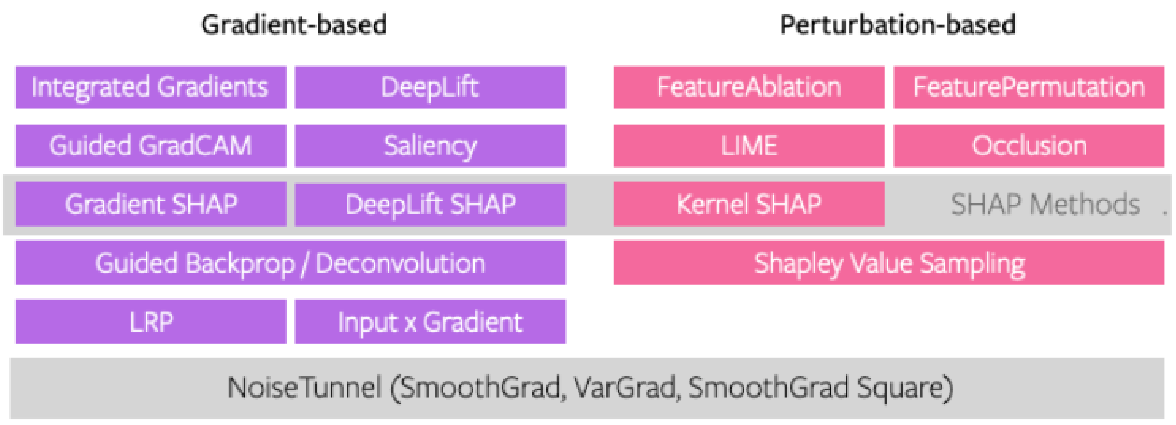
By analyzing neuron attributions, you can understand which specific neurons in the network are most responsible for certain predictions or decisions. This helps uncover patterns and characteristics of the model's internal representations and understand how different neurons interact to produce the final output.

Layer attribution:

Layer attributions focus on interpreting the behavior of entire layers within a neural network. They provide insights into the contribution of each layer's activations to the model's predictions. Layer attributions can be computed using methods such as Layer Integrated Gradients or Layer GradientShap.

Analyzing layer attributions helps understand the hierarchical and compositional nature of deep neural networks. It reveals how different layers capture and transform information as it propagates through the network, shedding light on the model's decision-making process.

- Most attribution algorithms in Captum can be categorized into gradient and perturbation-based approaches.



1) Integrated Gradients

This method computes the integral of gradients along a path from a baseline input (usually a reference point) to the input of interest. It assigns importance scores based on how the gradients change as features vary along the path.

2) **DeepLift**
DeepLIFT compares the activations of a given input to a reference activation, propagating the difference through the layers of the model to compute feature importance scores.

3) **Guided GradCAM**
Guided GradCAM (Gradient-weighted Class Activation Mapping) combines the concepts of GradCAM and guided backpropagation.
GradCAM is a technique that highlights the important regions of an input image by computing the gradients of the target class with respect to the feature maps of the CNN. It visualizes the areas of the image that strongly activate the target class.
Guided backpropagation, on the other hand, is a technique that propagates gradients back through the CNN to highlight the regions of the input image that activate specific neurons. It is used to visualize the important input features for individual neurons.

4) **Saliency**
Saliency methods work by analyzing and quantifying the importance or relevance of different elements or features in a given context, typically within images or visual data. These methods aim to identify the most salient regions or components that draw human attention or play a significant role in perception and understanding.
Different saliency methods may have variations in the specific algorithms and computations used. Some methods focus on bottom-up analysis of low-level image features, while others incorporate top-down information or task-specific knowledge to guide saliency computations.

5) **Guided Backprop/Deconvolution**
The guided backpropagation technique emphasizes positive image features that activate the target neuron while suppressing irrelevant or negative features. It helps to visualize the regions in the input image that are most responsible for activating specific neurons within the network.
Deconvolution is a similar technique that reverses the convolution operation performed during the forward pass of the neural network. It aims to reconstruct the pixel-wise activations that lead to the activation of a specific neuron. Deconvolution starts from the target neuron and propagates activations backward through the network, inverting the convolutional operations. The resulting feature map provides a visual representation of the relevant image features that contribute to the target neuron's activation.
Both guided backpropagation and deconvolution techniques provide interpretability and visualization tools to understand the image features that activate specific neurons in a deep neural network. These methods are helpful in understanding the model's decision-making process, identifying important image regions, and gaining insights into the learned representations within the network.

6) **LRP**

The key idea behind LRP (Layer-wise Relevance Propagation) is to distribute the relevance or importance of the network's output back through the layers of the network, assigning relevance scores to individual neurons or input features along the way. This allows for a fine-grained analysis of the network's behavior and helps understand which specific components contribute the most to its decisions.
LRP can be implemented using different propagation rules, such as the epsilon-LRP, Z+ rule, or alpha-beta rule, each with its own mathematical formulations and characteristics. These rules determine how relevance is propagated and redistributed through the network.

7) **Input X Gradient**
The Input X gradient method combines the gradients of the network's output with respect to the input features and the input features themselves. By multiplying these two quantities element-wise, the method determines the contribution of each input feature to the network's prediction. This method helps in understanding which input features or pixels contribute the most to the output of the neural network. It provides a localized analysis of feature importance and can be used for visualizing the significant regions in an image that influences the network's predictions.

8) **SHAP methods:**
SHAP (SHapley Additive exPlanations) is a unified framework for model interpretability that provides a set of methods to explain the predictions of machine learning models. SHAP methods are based on the concept of Shapley values from cooperative game theory and aim to allocate the contribution or importance of each feature to the overall prediction.

Gradient SHAP:
Gradient SHAP combines the concepts of Shapley values and gradient-based methods to explain the predictions of a machine learning model. It provides feature importance scores for individual input features, indicating their contributions to the model's output.

DeepLift SHAP:
DeepLift SHAP combines the DeepLIFT method with Shapley values to explain the predictions of deep learning models. It aims to provide feature importance scores, indicating the contributions of individual input features to the model's output.

Kernel SHAP:
Kernel SHAP is a model-agnostic method that can be applied to any type of model. It approximates the Shapley values by employing a sampling-based approach. Kernel SHAP randomly samples subsets of features and evaluates the model's output with and without these features to estimate their contributions.

9) **Feature Ablation**
Feature ablation is an interpretability technique used to assess the importance or contribution of individual input features or variables in a machine learning model. It involves systematically removing or disabling specific features from the input data and observing the impact on the model's performance or output.

10) **Feature Permutation**

Feature permutation, also known as feature importance permutation or ‘feature shuffling’ involves randomly permuting the values of a feature while keeping the other features intact and observing the impact on the model's performance or output. Feature permutation provides a practical and model-agnostic way to assess feature importance.

11) **LIME**

LIME (Local Interpretable Model-Agnostic Explanations) is an interpretability technique used to explain the predictions of machine learning models on an instance-by-instance basis. It aims to provide locally faithful and human-interpretable explanations for individual predictions, regardless of the underlying model's complexity.

LIME works by approximating the behavior of the underlying model in the vicinity of a specific instance of interest. It creates a simpler and interpretable model, called an "explainer" model, that can explain the predictions of the complex black-box model in a local and understandable manner. It focuses on capturing the local behavior around a specific instance, allowing practitioners to gain insights into the factors influencing the model's decision for that instance.

12) **Occlusion**

Occlusion is an interpretability technique which involves systematically occluding or covering portions of the image and observing the resulting changes in the model's output. By systematically occluding different regions of an input image, the occlusion technique helps identify the parts of the image that are critical for the model's decision-making process. Regions that, when occluded, significantly alter the model's prediction suggest their importance in the decision.

13) **Shapely Value Sampling**

Shapley value sampling is a technique used to estimate Shapley values in cooperative game theory and interpretability methods such as Shapley Additive exPlanations (SHAP). Shapley values quantify the contribution or importance of each feature or player in a cooperative game by considering all possible feature combinations and their average marginal contributions.
In situations where the number of features or players is large, computing the exact Shapley values for every subset becomes computationally expensive. Shapley value sampling offers an approximation method to estimate Shapley values efficiently.

- **SmoothGrad/ VarGrad/ SmoothGrad square**

SmoothGrad, VarGrad, and SmoothGrad Square are variations of the SmoothGrad technique used to enhance the interpretability and stability of feature attribution methods in deep learning models. These techniques aim to reduce noise and enhance the robustness of gradient-based explanations.

SmoothGrad: It involves generating multiple perturbed versions of the input and averaging the gradients obtained from these perturbations to smooth out noisy attributions.

VarGrad: VarGrad extends SmoothGrad by taking into account the variance or uncertainty of the gradients across the perturbed inputs. Instead of simply averaging the gradients, VarGrad also considers the variation or spread of the gradients. It provides an estimate of the uncertainty associated with each feature attribution by capturing the degree of consistency or inconsistency in the gradients obtained from different perturbations.

SmoothGrad Square: SmoothGrad Square is another variation of SmoothGrad that incorporates an additional squaring step. After averaging the gradients obtained from perturbed inputs, SmoothGrad Square squares the resulting gradients. This squaring operation further emphasizes the importance of features with high gradients while reducing the impact of features with low gradients. It aims to enhance the visual contrast and highlight salient regions in the attributions.

These variations of SmoothGrad, including VarGrad and SmoothGrad Square, offer refinements to the original technique to address specific challenges or improve the interpretability and stability of feature attributions. By considering gradient variance or applying non-linear operations like squaring, these techniques provide additional insights and visualizations to aid in understanding the importance of features in deep learning models.

- References:

1) Paper: https://arxiv.org/pdf/2009.07896.pdf
2) Github repository: https://github.com/pytorch/captum
3) Working notebook:
   https://github.com/pytorch/captum/blob/master/tutorials/Titanic_Basic_Interpret.ipynb
   https://github.com/pytorch/captum/blob/master/tutorials/CIFAR_TorchVision_Captum_Insights.ipynb

Palak Jain