Exercise 6

Gradient-based Explanations

Gradient-based approaches are among the most popular methods used for interpreting neural networks. They leverage white-box access to the neural network, as they rely on backpropagation to compute the gradients of an input with respect to a prediction. The intuition behind using gradients for explanatory purposes is that they tells us, how sensitive the prediction is to small changes in the input. In this assignment, we will generate different gradient-based explanations for a ResNet18 Model pretrained on the ImageNet Dataset by 1) applying off-the-shelf methods and 2) implementing gradient-based feature attribution from scratch.

- You can get a bonus point for this exercise if you pass at least 85% of the tests. Code is automatically tested after pushing. If your tests fail, either your code is wrong, or you solved the task differently. In the second case, we will manually check your solution.
- Three collected bonus points result in a 0.33 increase of the final grade.
- You are allowed to work in groups with up to three people. You do not need to specify your ErgebnisPIN as it is already collected from the entrance test.
- Follow the 'README.md' for further instructions.
- Finish this exercise by 24th November, 2021 at 11:59 pm.

Note: This assignment requires installing new packages: {torchvision, captum} (See *requirements.txt*)

1 Applying Gradient-based Feature Attribution Methods

In this first task, the goal is to use off-the-shelf implementations of different gradient-based feature attribution methods to obtain explanations. Here, we apply different methods implemented in Captum, a model interpretability library for PyTorch.

Associated file: *tasks/explainers.py*.

1.1 Gradient

Complete *get_gradient* to produce a saliency map based on the gradient with respect to the model's prediction as the target class, for a given input image. Use *captum.attr.Saliency*.

1.2 Integrated Gradients

The Integrated Gradients method interpolates gradients between a non-informative baseline $\bar{x}$ and the actual input x. Complete *get_integrated_gradients* to produce the Integrated Gradients with respect to the model's prediction as the target class, for a given input image. Use *captum.attr.IntegratedGradients*.

1.3 SmoothGrad

The SmoothGrad method helps to reduce noise by adding noise. Complete *get_smoothgrad* to produce a smoothed saliency map for a given input image. Use *captum.attr.NoiseTunnel* in combination with *captum.attr.Saliency*.

1.4 Preparing Feature Attribution Maps for Visualization

The resulting feature attribution maps have the same dimensions as the input to the neural network. For visualization purposes, we often aggregate them to smaller dimensions to produce heatmap-like outputs. Complete *aggregate_attribution* that aggregates an attribution map with three color channels to a single color channel by summing over the channel dimension.

For proper visualization, we additionally want to normalize the attribution maps. Complete *normalize_attribution* that first takes the absolute values of the attributions, then normalizes them into the range $[0, 1]$ by subtracting the minimum and afterwards dividing by the maximum.

1.5 Visualizing Feature Attributions

After generating all the explanations, we of course want to plot them. Complete the *plot_attributions* function that plots the image and the generated attributions in a row.

2 Gradient-based Feature Attribution from Scratch

Now, the task is to implement two of the methods from task 1 from scratch.
Associated file: *tasks/custom_explainers.py.*

2.1 Gradient

Complete *get_custom_gradient* that implements the simple gradient method:

$$A = \frac{\partial f(x)_c}{\partial x_i} \tag{1}$$

where A is the resulting feature attribution map, f is the neural network model, x is an input image and c is the index of the class we are interested in. That means $f(x)_c$ is the neural network output corresponding to class c. Here, the class we are interested in is the one that is predicted by the model.

2.2 Integrated Gradients

Now we want to implement the Integrated Gradients method from scratch.

First, complete the *get_path* function, that creates a path of images. The path starts from a baseline $\bar{x}$, ends with the actual image x and is filled with intermediate samples in between. This path essentially contains the samples

$$\tilde{x} = \bar{x} + \alpha(x - \bar{x}) \tag{2}$$

where $\alpha = 0$ for the baseline and $\alpha = 1$ for the actual image. Here, we choose a black image as the baseline.

Next, complete *get_custom_integrated_gradients* to generate the Integrated Gradients using the following equation:

$$A = (x - \bar{x}) \times \int_{\alpha=0}^{1} \frac{\partial f(x)_c}{\partial \tilde{x}_i} \bigg|_{\tilde{x}=\bar{x}+\alpha(x-\bar{x})} \tag{3}$$

where the integral is approximated by averaging the gradients for each sample in the path.