Sanity Checks for Saliency Maps

Alexander Haenel, Florian Kneifel, Lennart Rosseburg

July 2021

1 Introduction

Highlighting important features of the input data becomes more and more relevant when it comes to debugging or understanding learned models. Explanation methods are used as tools for this purpose inside environments where an input is considered to be relevant for the predictions of a learned model. Especially, *saliency methods* have emerged as a popular class of tools designed to do that task of elucidating important aspects from an input, which is typically an image, and through that creating a so-called heat map. Based on their results, the individual decisions made by deep-learning models are tried to be explained. (Adebayo et al. 2018)

As more contributions were made, a methodological challenge became clear: "*the difficulty of assessing the scope and quality of model explanations*" (Adebayo et al. 2018). Adebayo et al. (2018) addressed this challenge, as they did not find it sufficient, but rather misleading, to rely solely on visual assessments. They showed with multiple experiments that several existing saliency methods are independent to both the model and the data generating process. As consequence, those methods are not suitable for tasks sensitive to either model or data, like finding outliers, etc. To approach this problem, and to properly evaluate explanation methods, a methodology based on statistical randomization tests was proposed. They compare the results from the natural experiment with results from two distinct instantiations of an randomized experiment. Their methodology can be thought of as a sanity check for saliency methods. (Adebayo et al. 2018)

The first instantiation is the *model parameter randomization test*. Through comparison with the output of an randomly init. untrained network of the same architecture, this test checks if the saliency method depends on the learned parameters of the model. If so, the output should differ drastically between both cases. Otherwise, the saliency method is insensitive to the models properties. They also tested different randomization techniques. Those were: randomly re-initializing all weights completely or in cascading order(starting from the top layer) and independently randomizing a single layer at a time while keeping others fixed. (Adebayo et al. 2018)

The second test, *the data randomization test*, compares two different results of a saliency method. Both on the same model and dataset, but one time the datasets labels are randomly permuted. This is done in order to test the sensitivity of the method to the relationship between instances and labels. So again, significant different outputs should be expected. Elsewise the saliency method is insensitive to relationships between instances and labels from the original data and thereby cannot explain mechanisms that depend on that. (Adebayo et al. 2018)

2 Re-implement a sanity check using PyTorch (and Captum)

For this and the following badges please refer to the *src* and *results* folder or to the notebook "SaliencyMapsCNN.ipynb".

We decided to re-implement the *data randomization test*, or sanity check, from the paper. In this re-implementation we are reusing an CNN from one of our previous Exercises. The CNN architecture looks as follows: input → conv (3x3, 48) → ReLu → conv (3x3, 96) → ReLu → max-pool (2x2) → conv (3x3, 192) → ReLu → max-pool (2x2) → fully connected (64 units) → ReLu → fully connected (10 units). We train the model with the stochastic gradient descent (SGD) optimizer for 5 iterations while using an

momentum of 0.9 on the CIFAR10 dataset. With the descried setup we achieved an average accuracy of 75%.

figs. 1 and 2 shows our results obtained with the integrated gradients saliency method on one of the CIFAR10 images. It can be observed, that the saliency map trained with randomized labels differs strongly from the one trained correctly.

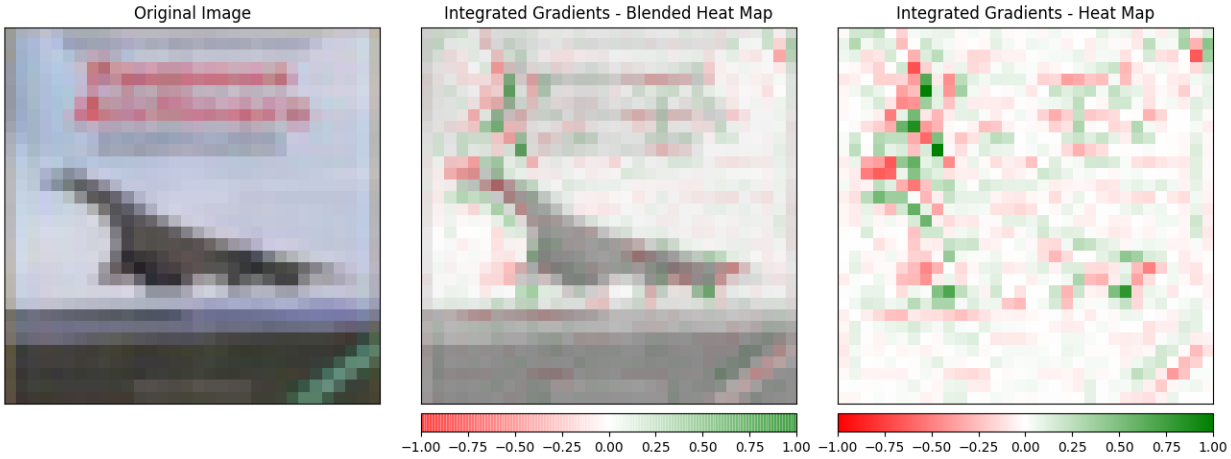


Figure 1: Integrated Gradients

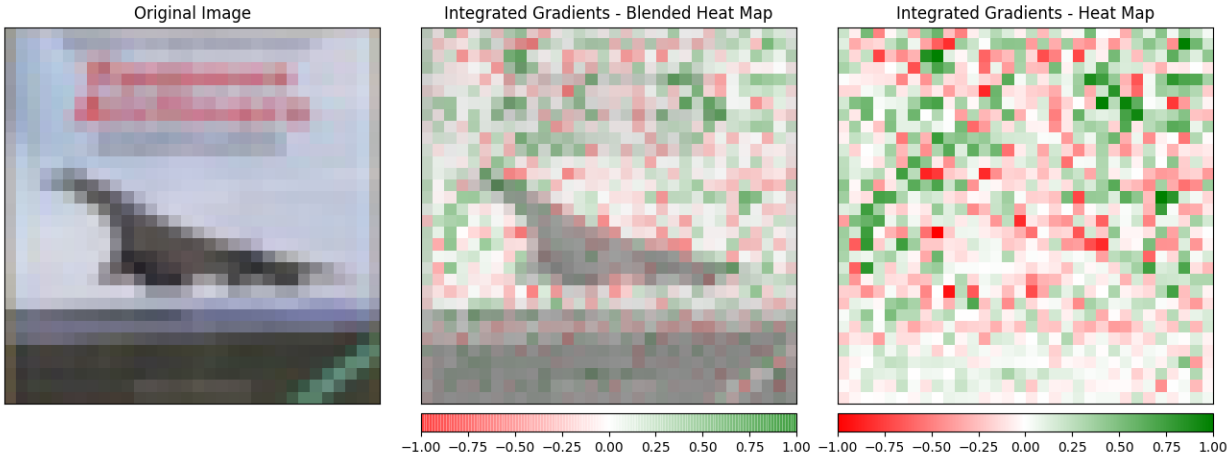


Figure 2: Integrated Gradients with randomized labels

3 Experiment with different saliency methods

Additionally to the integrated gradients saliency methods we also implemented DeepLift and an occlusion-based saliency method. From the figures below you can see that the saliency map generated from DeepLift is very simular to the one from integrated gradients, which also applies to the model trained on randomized labels. On the other hand the occlusion-based saliency map even on the correctly trained model looks a bit random and therefore there is no big difference to the model trained with randomized labels. The reason may be, that the model is not trained well enough with an accuracy of 75%.

figs. 3 to 6 show our results obtained with the named salicy methods on one of the CIFAR10 images.

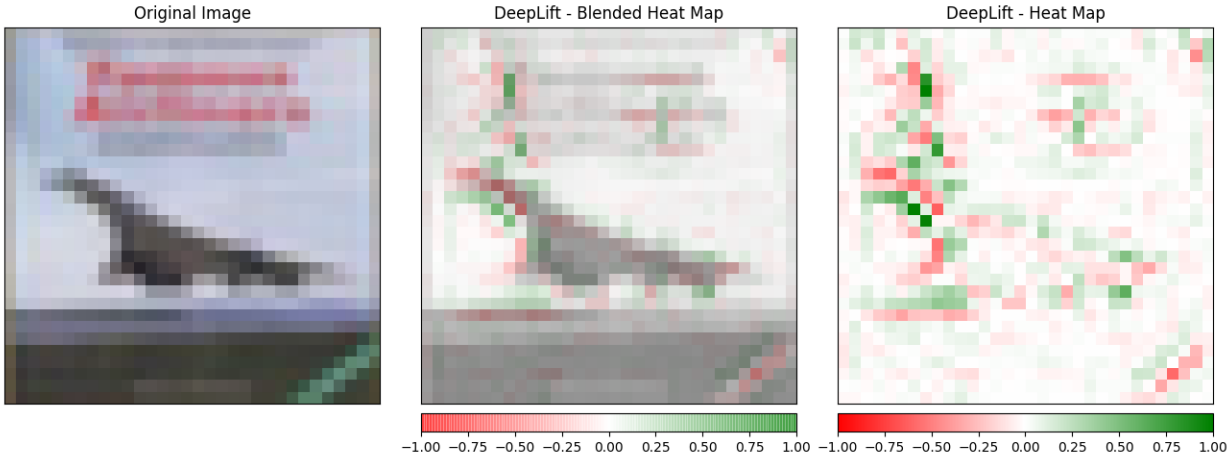


Figure 3: DeepLift

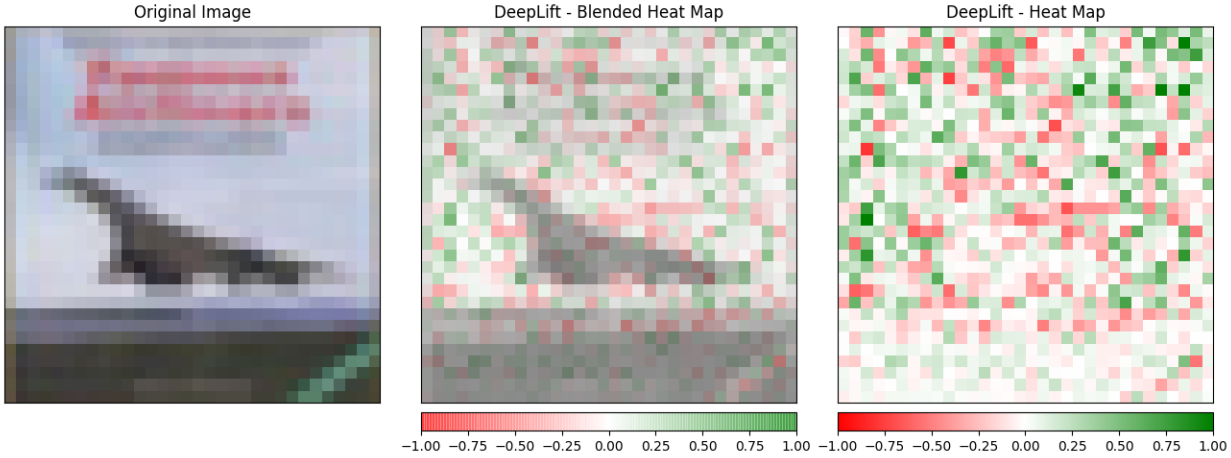


Figure 4: DeepLift with randomized labels

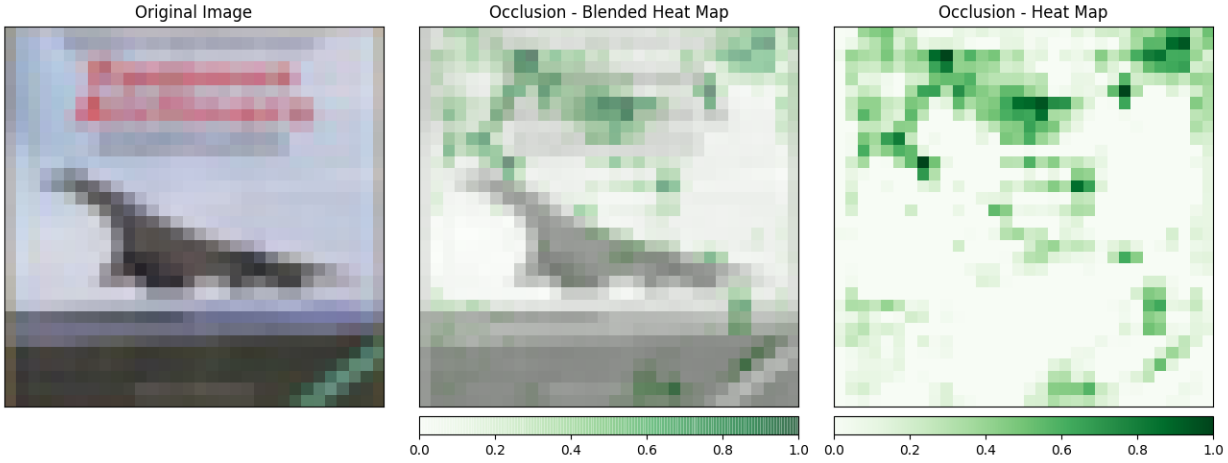


Figure 5: Occlusion-based saliency method

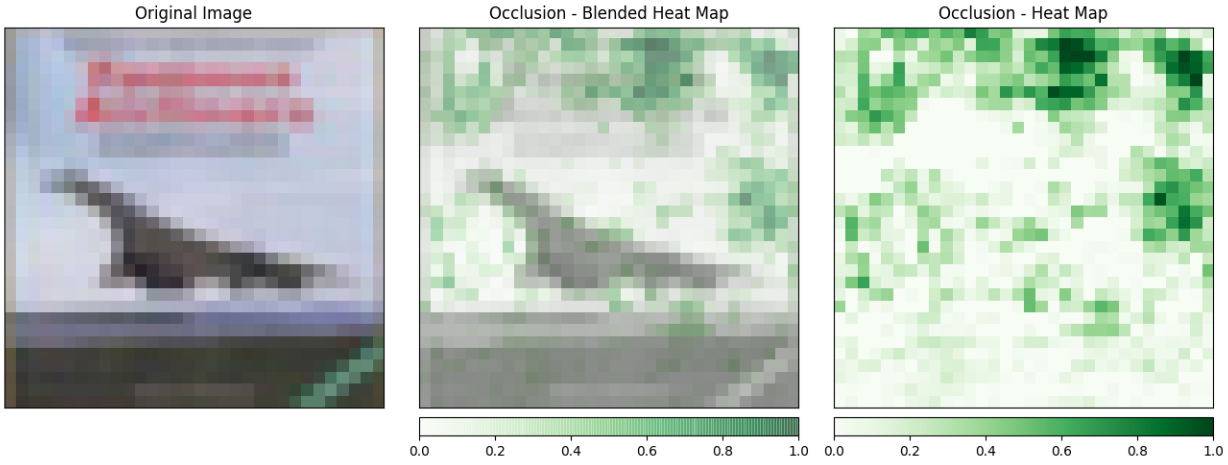


Figure 6: Occlusion-based saliency method with randomized labels

4 Run experiments on a different image classification dataset

Instead of using the CIFAR10 dataset, we also tested the methods on the MNIST dataset. Since both datasets are structured equally, it was pretty easy to replace the dataset. Only two small adjustments where necessary. The input channels changed from three RGB channels for the CIFAR10 dataset to one input channel because of the gray-scale images on the MNIST dataset. Furthermore the number of neurons on the flattened linear layer changed, because of the different image sizes on CIFAR10 with 32x32 and on MNSIT with 28x28.

Altogether we achieved an accuracy of 99% with the described model. Somehow the captum module decided to show gray-scale images as purplish images but this shouldn't bother that much.

figs. 7 and 8 show our results obtained on one of the MNIST images with the integrated gradient saliency method. The saliency map is only visible where the number is actually written, because everywhere else the image is black (in the figures below purple) and therefore the input on these parts zero.

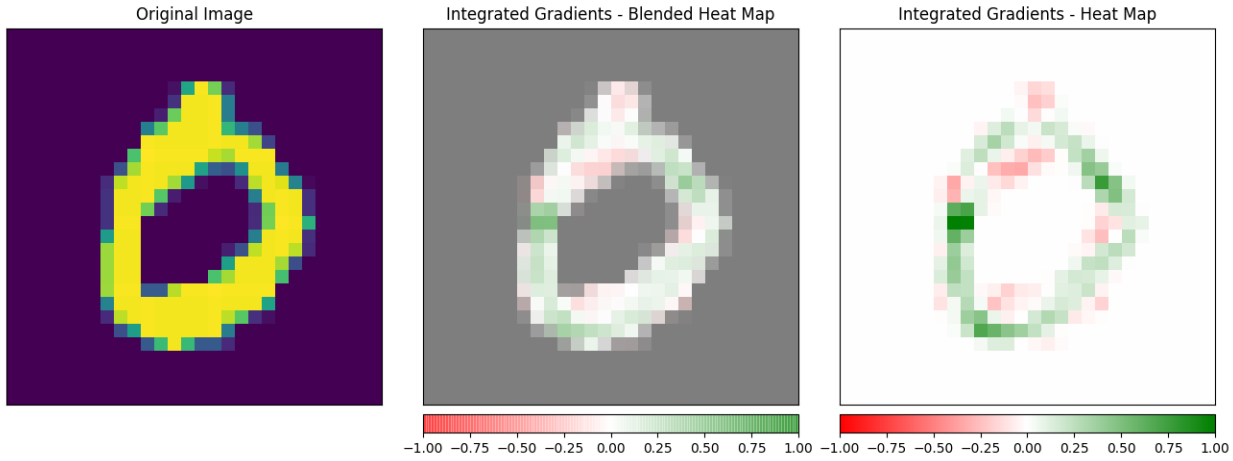


Figure 7: Integrated Gradients on an MNIST image

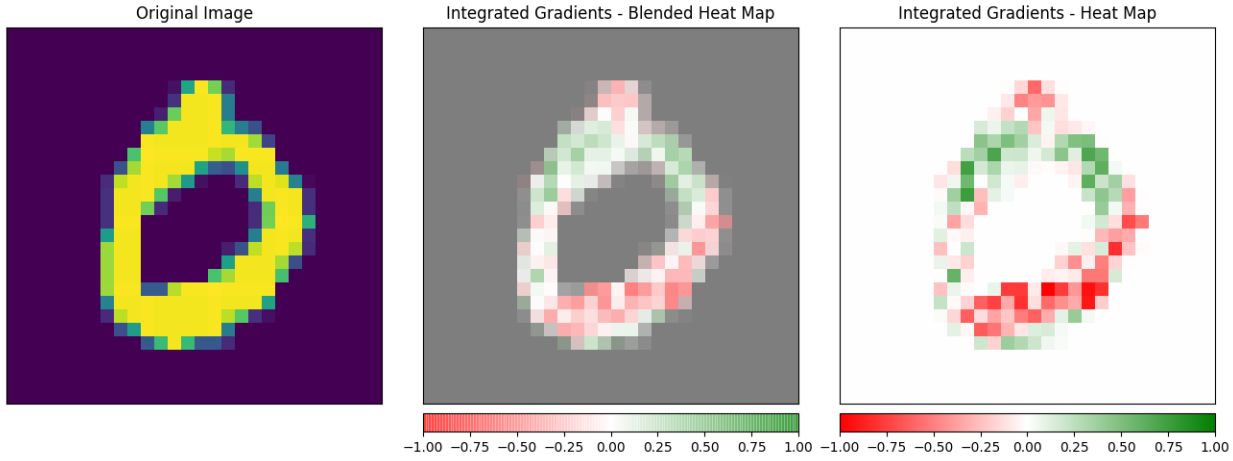


Figure 8: Integrated Gradients with randomized labels on an MNIST image

5 Run experiments on non-image data

For this badge please refer to the notebook "captum_on_timeseries.ipynb"

5.1 data

As a simple toy-dataset for timeseries data, we use the "flights" dataset that comes with the seaborn library. The dataset contains the number of flight passengers per month for the years 1949 to 1960. So in total there are 144 datapoints. Just by looking at the data we can see a yearly pattern, but also a steady increase on a macro scale. Also for easier use the data in the notebook gets normalized before training a model. The task will be set up as a regression problem: The model gets the data from the last 10 months and has to predict the next month. Note that here we intentionally used a window smaller than 12, because otherwise the task will become to easy because of the intra-year-pattern.

5.2 simple linear model

For the first experiment we use a relatively simple linear model with the following architecture:

- Input of size 10
- Linear-Layer with 3 neurons
- Sigmoid-Layer
- Linear-Layer with 1 neuron

Considering the simplicity of the model, we get pretty accurate predictions:

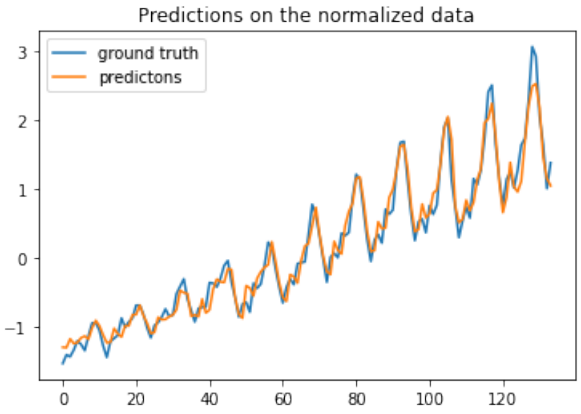


Figure 9: linear model performance

First we can look at which input feature on average has the most importance for the prediction:

There seems to be a pretty strong focus from the network towards the input at index 9. That is not surprising, considering that the number of flights most of the time is not drastically different from the last month. Looking at the average importance of neuron 0 in 11a and the negative correlation of almost every input for neuron 0 in 11b one might conclude, that neuron 0 is trained to predict a decline, while neuron 1 and especially neuron 2 are mostly predicting increasing flight passengers.

5.2.1 Pattern recognition

The result-pictures for this section are not added to the report, because they took up too much space. You can find them in the second half of the notebook captum_on_timeseries.ipynb

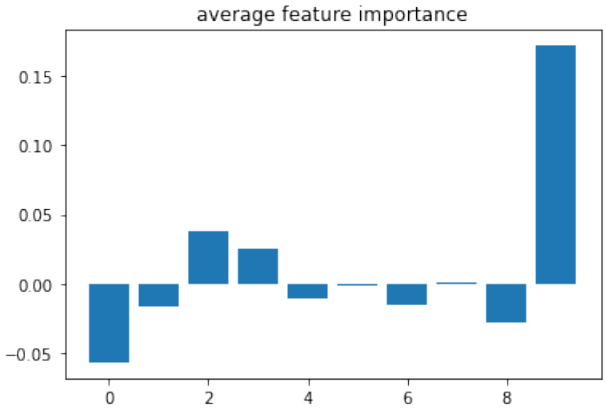
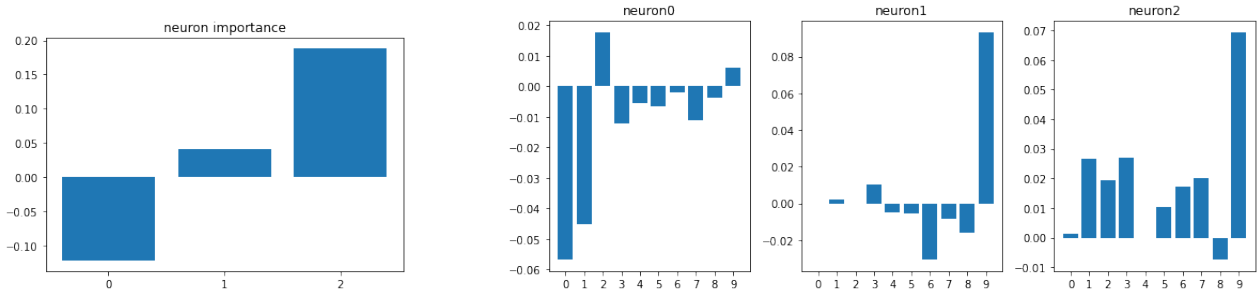


Figure 10: average feature importance



(a) neuron importance

(b) average feature importance per neuron

Figure 11: average importance

Because of the way the data is organized, we can use a step size of 12 to get the same month for consecutive years. At first sight, we might assume that there is a certain regularity in the way the inputs are interpreted. But if we use a random model to repeat the same experiment, we see similar patterns emerging. (however because the attributions are way smaller a factor 10 is added to the visualization) In conclusion, we can not really get any insight in the pattern recognition performed by the neural network from this experiment.

For the last experiment we use a CNN to better resemble an architecture, that one would also find for image tasks. Just comparing the different methods provided by Captum shows quite different results[1]. Especially looking at the input at index 9 shows a strong positive signal from the method "saliency" but is negative for "integrated gradients" and "deep lift".

6 Conclusion

All in all it is interesting to see which neuron behaves which way, and how the input features are correlated to the output. However interpreting the workings of the neural net from this information is not possible. For the timeseries-task it is even harder, because even with a lot of training we as humans can not simply look at a short timeseries plot and predict the next value. At picture classification humans are naturally gifted and can generally perform well. But even there we sadly can not gain much insight in how the neural net "sees" pictures.

The proposed sanity checks (Adebayo et al. 2018) of randomizing either the model parameters or the training labels are very helpful to verify, that a saliency method is indeed representing the neural network. For visual tasks this is quite important, because in many cases a basic edge-detector might fool us into thinking, that our network actually extracted all the meaningful structures.

[1] last cell of the notebook