

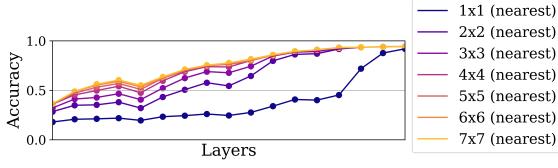
A Experiments on the ablation of probe performance due to downsampling strategies

As a concession to practicality, it is necessary to reduce the dimensionality of the feature maps to train logistic regression probes on the output of convolutional layers. This problem is addressed by the original authors [10] with two fairly crude solutions. The first being a global average pooling on the feature map and the second one a random selection of positions on the feature map. We reject the random strategy, since we want to avoid adding a random component to our measurements that may introduce noise or instability. The global pooling strategy looks more promising to us, since global pooling is also performed on most recent architectures as an interface between convolutional and dense sections of the network [9, 8, 12, 15, 16, 18]. However, global average pooling inside neural architectures is generally performed on the last layer’s output, which can be expected to be fairly low dimensional on the height and width axis compared to earlier layers. Furthermore, due to the smaller receptive fields of earlier layers, the encoded information will be more local and thus more heterogeneous based on the position of the entry. Since these circumstances are likely to negatively affect the performance and / or introduce artifacts into the probe performance measurements of early layers, we decide to look for less invasive downsampling strategies.

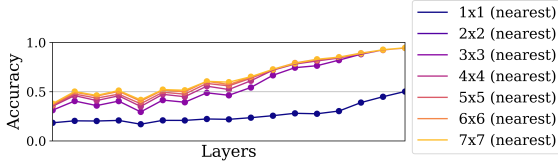
We test downsampling of feature maps using the nearest interpolation algorithm as well as adaptive average pooling to a smaller feature map size. For testing, we use two models as test benches. First, the modified version of ResNet18 described in the last experiment of section 4.3, which is trained on a 32×32 pixel resolution. Second, the original ResNet18 implementation trained on 224×224 pixel input resolution. Both models are trained for 90 epochs using stochastic gradient descent with an initial learning rate of 0.1 and a momentum of 0.9. The learning rate is multiplied with 0.1 every 30 epochs. These training setups both feature no tail pattern, which is important for testing the effect of the downsampling strategies, since we expect more aggressive downsampling to have a negative effect on probe performance. Negative effects would be harder to interpret on late layers with a tail pattern, since a tail effectively means that the problem is already solved and the layers perform basically on the same performance level as the output. We choose two input resolutions to observe the effects of downsampling on two different scales.

We train probes on feature maps reduced to a maximum of 1, 2, 3, 4, 5, 6 and 7 pixels in height and width using both average pooling and downsampling.

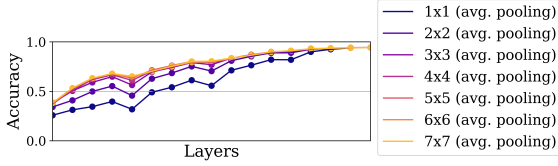
The results can be seen in Fig. 8. On large feature map resolutions, we observe that both strategies produce very similar patterns. However, the structural integrity of those patterns is better maintained on average pooled feature maps. When reduced to a single depth-vector, the nearest-downsampling destroys the pattern otherwise visible on any other downsampling resolution. Other than that, increasing the size of the reduced feature maps has a globally negative impact on probe performance. Earlier layers being affected worse from the decrease than later layers, which is expected for the aforementioned reasons. Based on the results of these experiments, we decided to compute probe performances in this paper using feature maps adaptive average pooled to a resolution of 4×4 , which is computable with reasonable computational resources while maintaining the relative structure of the probe performances.



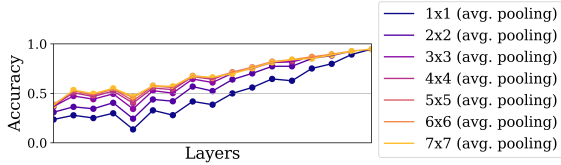
(a) Probes trained on downsampled feature maps. The model was trained on 32×32 input resolution.



(b) Probes trained on downsampled feature maps. The model was trained on 224×224 input resolution.



(c) Probes trained on adaptive average pooled feature maps. The model was trained on 32×32 input resolution.



(d) Probes trained on adaptive average pooled feature maps. The model was trained on 224×224 input resolution.

Figure 8: As expected, more aggressive reduction in feature maps sizes negatively impact the probe performance. Average pooling seems to be able to maintain the structure of probe performances better. Downsampling to a single pixel induces heavy artifacts, destroying the otherwise prevalent pattern of probe performances.

B Details on experimental setups

B.1 Details on experimental setups from experiments in section 3

B.1.1 Dataset and data augmentation

The experiments are conducted on CIFAR10. The images are channel-wise normalized with $\mu = (0.4914, 0.4822, 0.4465)$ and $\sigma = (0.2023, 0.1994, 0.2010)$. At training time, the images are first cropped randomly with a 4 pixel zero-padding on all edges. The size of the crop is 32×32 pixels. Then the crops are horizontally flipped randomly with a probability of 50%. The images of the training set are reshuffled after each epoch.

B.1.2 Models

The experiments use VGG11, 13, 16 and 19 as well as four additional variations of the aforementioned architectures. The variations have all filter sizes reduced by a factor of 2, 4, 8 and 16. Furthermore, the architectures are slightly modified by adding batch normalization layers after each convolutional layer. In addition, the flattening layer serving as the connection between convolution feature extractor and densely connected classifier is replaced by a global pooling layer. The variations are included in order to not only include models of various depth, but also of varying width in our results. These are two common degrees of freedom in neural architecture design. The modification to the architectures are made to include common architectural features that can be considered standard in most modern architectures. Furthermore, the global pooling layer makes the models agnostic towards the input resolution. This enables us to alter the input resolution without changing the number of parameters inside the model. PCA-Layers for projecting the network are added after each convolutional and linear layer.

B.1.3 Training setup and parameters

We use the same training setup for all models we test in this chapter. Since we are interested in in-development scenarios, we do not apply hyperparameter optimization. Instead, we use default-values of PyTorch wherever possible and otherwise settings that are generally in the common range of hyperparameters used for similar classification tasks. The exact hyperparameter settings are depicted in Table 3. We find that 30 epochs is enough time for all models to converge to a stable solution on CIFAR10.

Parameter	Values
Epochs	30
Batch size	128
Optimizer	ADAM
ADAM: beta1	0.9
ADAM: beta2	0.999
ADAM: epsilon	1e-8
ADAM: learning rate	0.001

Table 3: Hyperparameters common to each of the experiments in Section 3

B.1.4 Number of experiments conducted

In the experiment in Fig. 1 a total of 60 models are trained. Each neural architecture is trained 3 times using the same setup. The experiments in Tables 1 and 2 are repeated 26 and 40 times respectively on the same model using the same setup. We also ran 15 additional experiments on ResNet18 and VGG11, similar to the aforementioned experiments. The results of these are depicted in Table 9 and 12.

B.1.5 Details on PCA-Layer Projections

The eigenvalues and eigenvectors of the output of all PCA-Layers are computed when the switch from training to testing occurs at the end of an epoch. At this point, the projection matrix is computed and the aggregation variables (running sum, running squares and number of seen samples) are reset in each PCA-Layer. The PCA-Layers keep the last computed covariance matrix in memory as an internal variable. This allows us to recompute the projection matrix $P_{E_l^k}$.

B.2 Details on experimental setups from experiments in section 4 and 5

B.2.1 Dataset

The experiments conducted in Section 4 use models trained on CIFAR10, ImageWoof and ImageNette. We additionally reproduce some results on MNIST and TinyImageNet and compute the saturation levels of ResNet18 on ImageNet. These results are not depicted in Section 4; however, these results are included in Appendix C. We choose these datasets to test our hypothesis on different levels of complexity, regarding the number of classes as well as the natural resolution of the images.

The preprocessing and data augmentation is the same as in Section 3. The input resolution differs depending on the dataset and the running experiment. If not mentioned otherwise, the images are processed in their native resolution. MNIST data is additionally transformed into RGB to avoid changes in the neural architecture. In any case, the resizing is performed after the augmentation pipeline is applied on a batch of images.

B.2.2 Models

The experiment uses the same models as in Section 3. Additionally, we train ResNet18 and ResNet34 with filter sizes reduced by a factor of 1, 2, 4 and 8. We include the ResNet architectures to include another architecture, with a feature (skip connections), that may affect how the information is flowing through the network. We also remove the skip-connections on ResNet18 and 34 for two experiments to observe the effects of disabled skip connections. Different from previously described experiments, none of these architectures have PCA-Layers.

B.2.3 Training setup and parameters

We choose a static training setup of all models and datasets, with the same reasoning as in the experiments conducted in Section 3. Compared to the setup described of the experiments in Section 3, the batch size is changed to 32 (16 in the cases of ResNet34, VGG16 and 19)

due to memory limitations. However, we find through brief exploration that slight changes in the hyperparameter optimization described here as well as additional epochs of training do not influence the results described in Section 4 in any meaningful way.

Parameter	Values
Epochs	30
Batch size	32 (16 for ResNet34, VGG16 and VGG19)
Optimizer	ADAM
ADAM: beta1	0.9
ADAM: beta2	0.999
ADAM: epsilon	1e-8
ADAM: learning rate	0.001

Table 4: Hyperparameters common to each of the experiments in Section 4

B.2.4 Probe setup

The data used for training the probes is extracted after the final epoch of training. In case of fully connected layers, the data is simply aggregated and saved as a single data matrix of shape ($samples \times \#neurons$). In case of convolutional layers, this is neither practical nor possible due to limitations in hardware. Alain and Bengio [1] propose Global Pooling or a random selection of features to bypass this issue. However, we are afraid that global pooling the entire feature map can potentially bias the data and thus the probe performance as a result. To mitigate this potential bias, we only adaptive average pool the feature map to ($4 \times 4 \times \#filters$). The reduced feature maps are then flattened into a vector and stored as a data matrix of shape ($samples \times 4^2 \cdot \#filters$). We are aware that this method still is not free of ablation. We study the effects of different downsampling techniques in appendix A. The result show that this approach has a tolerable impact for our purposes, since the visual difference in the structure probe performances is very slim compared to less aggressive downsampling strategies. The training test split remains unchanged from the original data. The probes are logistic regression classifiers minimizing cross entropy using the SAGA solver implementation of scikit-learn. The logistic regression is fitted for 100 epochs.

C Additional Results

In this section, we present additional results and insights from the experiments presented in section 4 and 5.

C.1 Feature map downsampling

The resolution of the feature map has a multiplicative effect on the number of computations required for updating the covariance matrix. Another problem is that the early convolutional layers yield more data points than later layers for computing the covariance matrix, since their feature map is larger. To address both issues, we experiment with downsampling the feature maps using the nearest interpolation. We find that downsampling feature maps such that the resolution of the feature map never exceed 32×32 a pixel did not visibly change the

saturation pattern. We did not apply this method in any of the experiments, since we did not explore the biases induced by this method enough. We include this section only to mention that this a potential path for making the computations more efficient.

C.2 On-line covariance computation and floating point precision

Another issue when on-line computing a covariance matrix is the precision of floating-point values. Neural networks are generally processed in full precision. However, for large amounts of data, the compounding round-off errors induced by the 32-bit precision of the variables may induce errors. For this reason, all computations concerning saturation are performed in double precision. This is also true for the PCA-Layers. Before the update of the covariance matrix is performed, the data is cast in double precision. The running sum, running squares are double precision float arrays as well.

C.3 Effect of eigenspace projections on the reconstruction of a convolutional autoencoder

To visualize the effect of projection into the eigenspace, we train an autoencoder on the Food101 dataset.

C.3.1 Convolutional autoencoder architecture

Encoder	Decoder
$512 \times 512 \times 3$ Input	(3×3) conv, 8 ReLU
(3×3) conv, 16 filters, ReLU	upsampling, nearest, scale-factor 2
(2×2) max pooling, strides 2	(3×3) conv, 8 filters, ReLU
(3×3) conv, 8 filters, ReLU	upsampling, nearest, scale-factor 2
(2×2) max pooling, strides 2	(3×3) conv, 16 filters, ReLU
(3×3) conv, 8 filters, ReLU	upsampling, nearest, scale-factor 2
(2×2) max pooling, strides 2	(3×3) conv, 3 filters, ReLU

Table 5: Convolutional Autoencoder Architecture. All convolutional Layers use same-padding

C.3.2 Hyperparameters

Parameter	Values
Input Resolution	(224×224)
Epoch	50
Batch size	128
Optimizer	ADAM
ADAM: beta1	0.9
ADAM: beta2	0.999
ADAM: epsilon	1e-8
ADAM: learning rate	0.0001

Table 6: Hyperparameters for the convolutional autoencoder.

C.3.3 Reconstruction examples for different values of δ

To test the effects of convolutional and linear projections.

δ	loss	Reconstruction
Original	-	
100%	0.033	
99.9%	0.065	
99.5%	0.089	
99%	0.120	
95%	0.216	
90%	0.234	

Table 7: Reconstruction examples for different values of δ . PCA is applied on all convolutional layers.




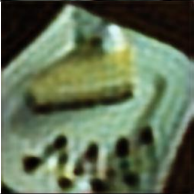
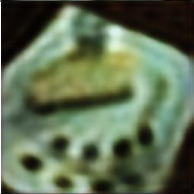


δ	$\dim E_{encoding}^k$	loss	Reconstruction
Original	-	-	
100%	8192	0.033	
99.9%	4374	0.035	
99.5%	1332	0.049	
99%	597	0.062	
95%	17	0.148	
90%	1	0.222	

Table 8: Reconstruction examples for different values of δ . PCA is applied on the fully connected encoding layer.

C.4 Probe performances and saturation patterns of ResNet18 and 34 with disabled skip-connections trained on CIFAR10

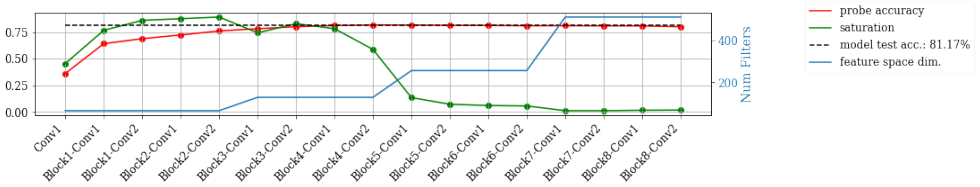


Figure 9: ResNet18 without Skip connections trained on CIFAR10 with 32×32 pixel input resolution

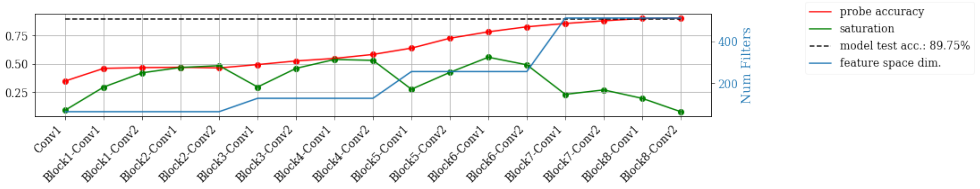


Figure 10: ResNet18 without Skip connections trained on CIFAR10 with 224×224 pixel input resolution

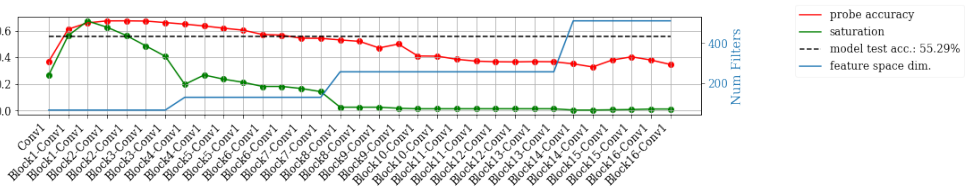


Figure 11: ResNet34 without Skip connections trained on CIFAR10 with 32×32 pixel input resolution

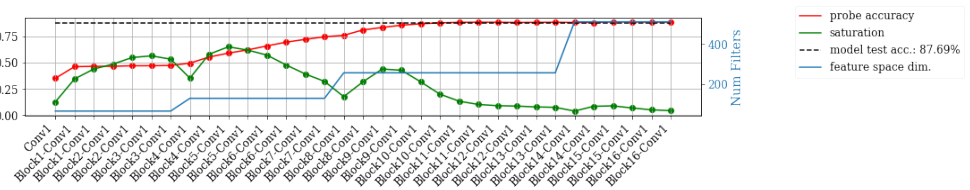


Figure 12: ResNet34 without Skip connections trained on CIFAR10 with 224×224 pixel input resolution

C.5 Probe performances and saturation patterns for TinyImageNet

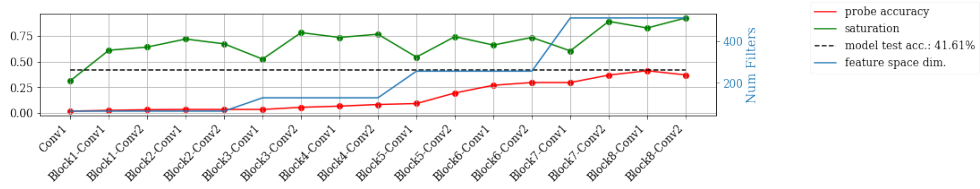


Figure 13: ResNet18 trained on TinyImageNet with 64×64 pixel input resolution

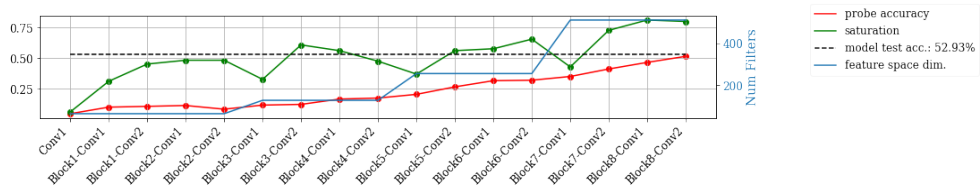


Figure 14: ResNet18 trained on TinyImageNet with 224×224 pixel input resolution

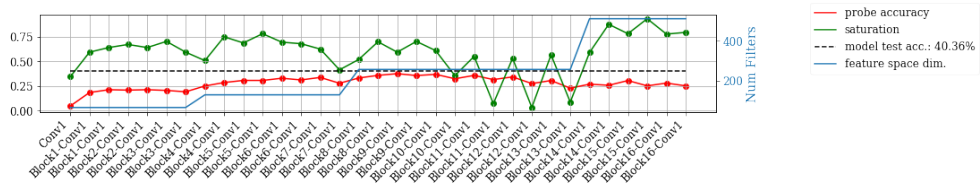


Figure 15: ResNet34 trained on TinyImageNet with 64×64 pixel input resolution

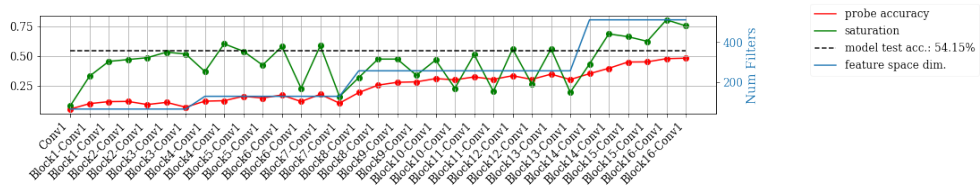


Figure 16: ResNet34 trained on TinyImageNet with 224×224 pixel input resolution

C.6 Probe performances and saturation patterns for MNIST

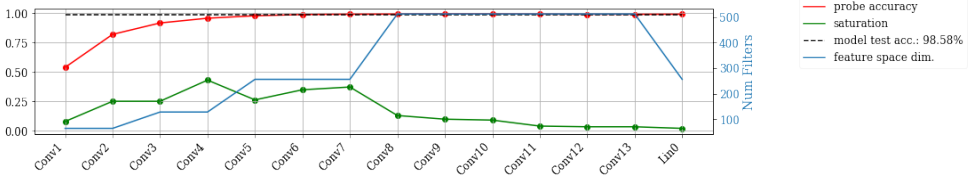


Figure 17: VGG16 trained on MNIST width 32×32 pixel input resolution

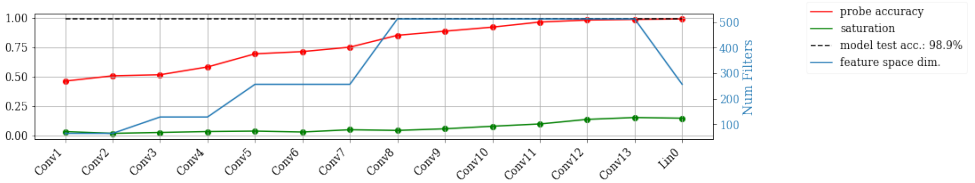


Figure 18: VGG16 trained on MNIST width 224×224 pixel input resolution

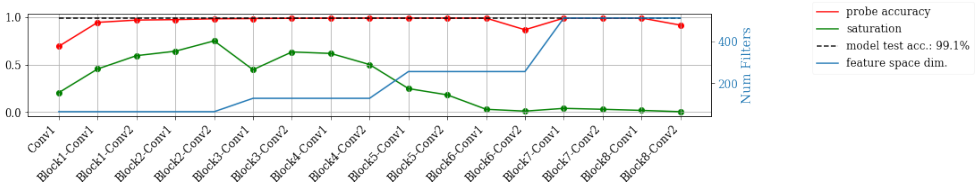


Figure 19: ResNet18 trained on MNIST width 32×32 pixel input resolution

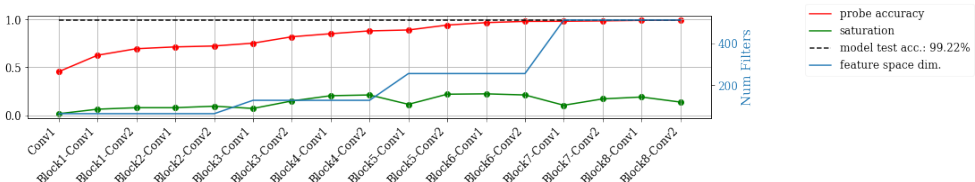


Figure 20: ResNet18 trained on MNIST width 224×224 pixel input resolution

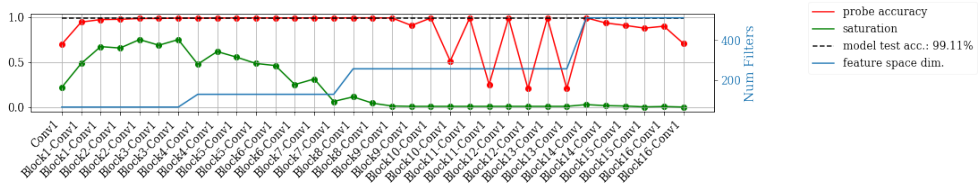


Figure 21: ResNet34 trained on MNIST width 32×32 pixel input resolution

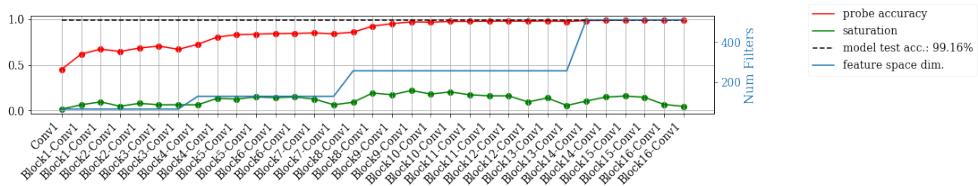


Figure 22: ResNet34 trained on MNIST width 224×224 pixel input resolution

C.7 Collages of VGG and ResNet-style models

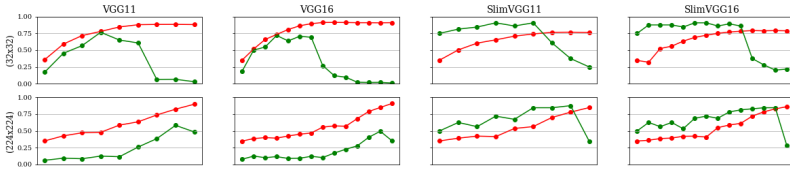


Figure 23: Similar VGG-style models trained on CIFAR10. The model are altered in depth, filter size and input size. Their basic architecture however stays the same. The slim version of the models has all filter sizes reduced by a factor of 8.

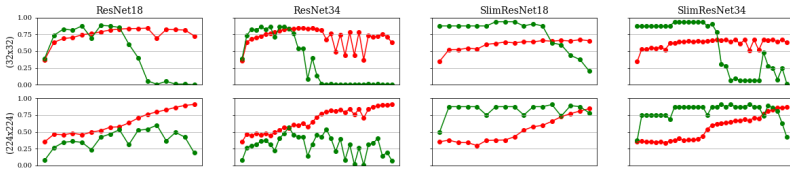


Figure 24: Similar ResNet-style models trained on CIFAR10. The models are altered in depth, filter size and input size. Their basic architecture however stays the same. The slim version of the models has all filter sizes reduced by a factor of 8.

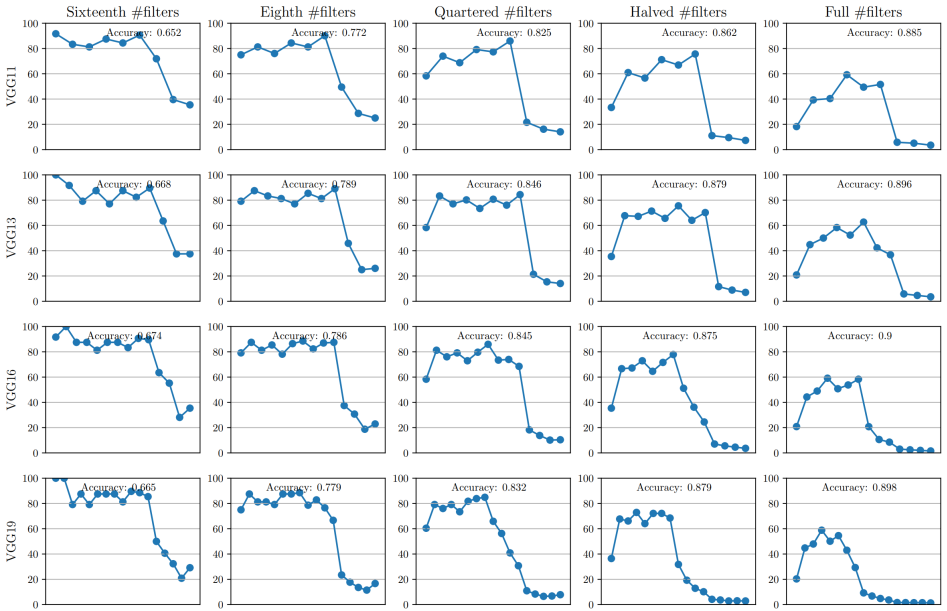


Figure 25: Layer-wise saturation of all tested CNN architectures trained on CIFAR10 for 30 epochs in the configuration for Section 3.3. The input resolution is (32×32) pixels for all models. The layers are represented on the x-axis in the same sequence as the information is propagated through the network at inference time. The y-axis describes the saturation value. Note the gradual change in the distribution while the number of filters and depth increase.

C.8 Full t -Test tables of VGG11, VGG13, VGG16 and ResNet18

Explained σ	Mean difference	sample σ	t-stat	p-value
0.9999	-0.0001	0.0007	-0.714	0.487
0.9998	-0.0005	0.0008	-2.44	0.029
0.9997	-0.0007	0.0012	-2.07	0.058
0.9996	-0.0005	0.0018	-1.1	0.292
0.9995	-0.0010	0.0018	-2.16	0.049
0.9994	-0.0008	0.0016	-1.8	0.094
0.9993	-0.0010	0.0015	-2.72	0.017
0.9992	-0.0013	0.0014	-3.65	0.003
0.9991	-0.0011	0.0018	-2.51	0.025
0.999	-0.0015	0.0021	-2.74	0.016
0.998	-0.0019	0.0029	-2.49	0.026
0.997	-0.0013	0.0034	-1.46	0.166
0.996	-0.0009	0.0039	-0.888	0.390
0.995	0.0005	0.0037	0.537	0.600
0.994	0.0022	0.0038	2.2	0.045
0.993	0.0056	0.0071	3.03	0.009
0.992	0.0114	0.0110	4.03	0.001
0.99	0.0260	0.0179	5.61	0.000
0.98	0.1073	0.0253	16.4	0.000
0.97	0.2824	0.0954	11.5	0.000
0.96	0.4356	0.0767	22	0.000
0.95	0.5118	0.0616	32.2	0.000
0.94	0.5658	0.0568	38.6	0.000
0.93	0.6385	0.0476	52	0.000
0.92	0.7070	0.0510	53.7	0.000
0.91	0.7574	0.0240	122	0.000
0.9	0.7727	0.0090	333	0.000

Table 9: Sum of projections in VGG11 (n=15). $\mu \neq 0$ (p<.01) in **bold**.

Explained σ	Mean difference	sample σ	t-stat	p-value
0.9999	-0.0004	0.0008	-2.42	0.023
0.9998	-0.0005	0.0009	-2.81	0.010
0.9997	-0.0010	0.0010	-5.26	0.000
0.9996	-0.0009	0.0010	-4.92	0.000
0.9995	-0.0011	0.0010	-5.46	0.000
0.9994	-0.0012	0.0012	-4.91	0.000
0.9993	-0.0012	0.0012	-4.83	0.000
0.9992	-0.0013	0.0013	-5.17	0.000
0.9991	-0.0016	0.0015	-5.48	0.000
0.999	-0.0017	0.0016	-5.50	0.000
0.998	-0.0017	0.0022	-3.92	0.001
0.996	-0.0005	0.0030	-0.910	0.371
0.994	0.0037	0.0043	4.45	0.000
0.992	0.0096	0.0062	7.91	0.000
0.99	0.0178	0.0136	6.68	0.000
0.98	0.1123	0.0377	15.2	0.000
0.97	0.2254	0.0578	19.9	0.000
0.96	0.4803	0.1022	24.0	0.000
0.95	0.7026	0.0368	97.3	0.000
0.94	0.7536	0.0227	169	0.000
0.93	0.7654	0.0202	193	0.000
0.92	0.7785	0.0164	242	0.000
0.91	0.7867	0.0143	280	0.000
0.9	0.7929	0.0117	345	0.000

Table 10: Sum of projections in VGG13 (n=26). $\mu \neq 0$ ($\alpha = 0.01$) in **bold**.

Explained σ	μ_{diff}	σ_{sample}	t-stat	p-value	μ_{Sat}	σ_{Sat}	$\mu(\sum dim E_l^k)$
0.9999	-0.0003	0.0008	-2.65	0.011	60.0	0.6	2613 ± 102
0.9998	-0.0006	0.0011	-3.31	0.002	54.5	0.6	2268 ± 97
0.9997	-0.0006	0.0014	-2.82	0.008	51.2	0.7	2071 ± 93
0.9996	-0.0003	0.0016	-1.28	0.208	48.8	0.6	1938 ± 88
0.9995	-0.0001	0.0017	-0.352	0.727	47.1	0.7	1841 ± 86
0.9994	0.0007	0.0019	2.18	0.035	45.6	0.7	1766 ± 84
0.9993	0.0009	0.0022	2.62	0.012	44.5	0.7	1705 ± 83
0.9992	0.0012	0.0031	2.42	0.020	43.4	0.7	1653 ± 80
0.9991	0.0016	0.0032	3.14	0.003	42.5	0.7	1608 ± 79
0.998	0.0107	0.0148	4.57	0.000	36.0	0.7	1318 ± 73
0.996	0.0771	0.0585	8.33	0.000	30.0	0.7	1074 ± 67
0.994	0.1873	0.0812	14.6	0.000	26.3	0.7	934 ± 62
0.992	0.2754	0.0822	21.2	0.000	23.7	0.6	837 ± 58
0.99	0.3643	0.0900	25.6	0.000	21.8	0.6	765 ± 54
0.98	0.6176	0.0413	94.6	0.000	16.1	0.5	556 ± 41
0.97	0.6559	0.0386	107	0.000	13.1	0.4	451 ± 32
0.96	0.7008	0.0384	115	0.000	11.2	0.3	385 ± 27
0.95	0.7351	0.0337	138	0.000	9.8	0.3	339 ± 24
0.94	0.7550	0.0265	180	0.000	8.8	0.2	303 ± 21
0.93	0.7639	0.0231	209	0.000	7.9	0.2	275 ± 19
0.92	0.7727	0.0167	293	0.000	7.2	0.2	252 ± 17
0.91	0.7775	0.0143	344	0.000	6.6	0.2	233 ± 16
0.9	0.7796	0.0127	387	0.000	6.1	0.2	215 ± 15

Table 11: Sum of projections in VGG19 (n=40). $\mu \neq 0$ (p<.01) in **bold**.

Explained σ	μ_{diff}	σ_{sample}	t-stat	p-value	μ_{Sat}	σ_{Sat}	$\mu(\sum dim E_l^k)$
1.0	0.0000	0.0000	nan	nan	100.0	0.0	3904 ± 0
0.9999	-0.0002	0.0012	-0.52	0.611	78.5	0.5	2338 ± 87
0.9998	0.0000	0.0013	0.0796	0.938	75.6	0.4	2153 ± 74
0.9997	-0.0002	0.0016	-0.521	0.610	73.9	0.4	2043 ± 67
0.9996	-0.0009	0.0020	-1.66	0.119	72.5	0.4	1963 ± 63
0.9995	-0.0005	0.0022	-0.813	0.430	71.3	0.4	1900 ± 61
0.9994	-0.0006	0.0019	-1.18	0.256	70.4	0.3	1847 ± 57
0.9993	-0.0007	0.0019	-1.48	0.162	69.4	0.4	1802 ± 55
0.9992	-0.0007	0.0022	-1.29	0.217	68.7	0.4	1763 ± 54
0.9991	-0.0006	0.0022	-1.13	0.279	67.9	0.4	1728 ± 52
0.998	0.0031	0.0046	2.63	0.020	62.7	0.3	1493 ± 40
0.996	0.0213	0.0285	2.9	0.012	57.4	0.4	1294 ± 32
0.994	0.0389	0.0454	3.32	0.005	54.0	0.5	1181 ± 28
0.992	0.0579	0.0596	3.76	0.002	51.3	0.5	1100 ± 27
0.99	0.0812	0.0782	4.02	0.001	49.2	0.6	1038 ± 26
0.98	0.1899	0.1042	7.06	0.000	41.7	0.7	841 ± 26
0.97	0.2918	0.1057	10.7	0.000	37.0	0.7	731 ± 26
0.96	0.3649	0.0834	16.9	0.000	33.6	0.6	654 ± 25
0.95	0.4333	0.0757	22.2	0.000	30.9	0.6	595 ± 24
0.94	0.4544	0.0667	26.4	0.000	28.6	0.6	548 ± 24
0.93	0.4787	0.0668	27.7	0.000	26.7	0.6	508 ± 24
0.92	0.4896	0.0638	29.7	0.000	25.1	0.6	475 ± 23
0.91	0.5119	0.0582	34	0.000	23.6	0.6	446 ± 22
0.9	0.5296	0.0574	35.8	0.000	22.4	0.6	421 ± 21

Table 12: Sum of projections in ResNet18 (n=15). $\mu \neq 0$ (p<.01) in **bold**.

D Tail Patterns on various Architectures and Datasets

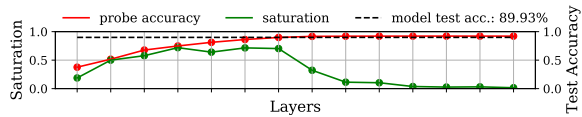
In this section, we will provide additional tail patterns that were observed during experiments. The black vertical bar in some of these plots marks the first layer with the receptive field size of the input greater than the input resolution. We find that this property predicts unproductive sequences of layers well for sequential architecture like the VGG-network family but not when more than one pathway is present (for example skip or dense connections). The experiments use the same experimental setup described in appendix section 4.3.

D.1 Different Types of Tail Patterns - A brief explanation

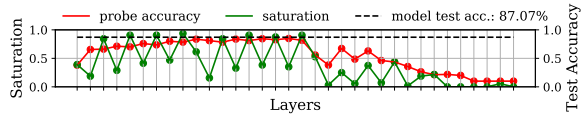
We find that saturation is subject to noise induced by certain features of the neural architecture like the increase or decrease in filters from layer to layer, the use of 1×1 convolutions and downsampling layers are common culprits for zig-zag-like behavior or sudden dips and spikes in saturation, an example for the latter is DenseNet18 in figure 26 (b). It has to be stressed that these factors are not random or create non-reproducible perturbations. Instead, they usually result in anomalous patterns that are very stable over multiple runs (which is exemplified in Section 4.1).

Logistic regression probes are considerably more robust against the aforementioned properties. However, they are influenced by the path the information takes during the forward pass, revealing different *types* of tail patterns that can be differentiated based on the processing in the tail-layers. The three examples found commonly are exemplified in figure 26. These examples also give insights into how neural networks process information differently, which is the main reason why we dedicate an additional section to these findings in the appendix. All the networks are trained on Cifar10 using a 32×32 pixel input resolution. In figure 26 (a) we find a pass-through tail, where the layers process the information but do not advance the quality of the intermediate solution. We find this type of tail pattern is typical for sequential neural networks (which you can see from other results in appendix C.4, C.6) and C.5). The second type of tail, depicted in figure 26 (b), is caused by the multiple pathways inside the DenseBlock of DenseNet. Information can pass from any previous layer to the current layer within the DenseBlock, effectively allowing the information to skip layers. When layers are skipped, the intermediate solution quality degrades and instantaneously recovers after the skipped section is over. The latter is apparent in the depicted example by the high model performance relative to the probe performance of the last DenseBlock layers. This phenomenon was initially observed on a simple MLP-example by Alain and Bengio [14]. If necessary, the signal may jump more than a single building block in the architecture. An example of which can be seen in figure 26 (c) on a ResNet34 architecture. This jumping is indicated by the zig-zag-pattern in the probe performance, where the higher performing layer resembles the first and lower performing layer the second layer of a residual block.

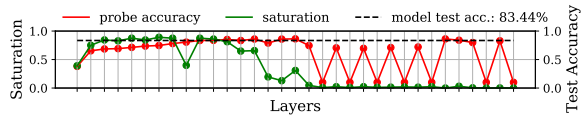
This shows that architecture decisions, influencing the potential pathway’s information can take from input to output, can have a significant influence on the way the model processes (or chooses not to process) information. In any case, the semantic of the tail-pattern remains unchanged, since a skipped layer and an unproductive layer can both be considered a parameter and computational inefficiency.



(a) VGG16 tail layers maintain the quality of the intermediate solution



(b) The tail of DenseNet18 shows a decay in probe performance, indicating that the last DenseBlock is skipped entirely [10].



(c) ResNet34 skips most residual blocks in the tail, which is apparent by the zig-zag pattern in probe performances caused by the starts and end of skip-connections [10].

Figure 26: Depending on the neural architecture, tail patterns may deviate in their appearance in probe performance. In sequential architectures (a) the layers maintain the quality of the intermediate solution. If shortcut connections exist in the architecture, layers may be *skipped*. Skipped layers are apparent by their decaying probe performance [10]. This is apparent on DenseNet18 (b) and ResNet34 (b) where a single DenseBlock and multiple ResidualBlocks are skipped respectively. All models are trained on Cifar10 at native resolution.

D.2 VGG11, 13, 16, 19 - MNIST

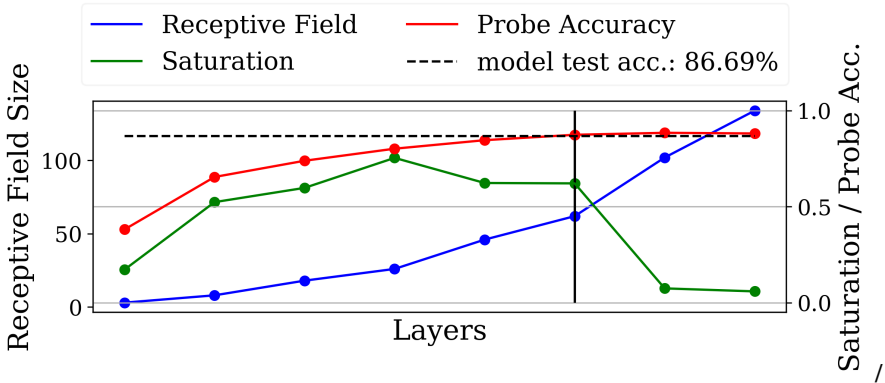


Figure 27: VGG11 - Mnist - 32×32 input resolution.

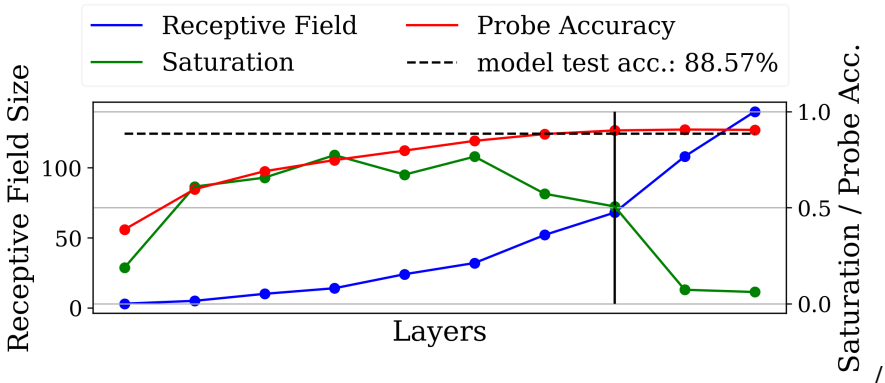


Figure 28: VGG13 - Mnist - 32×32 input resolution.

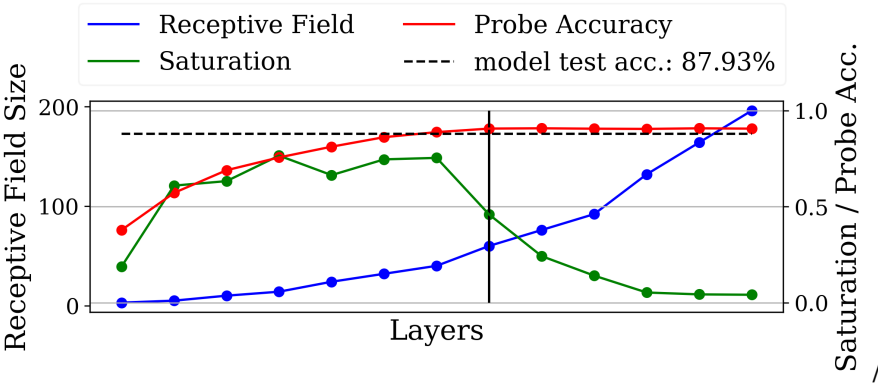


Figure 29: VGG16 - Mnist - 32×32 input resolution.

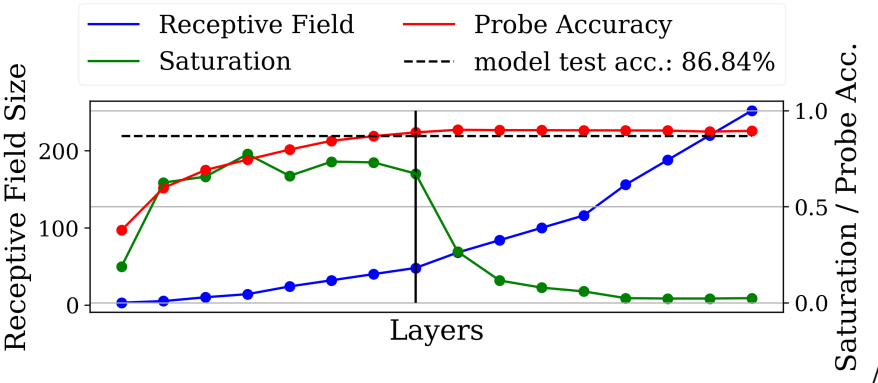


Figure 30: VGG19 - Mnist - 32×32 input resolution.

D.3 VGG11, 13, 16, 19 - TinyImageNet

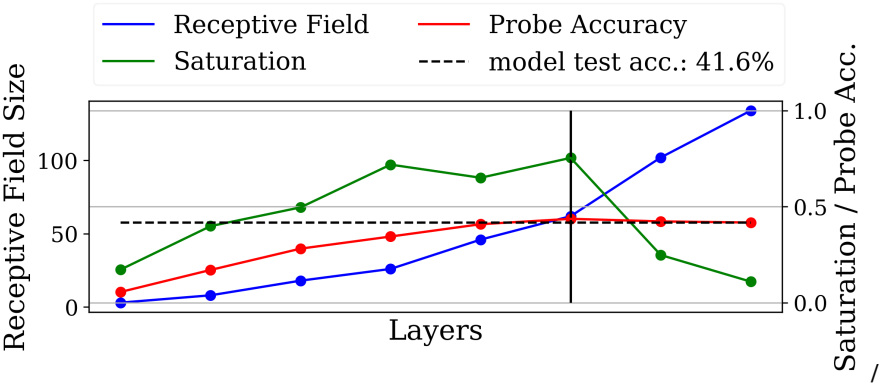


Figure 31: VGG11 - TinyImageNet - 32×32 input resolution.

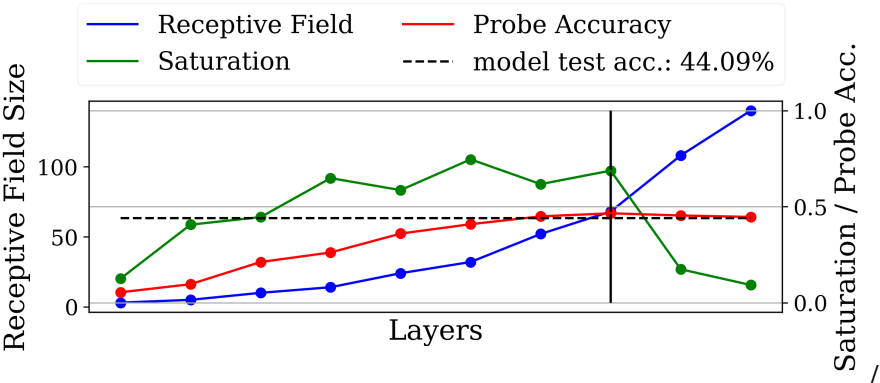


Figure 32: VGG13 - TinyImageNet - 32×32 input resolution.

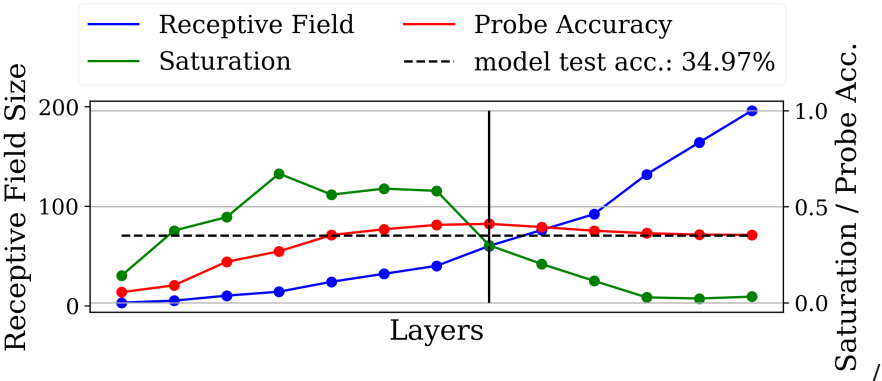


Figure 33: VGG16 - TinyImageNet - 32×32 input resolution.

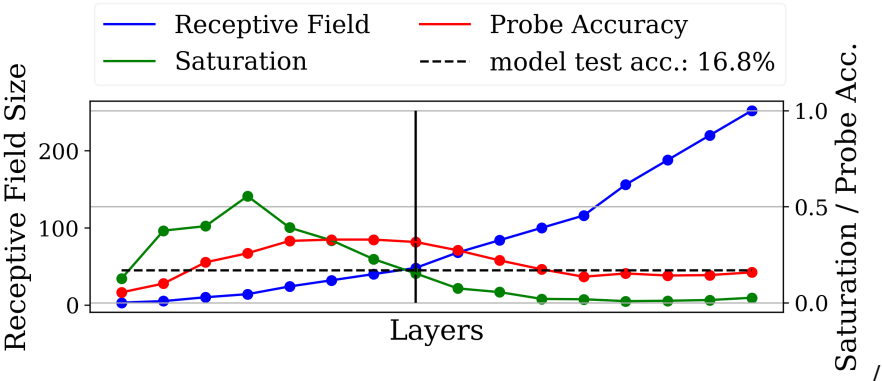


Figure 34: VGG19 - TinyImageNet - 32×32 input resolution.

D.4 DenseNet18, 65 - Cifar10

Interestingly, the skipping behavior observable in too deep ResNet-style architectures is not present in DenseNet-style networks. Instead, the probe accuracy degrades over entire regions of the network, indicating that these are likely skipped entirely.

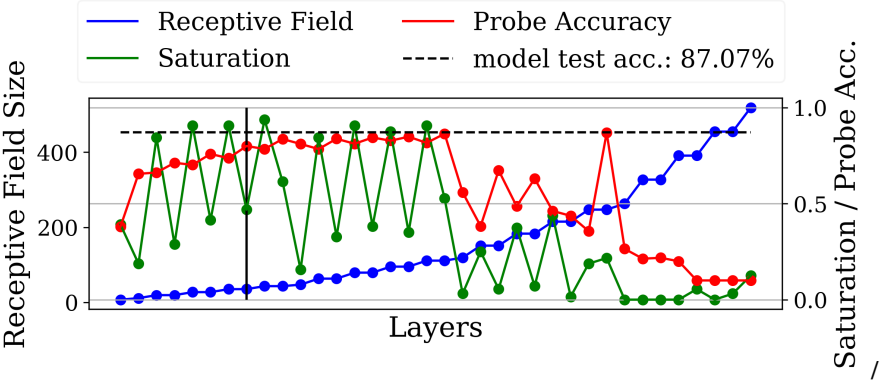


Figure 35: DenseNet18 - Cifar10 - 32×32 input resolution.

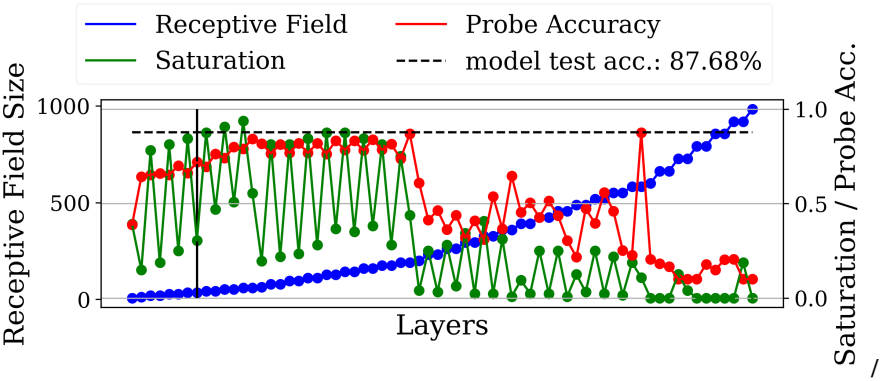


Figure 36: DenseNet65 - Cifar10 - 32×32 input resolution.

D.5 ResNet50 - Cifar10

Tail patterns are present in ResNet 50.

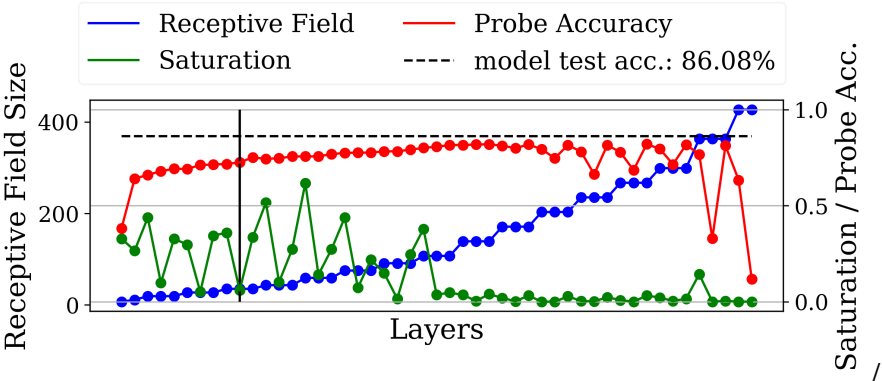


Figure 37: ResNet50 - Cifar10 - 32×32 input resolution.

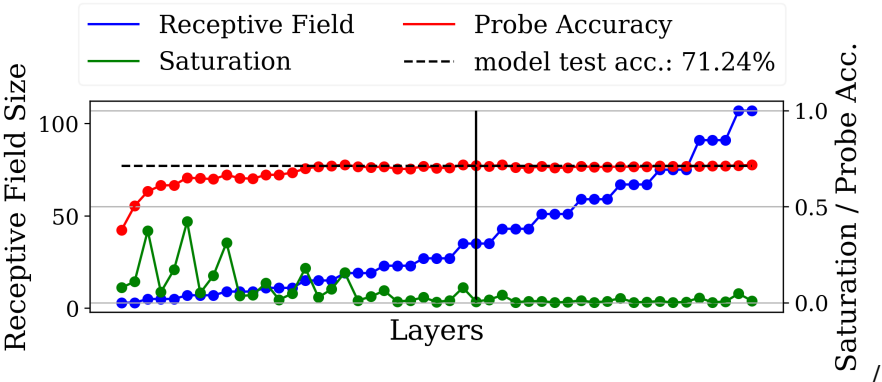
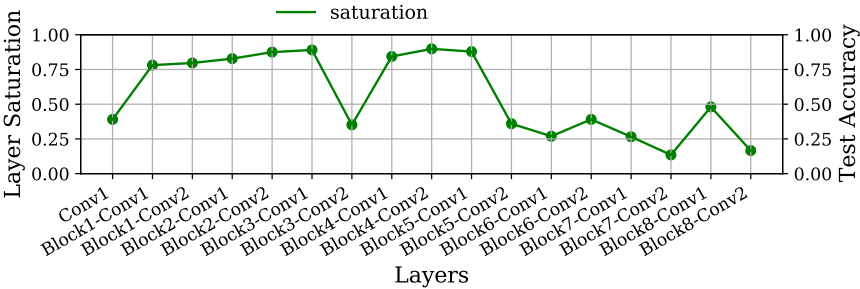


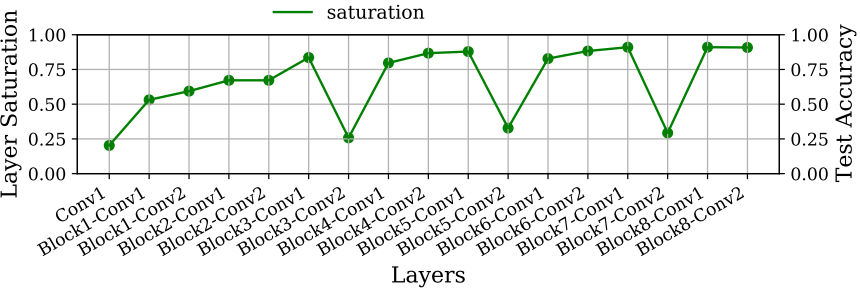
Figure 38: ResNet50 (removed stem) - Cifar10 - 32×32 input resolution.

D.6 Experiments on ImageNet and iNaturalist

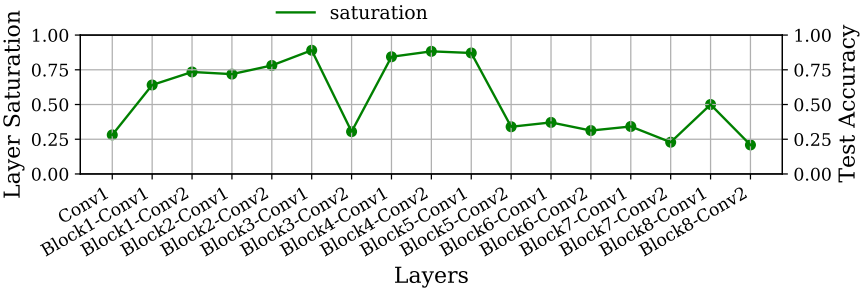
This set of experiments is an attempt to recreate the tail pattern phenomenon on ImageNet and iNaturalist. For these experiments, computing probe performances was not feasible due to resource limitations. For this reason, only saturation is provided. Each model is trained two times. Once on the design resolution of 224×224 pixels of the respective models (for reference purposes, we do not expect to see a tail pattern at this resolution) and once on 32×32 pixels, which reliably results in tail patterns for these models.



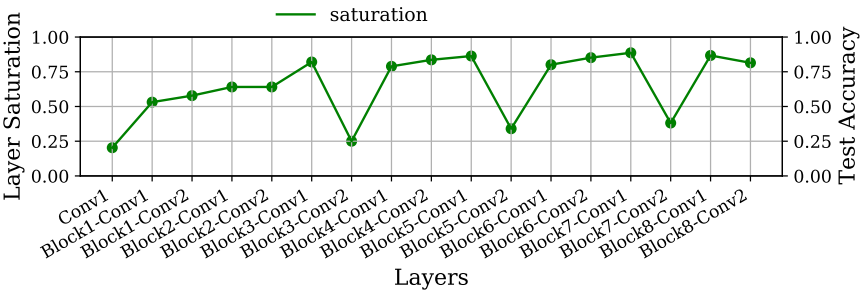
(a) ResNet18 - ImageNet - 32×32 - Test Accuracy: 25.88%



(b) ResNet18 - ImageNet - 224×224 - Test Accuracy: 65.63%

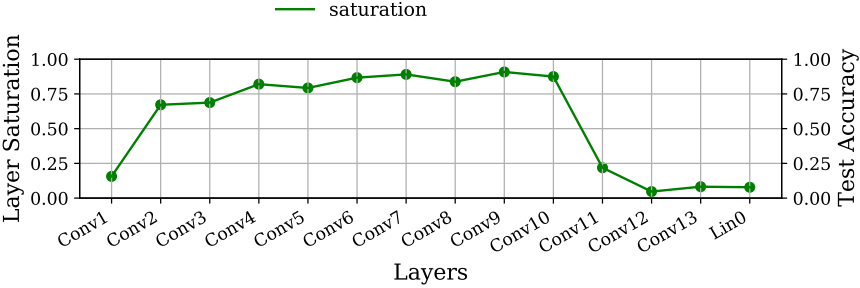


(a) ResNet18 - iNaturalist - 32×32 - Test Accuracy: 12.15%

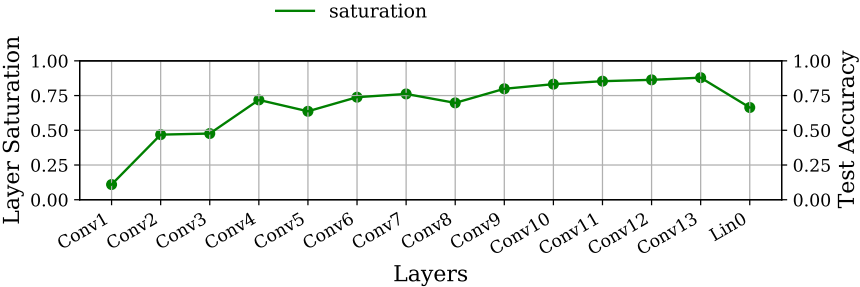


(b) ResNet18 - iNaturalist - 224×224 - Test Accuracy: 39.91%

Figure 39: ResNet18 trained on iNaturalist.

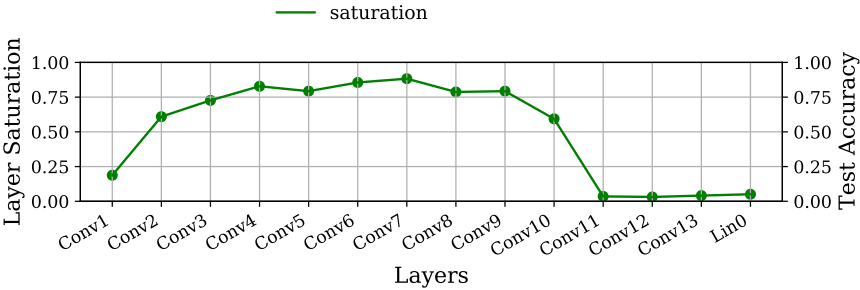


(a) VGG16 - ImageNet - 32×32 - Test Accuracy: 10.13%

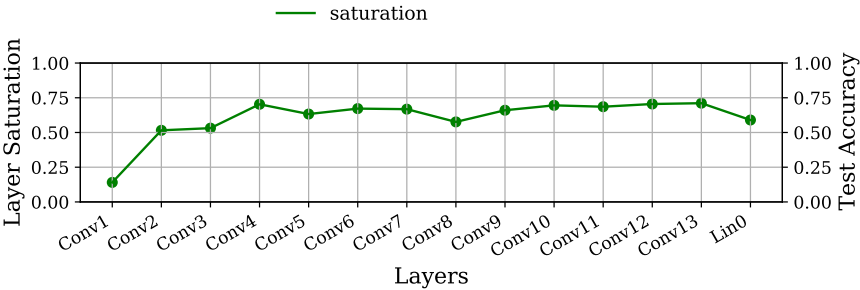


(b) VGG16 - ImageNet - 224×224 - Test Accuracy: 63.96%

Figure 40: VGG16 trained on ImageNet.



(a) VGG16 - iNaturalist - 32×32 - Test Accuracy: 17.85%



(b) VGG16 - iNaturalist - 224×224 - Test Accuracy: 52.11%

Figure 41: VGG16 trained on iNaturalist.

E Source Code

The experiments conducted in this work are done in two distinct repositories. The experiments themselves are conducted with the `phd-lab`-repository, which can be found here (including a manual): <https://github.com/MLRichter/phd-lab>.

The second repository is called *delve* and contains the logic for PCA-Layers (see section 3), on-line covariance approximation, as well experiment control. All three features are used by `phd-lab` to conduct the experiments in question. This project is currently in the process of being open sourced, is installable over PyPi and can be found here: <https://github.com/delve-team/delve>.