Sanity Checks for Saliency Maps

Konrad Wienecke 10005023, Hauke Hinrichs 10007270

Deep Learning 2021

Access to our code: *Click here*.

1 Introduction

The topic of our project is sanity checks for saliency maps. The corresponding paper is "Sanity Checks for Saliency Maps" from Adebayo et al. `https://arxiv.org/abs/1810.03292` with their attached code `https://github.com/adebayoj/sanity_checks_saliency`. We re-implemented the sanity check "cascading randomization", which was proposed in the paper, for numerous saliency maps and for two model architectures (Basic CNN and ResNet-18) trained on two different datasets (MNIST and ImageNet). Moreover we implemented a comparison of the structural similarity index measure (SSIM) for the saliency maps of the ResNet-18 during cascading randomization. You can find our code at `https://github.com/Heyjuke58/sanity_checks_pytorch`.

2 Re-implementation of a Sanity Check with PyTorch and Captum (Badge 1)

2.1 Model

We used exactly the same model structure as in the paper for the basic CNN classifying the MNIST dataset. You can see the exact structure below:

```
Sequential(
  (conv1): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (relu1): ReLU()
  (maxpool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (relu2): ReLU()
  (maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear1): Linear(in_features=3136, out_features=1024, bias=True)
  (relu3): ReLU()
  (linear2): Linear(in_features=1024, out_features=10, bias=True)
 )
```

Structure of our Basic CNN (string representation by PyTorch)

We trained this model on the MNIST dataset for a total of 50 epochs with the ADAM optimizer, a learning rate of $1e-4$ and a batch size of 50. These are exactly the same hyperparameters the authors of the papers stated in their code. We achieved a best validation accuracy of 0.994. The model parameters which achieved this best value were then chosen as our final parameters for the saliency experiments.

2.2 Cascading Randomization

In order to replicate the sanity checks proposed by the authors, we randomized the model as described in the following: A layer of the model is randomized by resampling the weights randomly from a normal distribution with the same mean and standard deviation of the prior model weights of that specific layer. For the cascading we followed the procedure from the paper. They randomized the model from the top layers to the bottom layers. The top layers in this case are the layers near the output (linear2 for our basic CNN structure). Respectively, the bottom layers are the ones near to the input. In each randomization step one more layer is being randomized, while all layers on top of this one layer are also randomized again. Therefore the saliency maps are computed for differently randomized models in each randomization step. As an example, here are the layers that are randomized in each step for the basic CNN structure:

Step	Layers randomized in this step
1	linear2
2	linear2, linear1
3	linear2, linear1, conv2
4	linear2, linear1, conv2, conv1

2.3 Saliency Method

We implemented the saliency method *Gradient* (in Captum: Saliency). In Figure 1 you can see our results for 5 MNIST examples.

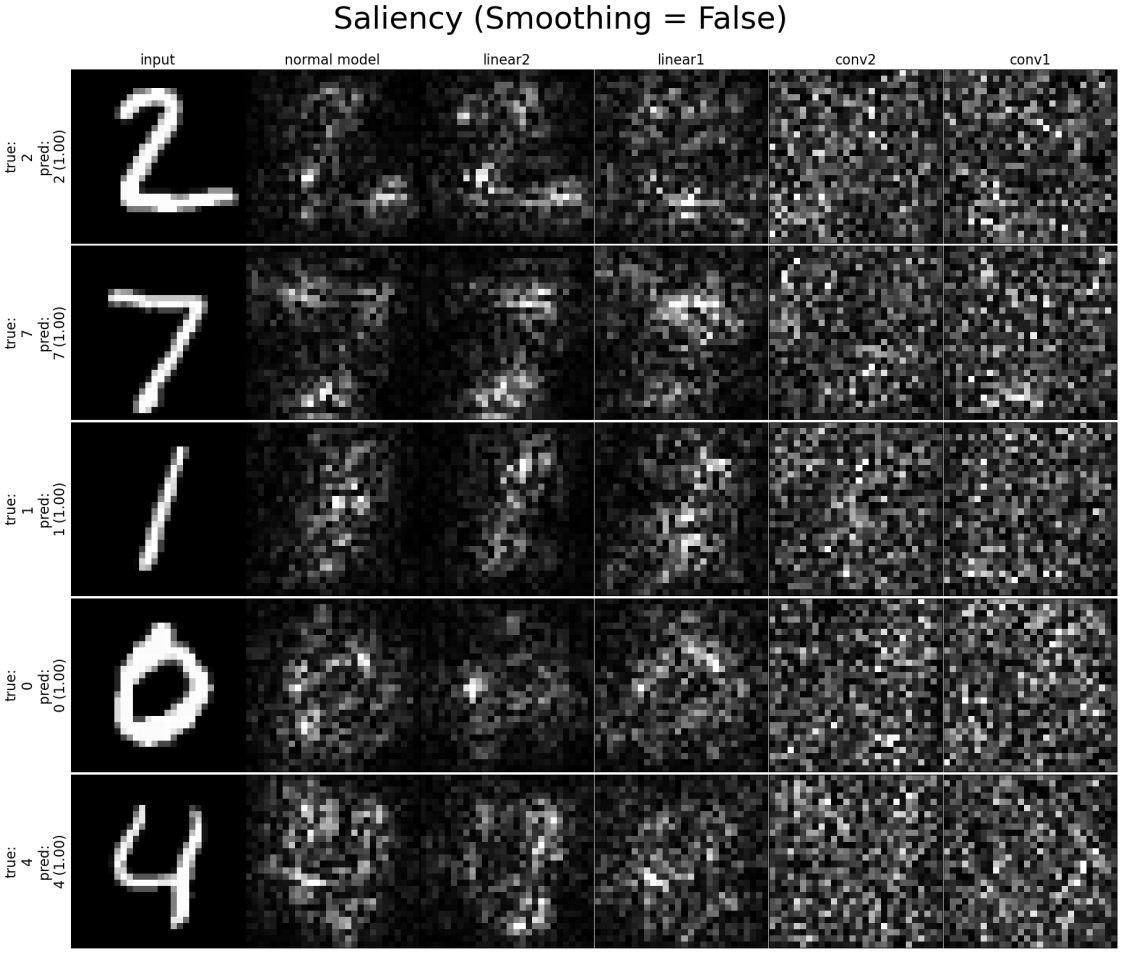

Figure 1: Sanity check of basic CNN on MNIST: Saliency method *Gradient* (no smoothing)

In the first row of Figure 2 you can see the results published in the paper for the same saliency method with the same basic CNN structure. The claimed sensitivity of the *Gradient* saliency method to the model parameter randomization by the paper is clearly also visual in Figure 1 when you see the scrambling of the map that happens at the lower layers. We note that the resulting saliency maps from the paper generally look a bit blurred whereas our results seem to look somewhat sharper.

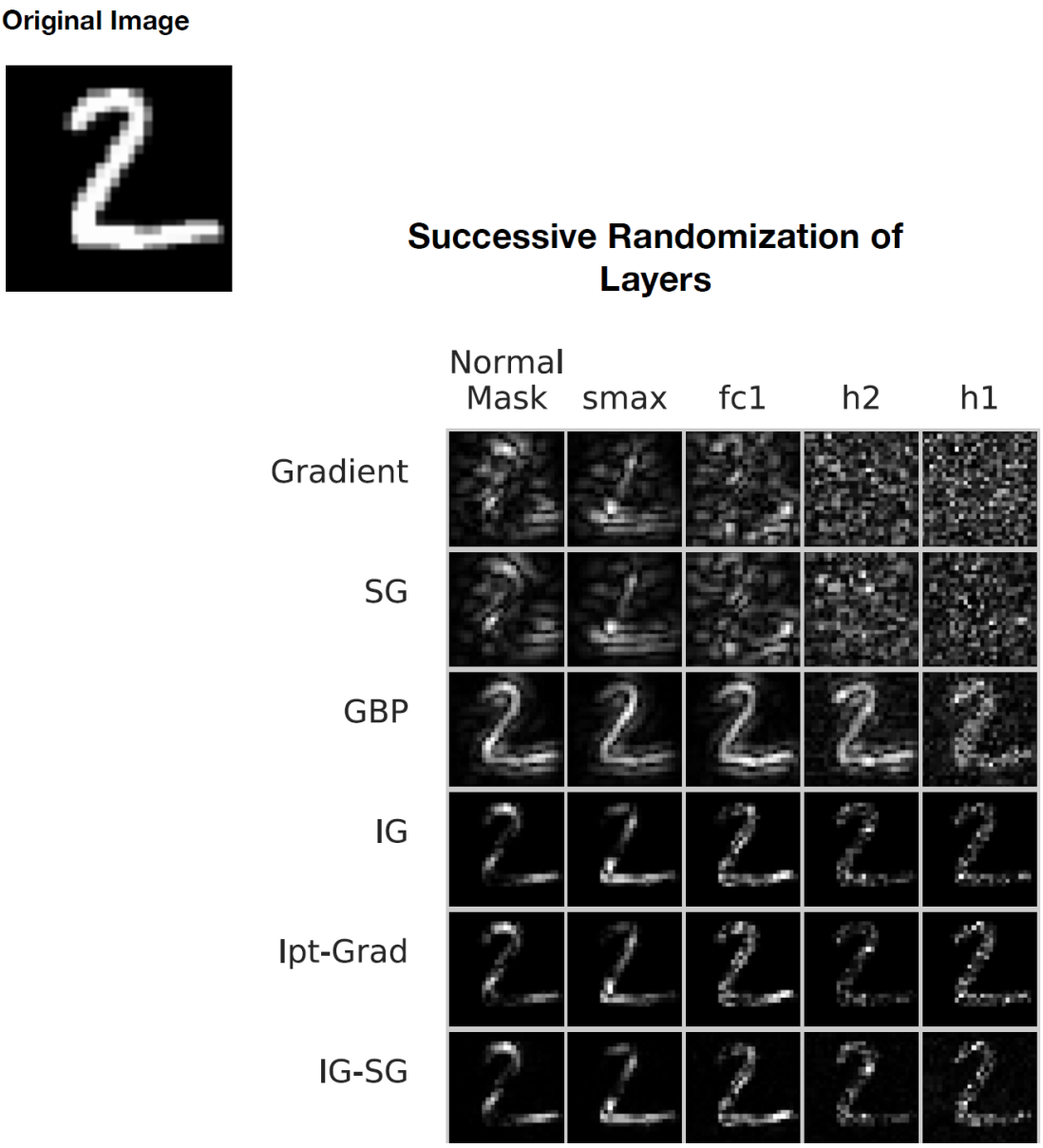

Figure 2: Part of a sanity check for successive (equivalent to cascading) randomization for the basic CNN model and numerous saliency methods on MNIST published in the paper. The layers `smax`, `fc1`, `h2` and `h1` are equivalent to the layers `linear2`, `linear1`, `conv2` and `conv1` in our model.
Clarification of the abbreviated saliency methods:

- SG $\equiv$ SmoothGrad
- GBP $\equiv$ Guided Back-propagation
- IG $\equiv$ Integrated Gradients
- Ipt-Grad $\equiv$ Gradient $\odot$ Input
- IG-SG $\equiv$ Integrated Gradients-SG

3 Experiments with Different Saliency Methods (Badge 4)

Next, we experiment with multiple saliency methods in addition to Gradient, while still working with the basic CNN and the rather simple MNIST dataset. We compare the different saliency methods regarding their output on the non-randomized model as well as during cascading randomization. These are the examined saliency methods, including the previously examined *Gradient* method:

1. Gradient (Captum module: Saliency)
2. SmoothGrad (Gradient combined with SmoothGrad (Captum module: NoiseTunnel))
3. Gradient $\odot$ Input (Captum module: InputXGradient)
4. Gradient $\odot$ Input-SG (Gradient $\odot$ Input combined with SmoothGrad)
5. Guided Back-propagation (Captum module: GuidedBackprop)
6. Integrated Gradients (Captum module: IntegratedGradients)
7. Integrated Gradients-SG (Integrated Gradients combined with SmoothGrad)

All of these saliency methods except for *Gradient* $\odot$ *Input-SG* are examined in the paper. The used saliency methods could be implemented without any further tuning of per-method parameters. As for using *Noise-Tunnel*, Captum's implementation of *SmoothGrad*, we extracted the appropriate parameters from the paper and the code:

1. The type of smoothing to be used: `nt_type = 'smoothgrad'`
2. The number of samples for smoothing: `nt_samples = 50`
3. The standard deviation for the additive noise: `stdevs = 0.15`

The results for all these saliency methods can be seen in Figure 3. In the following, we compare the saliency maps that we created with those shown in the paper (see Figure 2). We note that all of their saliency maps seem a bit blurred than ours, as previously noted. In some cases, some edges in the saliency maps are more prevalent in the paper's visualizations. This difference may exist because different packages were used to generate the saliency maps. Note also that results will inherently differ as two different trained model instances were used between our project and the paper.

3.1 Comparison of the Results

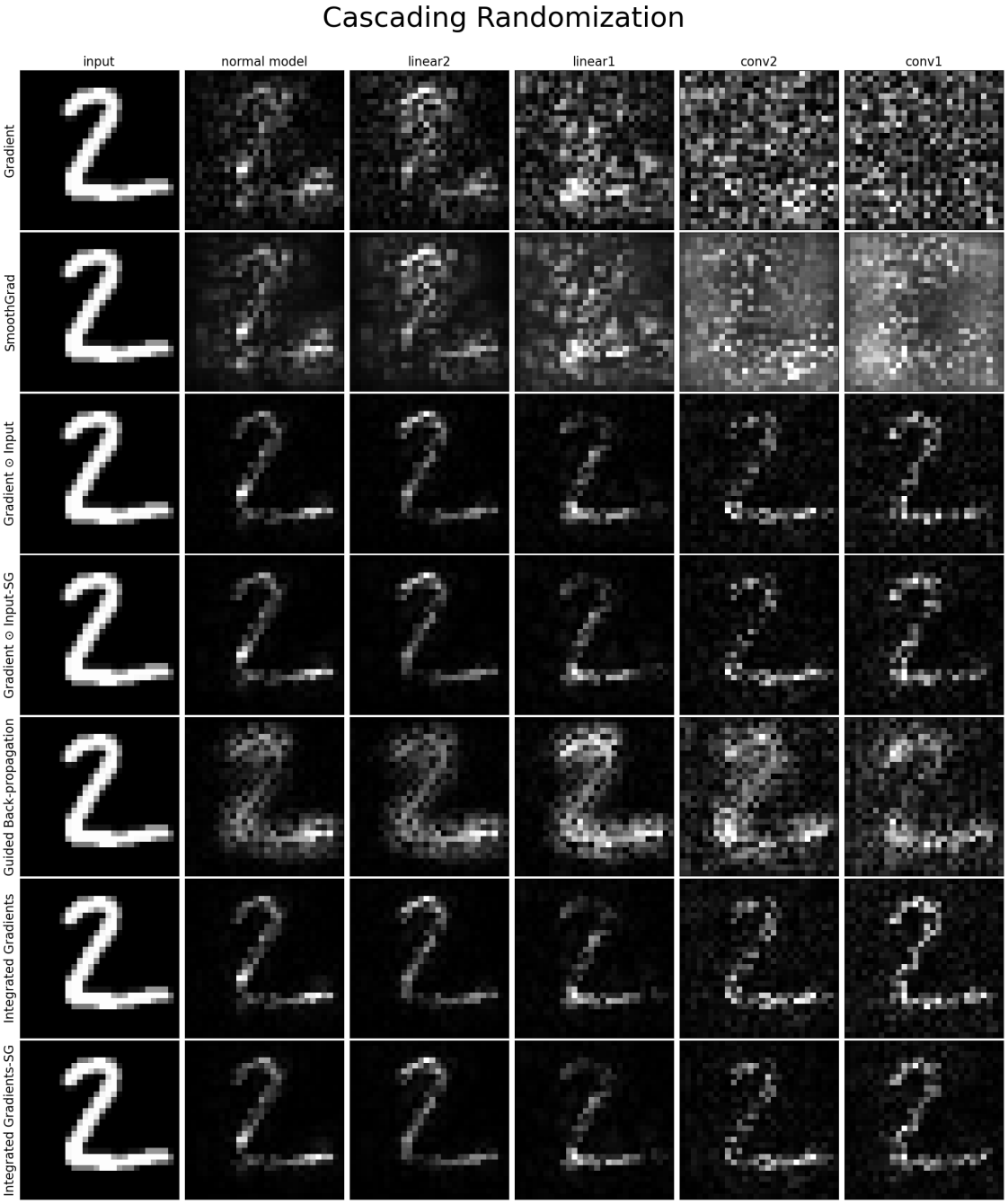

Figure 3: Sanity check of basic CNN on MNIST: Multiple saliency methods

SmoothGrad. All *SmoothGrad* saliency maps except those where the convolutional layers are scrambled look very similar to the *Gradient* maps, just a bit less noisy around the edges. A similar observation can be made when comparing the maps for these two saliency methods in the paper. However, our saliency maps start looking very different from those in the paper after randomizing the convolutional layers. Here we can observe that while the saliency maps stay very noisy, they are generally much brighter than those shown in the paper.

Gradient ⊙ Input. Our results here look relatively similar to those in the paper. We note that when randomizing the convolutional layers, some noise appears in the parts of the image not occupied by the number. In the original saliency map, this noise is not visible. Another difference lies specifically in the saliency maps generated at the third stage of randomization. Here, the original saliency map shows a number that looks more coarse, and generally a bit brighter than what our saliency map shows.

Gradient ⊙ Input-SG. This method is not examined in the original paper. Our saliency maps here look very similar to those generated by the same, but unsmoothed, saliency method. The only visible difference seems to be a bit less noise in the last two stages of randomization.

Guided Back-propagation. Our results using *Guided Back-propagation* show similarities to those in the paper. A halo around the number is clearly visible even after the first three stages of randomization. Our saliency maps seem to become noisier more quickly in the steps where the convolutional layers are randomized.

Integrated Gradients and Integrated Gradients-SG. These two methods produce very similar saliency maps in both the original paper and in our reproduction. Our results for both of these methods closely resemble those found in the paper.

4 Experiments on the ImageNet Dataset with ResNet-18 (Badge 2 & 5)

The badges "Experiment with different model architectures" and "Run experiments on a different image classification dataset" could in our eyes not be separated reasonably. Therefore we will address them together in this section. As mentioned above we experimented with ResNet-18 which was pre-trained on the ImageNet dataset. We acquired it from the *torchvision* package found at `https://github.com/pytorch/vision`.

4.1 Cascading Randomization

A residual net tends to be deeper compared to other model architectures, since skip connection are employed. We decided therefore to not randomize each existing layer in its own step, but rather take whole residual blocks, of which the ResNet-18 has 8, and randomize them entirely in one step. These blocks contain multiple convolutional and batch-normalization layers that each have to be randomized. You can see in the model structure in the appendix how exactly the blocks are composed. In Figure 4, the column titles - with help of the model structure in the appendix - indicate how we grouped the blocks and the surrounding layers into the randomization steps. We again start at the top layer, which is closest to the output, in this case `fc`.

4.2 Experiments

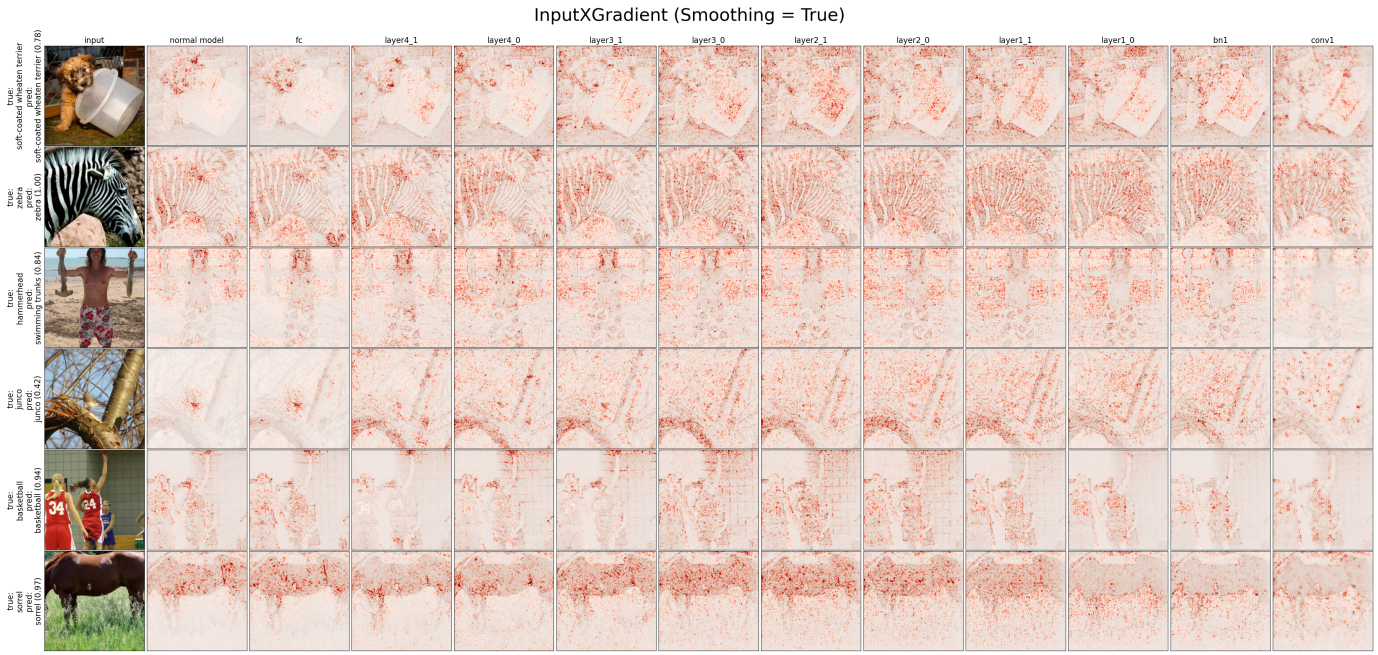

Figure 4: Sanity check of ResNet-18 on ImageNet: Gradient $\odot$ Input-SG

At first we selected a few example images from ImageNet and ran experiments for the saliency method *Gradient* $\odot$ *Input-SG*. For better comparing the saliency maps to the original images we chose to show the originals very lightly in the background of the maps. The following conclusions can be drawn from the results shown in Figure 4. Shift your attention to the saliency map we got from the normal model for the forth picture (a junco bird). Firstly the model correctly classified the junco with a confidence of 0.42. Secondly we can assess the model has clearly made the correct decision for the correct reason, since we see it mostly pointed its attention to the junco. When we now look at the maps in that row, progressing the amount of randomization of the model, the junco is certainly not any more the center of attention but rather edges are. In contrast to that, the third picture of a man holding a hammerhead shark is being classified erroneously as swimming trunks, although the model obviously does not really care for the actual swimming trunks but rather the mans face, nipples and the beach. With progressing randomization the activations of the map are basically all over the place.

4.3 Comparison of the Results

We now examine the effects of cascading randomization on multiple saliency methods. The previously established methods in Section 3 were all tried, however we had mixed results with some of the methods, namely *Guided Back-propagation* and *Integrated Gradients-SG*. We compare the resulting saliency maps (see Figure 6) with an example from the paper (see Figure 5). Notably, we are comparing saliency maps constructed throughout cascading randomization on the same image (cropped slightly differently), but with two different underlying models. This means that we cannot directly compare any two saliency maps. Rather, we roughly look at how the saliency maps evolve during cascading randomization.

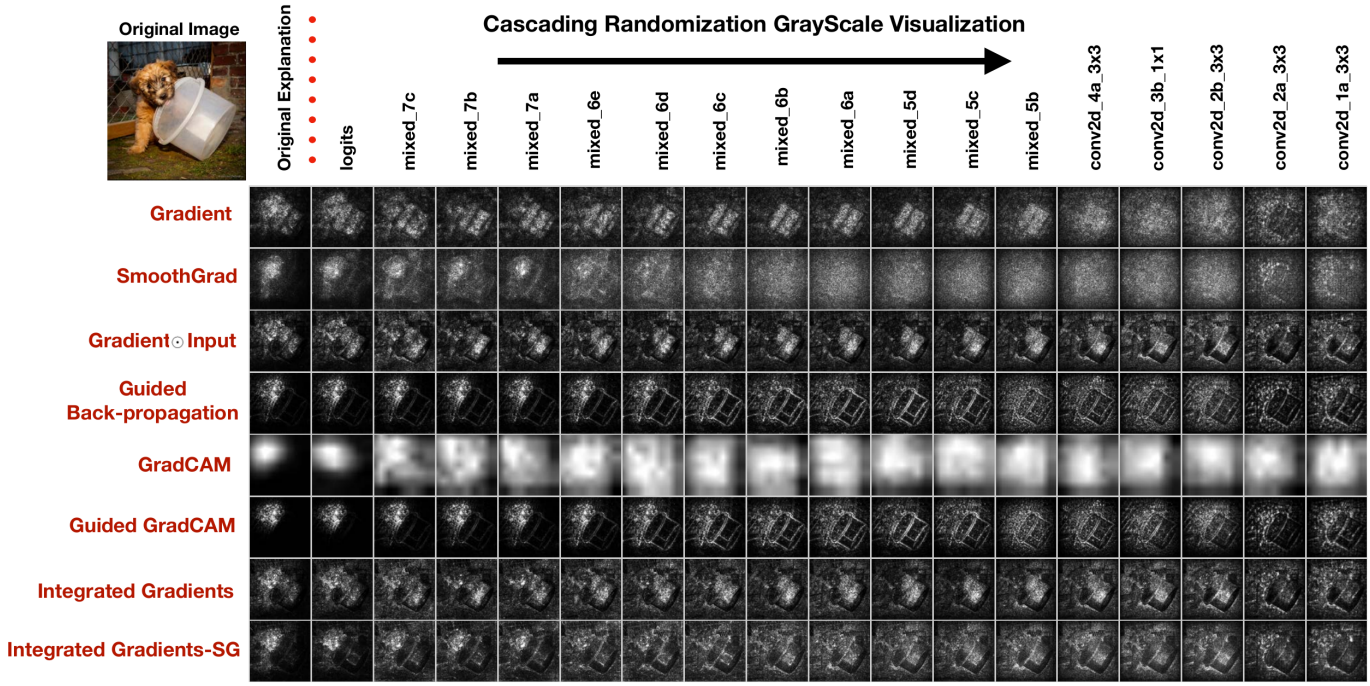

Figure 5: Sanity check for cascading randomization for Inception v3 on ImageNet using numerous saliency methods, from the paper.

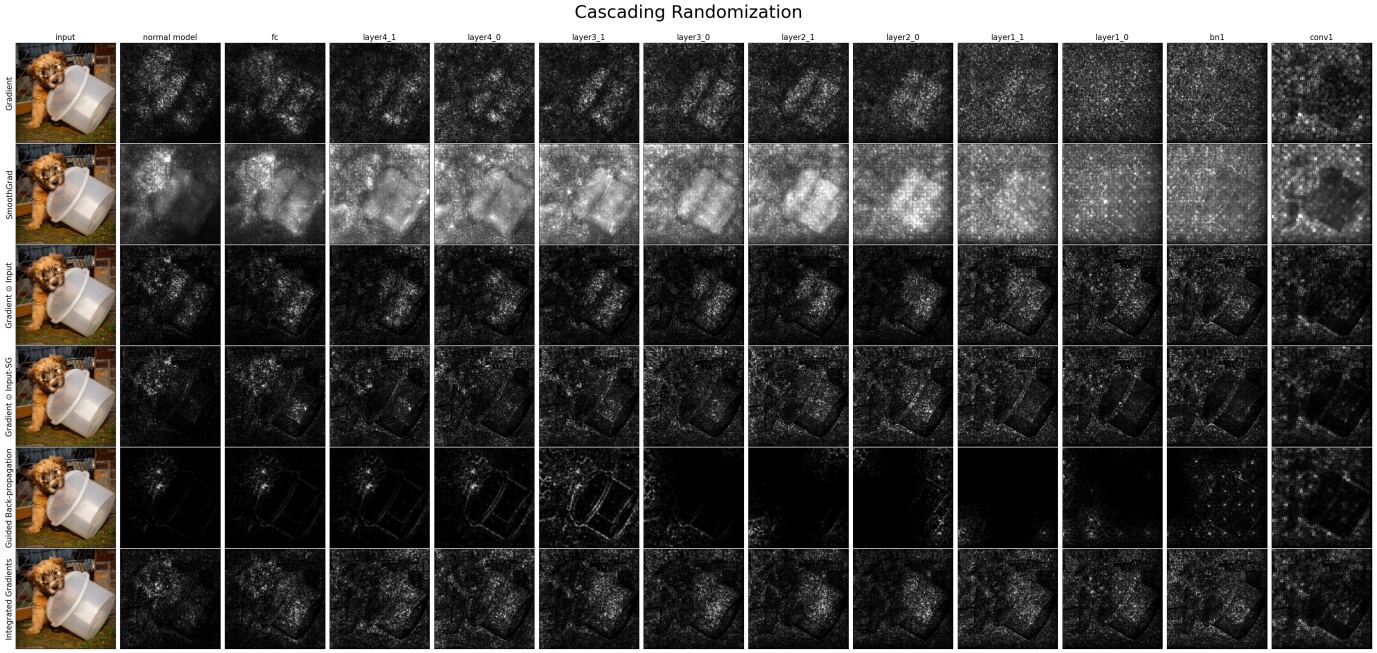

Figure 6: Sanity check for cascading randomization for ResNet-18 on ImageNet using numerous saliency methods.

Gradient. The saliency maps for *Gradient* show some interesting similarities. In our saliency maps, the bucket is clearly visible during randomization, until `layer1_1` is randomized. At this point, most detail is overwhelmed by noise. However, a sort-of negative image of the bucket can then be seen after full randomization. The saliency maps from the paper somewhat mimic this observation.

SmoothGrad. Here, our saliency maps show peaks around the face of the dog at start. During randomization, *SmoothGrad* behaves similarly to *Gradient*, but the maps are generally much brighter. The maps from the paper do not show a similar behaviour. Instead, details of the dog and bucket are now more quickly dissolved into noise. However, the maps start looking more similar to each other again after almost full randomization of the models. At the latter stages of randomization, some grid patterns can be observed in our maps. These could be artifacts due to JPEG-compression (which uses a grid size of 8) or due to the applied kernel (which uses a grid size of 7).

Gradient $\odot$ Input. Our results for this saliency method compare nicely to those from the paper. Throughout randomization, our maps change only very slightly. For most of the randomization, hotspots remain on the bucket. In contrast, the dog only appears in the maps of the unrandomized and the first randomized models. Towards the end, the maps become more noisy. In the maps from the paper, very similar observations can be made.

Gradient $\odot$ Input-SG. This method is not examined in the paper. However, we examined it on our CNN using MNIST data where we found almost no differences compared to the unsmoothed *Gradient $\odot$ Input*. Yet in the case of ImageNet, there are some interesting differences between the maps. For one, the original saliency map now shows clear hotspots on the dog rather than the bucket. While the bucket also becomes more prevalent during randomization, noise also builds up more quickly as some hotspots on the dog's nose and eyes remain.

Guided Back-propagation. PyTorch raised a warning when we calculated this saliency map:

```
UserWarning:
Using a non-full backward hook when the forward contains multiple autograd Nodes is
deprecated and will be removed in future versions.
This hook will be missing some grad_input.
Please use register_full_backward_hook to get the documented behavior.

UserWarning:
Attempting to normalize by value approximately 0, visualized results may be misleading.
This likely means that attribution values are all close to 0.
```

The *non-full backward hook* is used the Captum implementation of *Guided Back-propagation*. Our guess is that due to missing input gradients the "attribution values are all close to 0" and this is the reason why in the maps large parts of gradients are missing. Therefore the pictures do not have any validity.

Integrated Gradients. Before randomization, both original saliency maps of the two models show hotspots on the face of the dog. With cascading randomization, the bucket becomes more pronounced in the saliency maps for both models as hotspots on the face of the dog disappear more quickly in our saliency maps than in those presented in the paper.

Integrated Gradients-SG. This saliency method crashed our Jupyter kernel in the ResNet-18 and ImageNet setup. Therefore we can not evaluate it.

4.4 SSIM Comparison

Motivation. Since it is somewhat difficult to qualitatively compare the saliency maps, we use the structural similarity index measure (SSIM) to quantitatively compare the different saliency methods from our

implementation and theirs. We again have to keep in mind that we are comparing the SSIM of two different models, meaning that direct comparisons of values at specific randomization stages is not possible. Still we will point out a few things. But first we will clarify how we computed the score.

Approach. We took the original saliency map and compared it with the map after each randomization step for each saliency method. Before comparing the maps we normalized their channel values from 0 to 1, as done in the paper. Also, we use the same implementation of SSIM calculation from *skimage*. Due to runtime considerations, we reduce the amount of images over which the score is averaged in the paper from 50 to 20. You can see our results in Figure 8. The results published in the paper are shown in Figure 7.

Guided Back-propagation. We again had issues with *Guided Back-propagation*, namely that some computed SSIMs gave us a NaN. While not being entirely sure why this happened our guess is this is because the attributions were very sparsely correct and often close to 0, as seen before, so the SSIM could not be calculated. But overall this occurred in only 3 randomization steps (`layer2_1`, `layer2_0` and `layer1_1`). We handled it by ignoring the $NaNs$ and relying on the average of the remaining calculated scores, which was in the worst case 9 from 20 scores in `layer1_1`. Although our score for *Guided Back-propagation* might not be valid, we see that it matches the score from the paper, staying the most similar of all saliency methods throughout the cascading randomization. Qualitatively, we cannot fully verify this since in our case, looking at the row of *Guided Back-propagation* in Figure 6, we see that after `layer3_1` the maps are visually very different from before, being black almost everywhere. But since our quantitative result depicts a high remaining similarity we can confirm the insensitivity of *Guided Back-propagation* to the model parameters.

Integrated Gradients, Gradient $\odot$ Input, and Gradient $\odot$ Input-SG. Apart from that, our scores also in part match with those from the paper considering *Integrated Gradients* and *Gradient $\odot$ Input*. The only difference is that our score for those saliency methods (0.7) is a bit lower than the scores in the paper, but they also mingle around that value during the cascading randomization. *Gradient $\odot$ Input-SG* can also be assigned to this group of methods.

Gradient and SmoothGrad. In the case of *Gradient* the result from the paper and ours clearly differ. While the score in the paper is at around 0.7 at the deepest layer, ours is below 0.4. But still this discrepancy might be caused due to usage of different model architectures. While being very different it certainly displays that *Gradient* does have sensitivity to the model parameters and hence passes the sanity check. Also *SmoothGrad* has a lower SSIM compared to other methods and therefore would in our eyes also quantitatively pass the sanity check.

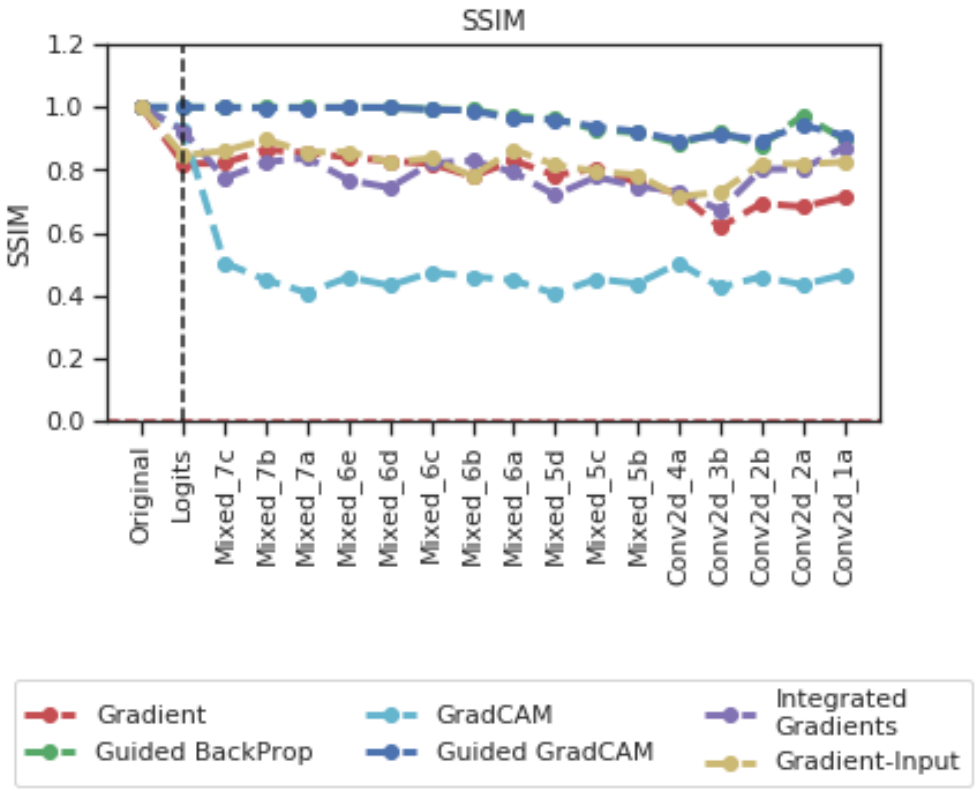

Figure 7: SSIM for different saliency methods for an Inception-V3 on ImageNet (averaged over 50 images) results from the paper

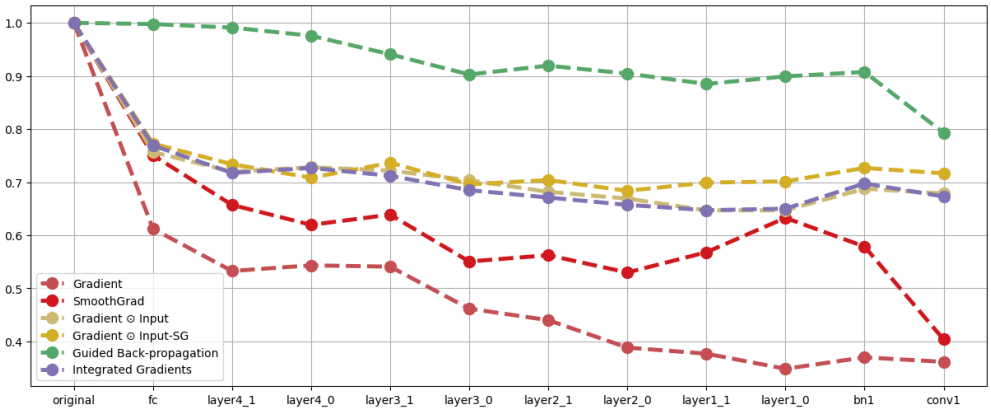

Figure 8: SSIM for different saliency methods for a ResNet-18 on ImageNet (averaged over 20 images)

A ResNet-18 Model Structure

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

```
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```