A study of AdaHessian: local minima and normalization

Florence Osmont, Haochen Su, Benoît Müller
EPFL – Optimization for Machine Learning – june 2023

***Abstract*—We aim to study some optimization properties of the algorithm AdaHessian, a second-order stochastic optimization algorithm.**

We use AdaHessian and Stochastic gradient descent(SGD) to solve an image classification task. We compare the accuracy and the type of the local minimum as we change the initialization and the optimization methods. We also test the impact of normalization on the convergence of the second order method.

We measure significant differences for the local minimum reached by AdaHessian and SGD. We found that SGD does not really escape from the result optimized by AdaHessian, but the converse is true. Normalization still has some effect on the optimization with a loss 2.5 times smaller, and influences the accuracy with a change of 9.18%.

I. Introduction

We aim to study some optimization properties of the algorithm AdaHessian[1], a second-order stochastic optimization algorithm that adaptively estimates the Hessian, in order to incorporate the curvature of the loss.

In its simplest form, a second-order method has for purpose to take information from curvature, and adapt the direction and size of the step, in order to accelerate convergence by reducing the number of iterations needed to converge. In addition to their high per-iteration cost, second-order methods are observed to result in bad accuracy.

The assertion of AdaHessian is that their method tends to resolve those problems and achieves new state-of-the-art results by a large margin as compared to other adaptive optimization methods. In particular, it says to solve the problem of normalization, so we compare the convergence and accuracy with and without normalization. In order to understand better the behavior of the method and the type of result it returns, we first initialize stochastic gradient descent (SGD)[2] with the model returned by AdaHessian, and vice versa, in order to see if the optimization continues and how. We also train with AdaHessian and SGD on a pre-trained network to see the impact of this initialization compared to a random one. We finally do a comparison of the models obtained through three different metrics.

II. Models and Methods

A. Dataset

We use the CIFAR10 dataset provided by PyTorch ([3]) which contains 50000 train samples and 10000 test samples. Each sample is a 32x32 color image attached to one of the 10 labels (plane, car, bird, cat, deer, dog, frog, horse, ship, truck). Different labels are mutually exclusive and each has 5000 images for training and 1000 images for testing.

Concerning normalization, since AdaHessian asserts that the estimation of the curve solves the problem of normalization, we artificially modify the normalization of the image channels and see if it impacts convergence or accuracy. We multiply the standard deviation of the three channels RGB by respectively 0.0001, 1, and 1000.

B. Model

We use ResNet18[4] as our model, which is an 18-deep-layers convolutional neural network. It contains blocks with convolution layers, batch normalization layers, max pooling layers and uses ReLu[5] as the activation function. Instead of simply stacking layers, ResNet18[4] explicitly lets the layers fit a residual mapping. The skip connections help to address the problem of vanishing and exploding gradients, which makes it easier to train and gain higher performance. After the residual blocks, the model follows with average pooling and a fully connected layer to do the classification.

C. Algorithms

We use two different optimizers. In a first time, we train our network with each of them and a random initialization controlled by the same seed. In a second time, we initialize the network with the weights obtained from one method and then train again with the other. We also trained using each method and the initialization of the pre-trained network provided by Pytorch.

1) SGD: The algorithm is presented in appendix*Stochastic Gradient Descent*, which is a classic first order approach.

2) AdaHessian: The algorithm is presented in appendix *AdaHessian*. Second order optimization algorithms have multiple advantages compared with first order. They give extra information on the curvature of an objective function that adaptively estimates the step-length of optimization trajectory during training in a neural network. They converge more rapidly and need less hyperparameter adjustment. But their main drawback is the expensive computational cost. Adahessian reduces this factor by a fast Hutchinson based method to approximate the curvature matrix, a root-mean-square exponential moving average to smooth out variations of the Hessian diagonal, and a block diagonal averaging to reduce the variance of Hessian diagonal elements.

D. Hyperparameters

The hyperparameters used can be found in Table I.

The momentum quantifies how much the algorithm takes

Method name	lr	Batch size	Epochs	Momentum	Betas	Weight decay	Hessian power
SGD	0.01	256	360	0.95	(0.9,0.999)	0.0005	-
SGD-pretrained	0.01	256	360	0.95	(0.9,0.999)	0.0005	-
AdaHessian	0.15	256	160	-	(0.9,0.999)	0.0005	1
AdaHessian-pretrained	0.15	256	160	-	(0.9,0.999)	0.0005	1

TABLE I: Hyperparameters of our methods

into account the last step-sizes to infer the new one. The weight decay quantifies the L2-Ridge regularization, which tends to decay the amplitude of weights to avoid over-fitting. The Hessian power is a special AdaHessian internal parameter, which is the power applied to the estimated second order term. We use 1 so it doesn't have an effect, and it is tested to have the best performance on our dataset and model.

We use k-fold validation on our training set to find the best hyperparameters and models (k=5). Then we test the models on an independent test set to evaluate their performance.

E. Methods to measure the similarity of two different networks

To compare the algorithms, we want to compare not only their accuracy but the local minima they attain. Hence, we want to compare their trained networks. To do so, we need to choose some similarity metrics. Looking at the work of [6], we could see that none performs overwhelmingly better than the other, and they sometimes contradict each other. Indeed, we want the measures to find two networks to be similar when they should be (specificity) but not when they shouldn't (sensitivity). We choose to use the two most popular ones CKA (Centered Kernel Alignment) and CCA (Canonical Correlation Analysis) and the more classical Orthogonal Procrustes distance (OPD). The CKA measure is defined in [7] with the purpose of identifying correspondences between representations in networks trained from different initializations. It has good specificity but bad sensitivity. In opposition, CCA has good sensitivity but a bad specificity. Finally, they showed in [6] that the orthogonal Procrustes distance seems to be doing better on average.

To compare the networks, we want to compare their layers. To do this, we compare their activation on the same dataset. Consider n data points and two layers with p_1 and p_2 neurons respectively. Let $A \in \mathbb{R}^{p_1 \times n}$ and $B \in \mathbb{R}^{p_2 \times n}$ be the matrix of neurons activation. The distance is then defined as follows ([6]).

*a) **Linear CKA** :*

$$d_{CKA}(A,B) = 1 - \frac{\|AB^T\|^2}{\|AA^T\|^2 \|BB^T\|^2}$$

*b) **CCA** :* Find the orthogonal bases (w_A^i, w_B^i) for two matrices such that after projection onto w_A^i, w_B^i, the projected matrices have maximally correlated rows. Then compute the ith canonical correlation coefficient ρ_i as:

$$\rho_i = \max_{w_A^i, w_B^i} \frac{\langle {w_A^i}^T A, {w_B^i}^T B \rangle}{\|{w_A^i}^T A\| \|{w_B^i}^T B\|}$$

such that $\langle {w_A^i}^T A, {w_A^i}^T A \rangle = \langle {w_B^i}^T B, w_A B^{iT} B \rangle = 0$ for $j < i$. They are used to define different scalar measures. We use the CVCAA variant in the implementation.

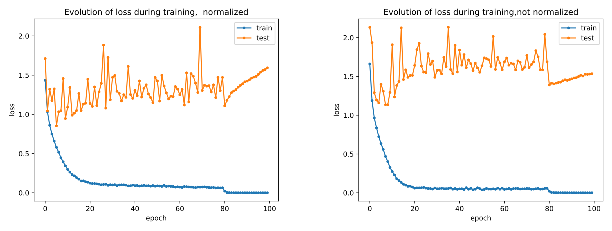
*c) **OPD**:* It corresponds to finding the solution to the problem consisting of finding the rotation of A minimizing the distance to B in the Frobenius norm.

$$d_{proc}(A,B) = \|A\|^2 + \|B\|^2 - 2\|AB^T\|$$

We use for our implementation the library [8] for CVCCA and OPD and the library [9] for Linear CKA. We used all the test samples of CIFAR10 for testing CKA and the first 20 for the two other measures. This difference is due to the implementation of CKA by [9] that is more optimized.

III. Results

First, we got for each method the accuracy presented in the table II. The graph of loss and accuracy evolving with epochs in figure4 and figure3. The full results for the comparison between the networks are in the appendix in the figures 5, 6, 7 respectively for the CKA, OPD and CVCCA measures (as it was the less meaningful metric, there are less for CVCCA). Note that the convention for the layers was different for the two libraries. The label AdaHessian_SGD means the network is first trained with AdaHessian and then with SGD (and vice versa). We present in Figure 2 only the most interesting graphics for the CKA measure, as it is the one that handles the best the change of initialization.



(a) normalized (b) not normalized

Fig. 1: Effect of normalization

Method name	Test accuracy	Training time
SGD	0.7829	$25.48 \pm 0.22s$
SGD-pretrained	0.8309	$25.48 \pm 0.22s$
AdaHessian	0.7844	$44.02 \pm 0.25s$
AdaHessian-pretrained	0.8060	$44.02 \pm 0.25s$

TABLE II: Test accuracies and training time per iteration

First, we can see that for any initialization and optimizer, the first layers tend to be more similar than the last ones, thus the last are more specific. Moreover, they are also very different from each other inside the same network. It is the

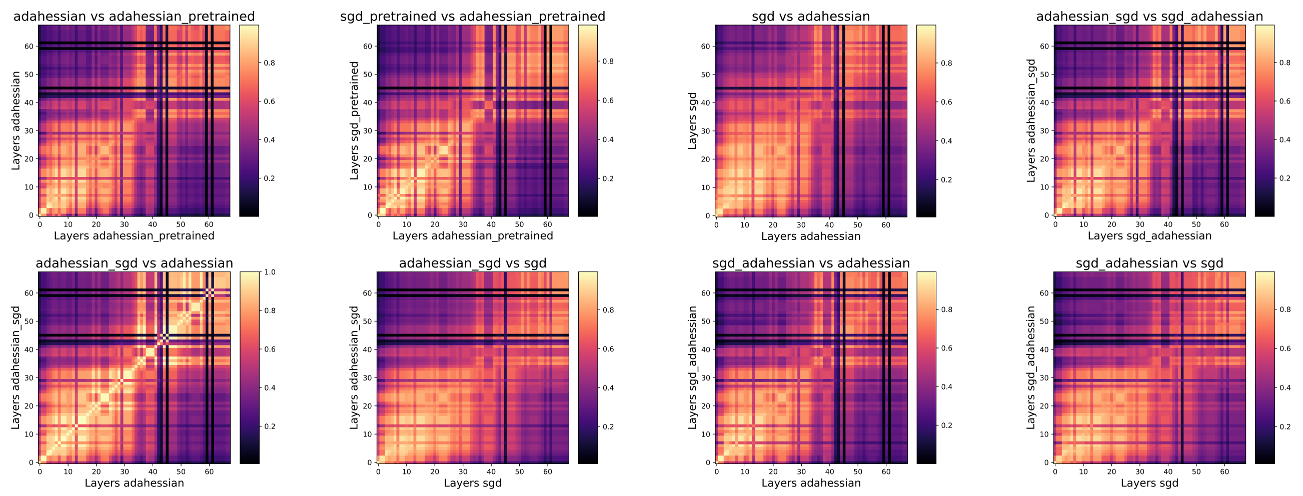


Fig. 2: Some comparison for the CKA measure

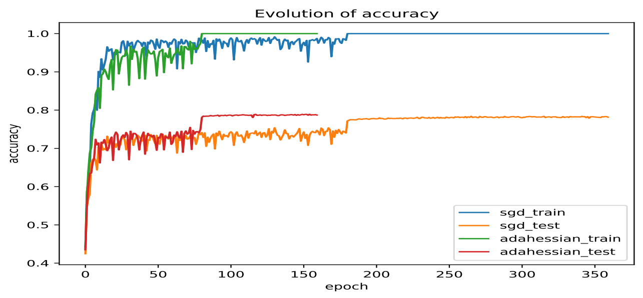


Fig. 3: Evolution of the accuracy with respect to the epochs

most marked dissimilarity, as it is highly present in all three measures. The next results are mostly seen from the CKA measure tested with more data.

As we could expect, using the pretrained version of the network reduces the difference made by the choice of the optimizer. We can interpret that as having an initialization already close to a local minimum, in particular for the first layers that are identical.

Next, we can see that some layers have 0 similarities with the others in both the CKA and OPD metrics. It corresponds to the first convolutional layer of the third block for SGD and to the first convolutional layer of the third and fourth blocks for AdaHessian. We visualize the first layer3.1Conv2d in the two models in 8 and 9. Looking at them in more detail, we could see that they are not trivial to analyze. The visualization makes sure that our training goes normally and that this effect is caused by different properties of the optimizers.

Finally, we see that AdaHessian_SGD and SGD_AdaHessian give different weights, except for the first layers. AdaHessian_SGD gives similar weights as AdaHessian alone, showing that SGD cannot escape from the local minima given by AdaHessian. SGD_AdaHessian is different from SGD and AdaHessian. Thus, the local minimum obtained by SGD is not optimal and is escaped by AdaHessian to reach another.

Concerning the effect of normalization, we plot the evolution of the loss in Figure 1. We see that visually the overwhole convergence is very well preserved. However, with normalization, the accuracy is 72.93%, and it is 63.75% without, so normalization does help to increase accuracy. We also note from a pure optimization point of view that with normalization the final train loss is only 7.998e-05, but it is 2.037e-4 without it. This shows that the normalization helps to actually minimize a bit more the loss, and in our case, this resulted also in better accuracy on a test set so a better generalization. Even though we see some influence, it must be said that it is nothing compared to the very bad and unstable convergence properties of gradient descent for ill-conditioned problems. So, we conclude that normalization is well handled by the second order, but it stays a good practice to obtain the best results possible.

IV. Discussion

As discussed in the [7] paper, there is actually no absolute and perfect similarity measure. They do not always agree, because they were created using different intuitions. This active field would be an interesting perspective to investigate more. Properties such as norm, effective rank, sparsity, and eigenspectrum of linear layers, could give interesting information and should be compared with the similarity norms that we have used. Also, a study of the samples that are misclassified could exhibit differences between first and second order algorithms.

V. Summary

We have shown that the last layers are more sensitive to changes and more specific than the first ones. In particular, some of them are completely dissimilar from the others. Moreover, we obtained different properties for the local minimum reached by AdaHessian and SGD. Then, we could see that even for second order method, normalization has still some effect on optimization and accuracy, but it guarantees a stable convergence.

References

[1] Z. Yao, A. Gholami, S. Shen, M. Mustafa, K. Keutzer, and M. W. Mahoney, "Adahessian: An adaptive second order optimizer for machine learning," 2021.

[2] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[3] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[5] A. F. Agarap, "Deep learning using rectified linear units (relu)," 2019.

[6] F. Ding, J.-S. Denain, and J. Steinhardt, "Grounding representation similarity with statistical testing," 2021.

[7] S. Kornblith, M. Norouzi, H. Lee, and G. Hinton, "Similarity of neural network representations revisited," 2019.

[8] R. Hataya, "anatome, a pytorch library to analyze internal representation of neural networks," 2020. [Online]. Available: https://github.com/moskomule/anatome

[9] A. Subramanian, "torch cka," 2021. [Online]. Available: https://github.com/AntixK/PyTorch-Model-Compare

Appendix

Algorithm 0.0 The stochastic gradient descent algorithm; in our implementation the learning rate γ is updated using MultiStepLR

```
Input: initial weights $w^{(0)}$, number of iterations $T$
Output: final weights $w^{(T)}$
1.  for $t = 0$ to $T - 1$
2.      estimate $\nabla\mathcal{L}(w^{(t)})$ using a subset of train set
3.      compute $\Delta w^{(t)} = -\nabla\mathcal{L}(w^{(t)})$
4.      select learning rate $\gamma$
5.      $w^{(t+1)} := w^{(t)} + \gamma\Delta w^{(t)}$
6.  return $w^{(T)}$
```

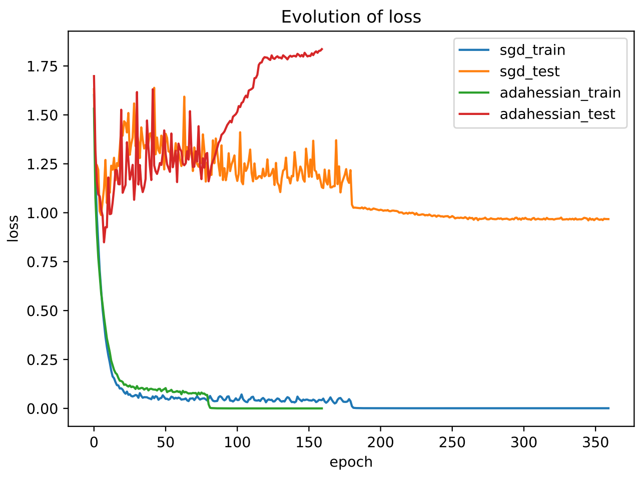


Fig. 4: Loss evolving with epochs graphs

Algorithm 0.0 The AdaHessian algorithm; in our implementation the learning rate γ is updated using MultiStepLR

```
Input: initial weights $w^{(0)}$, learning rate $\gamma$, exponential decay rates
       $\beta_1$, $\beta_2$, Block size $b$, Hessian Power $k$, number of iterations $T$
Output: final weights $w^{(T)}$
1.  set $m_0 = 0$, $v_0 = 0$
2.  for $t = 0$ to $T - 1$
3.      $g_t \leftarrow$current step gradient
4.      $D_t \leftarrow$current step estimated diagonal
5.      Compute $D_t^{(s)}$ based on
            $$D^{(s)}[ib + j] = \frac{1}{b}\sum_{k=1}^{b} D[ib + k]$$
        for
            $$1 <= j <= b, 0 <= i <= d/b - 1$$
6.      Update $\bar{D}_t$ based on
            $$\bar{D}_t = \sqrt{\frac{(1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}D_i^{(s)}D_i^{(s)}}{1-\beta_2^t}}$$
7.      Update $m_t$ and$v_t$ based on
            $$m_t = \frac{(1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i}g_i}{1-\beta_1^t}, v_t = (\bar{D}_t)^k$$
8.      $w^{(t)} \leftarrow w^{(t-1)}$-$\gamma m_t/v_t$
9.  return $w^{(T)}$
```

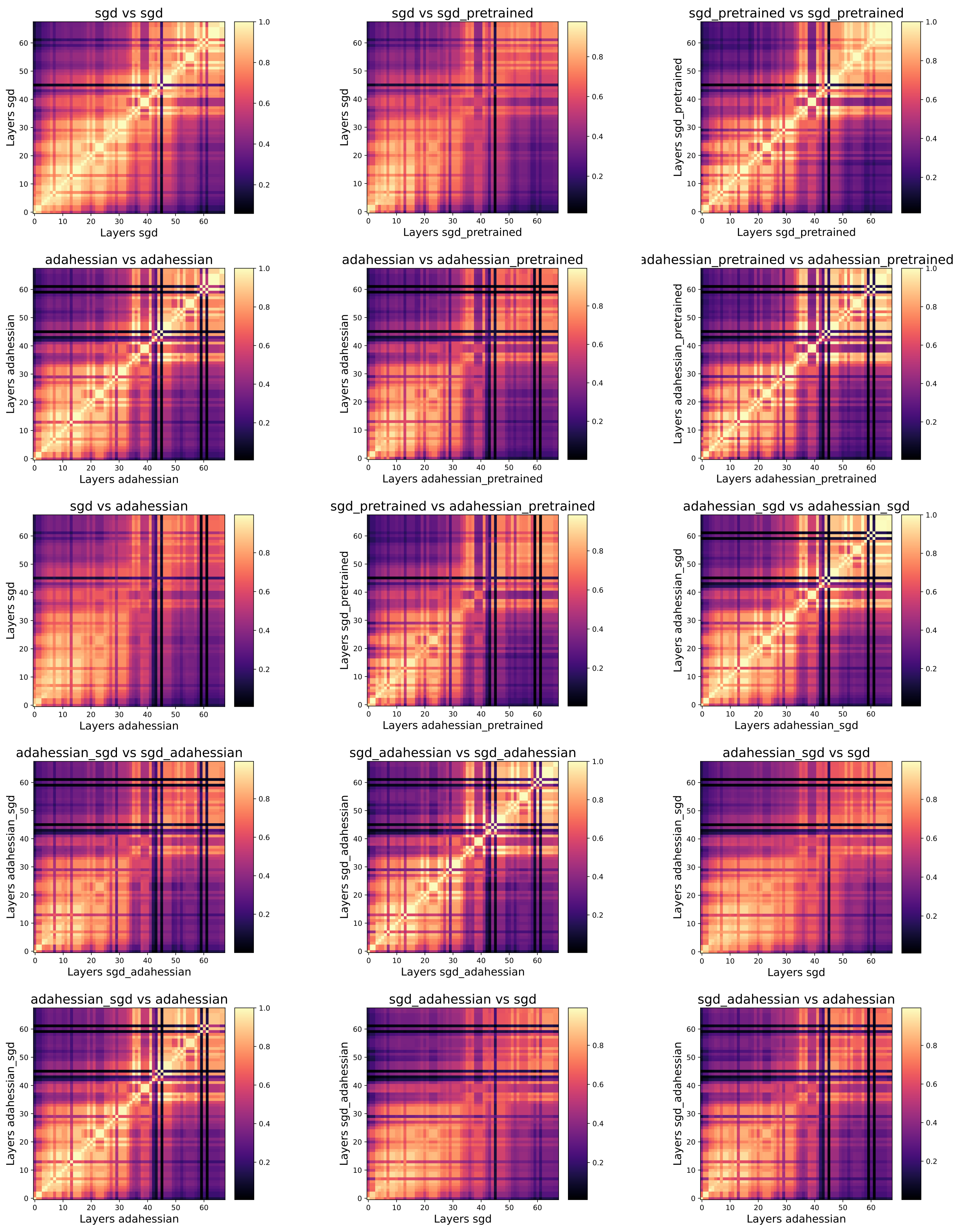


Fig. 5: Comparisons with CKA

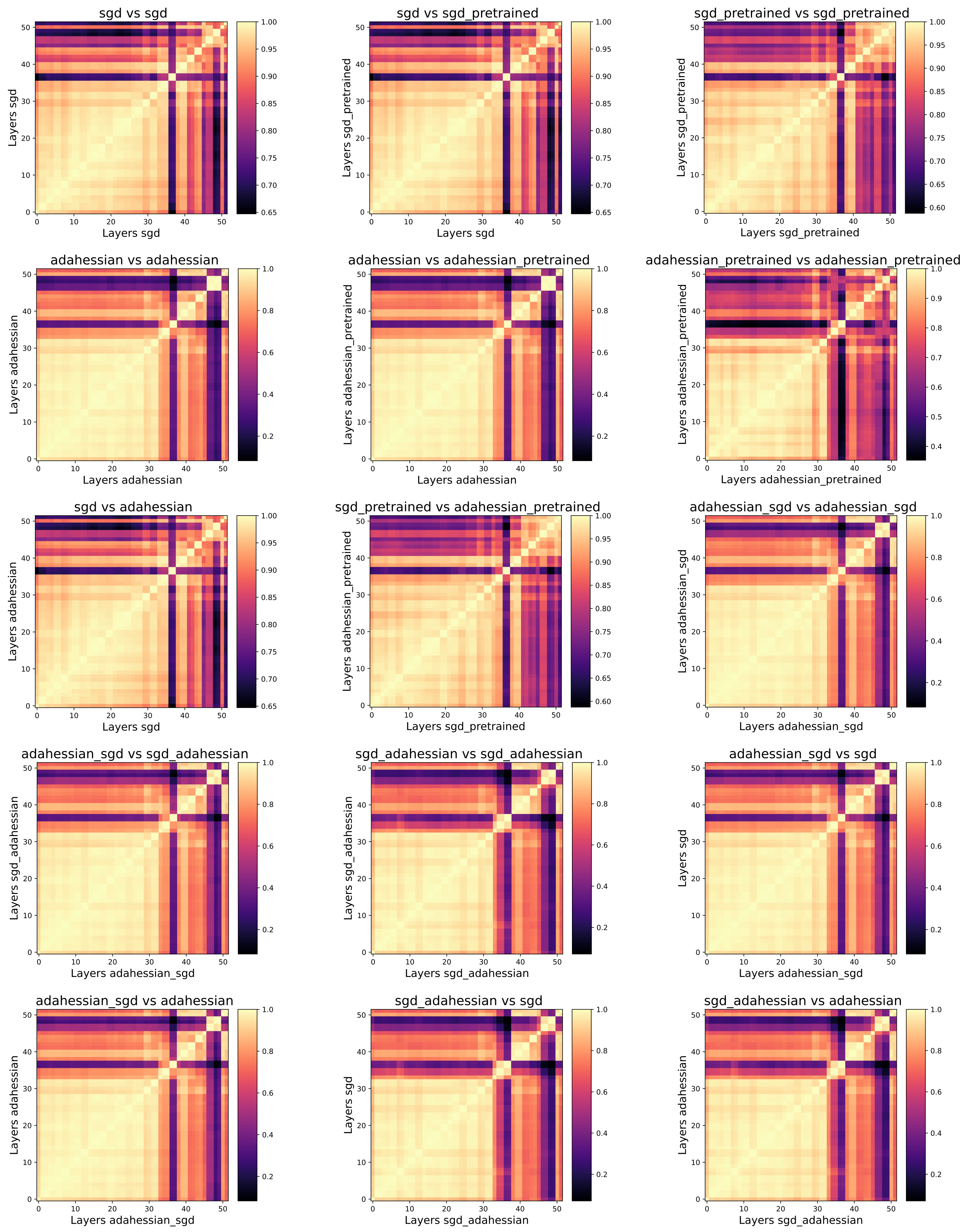


Fig. 6: Comparisons with OPD

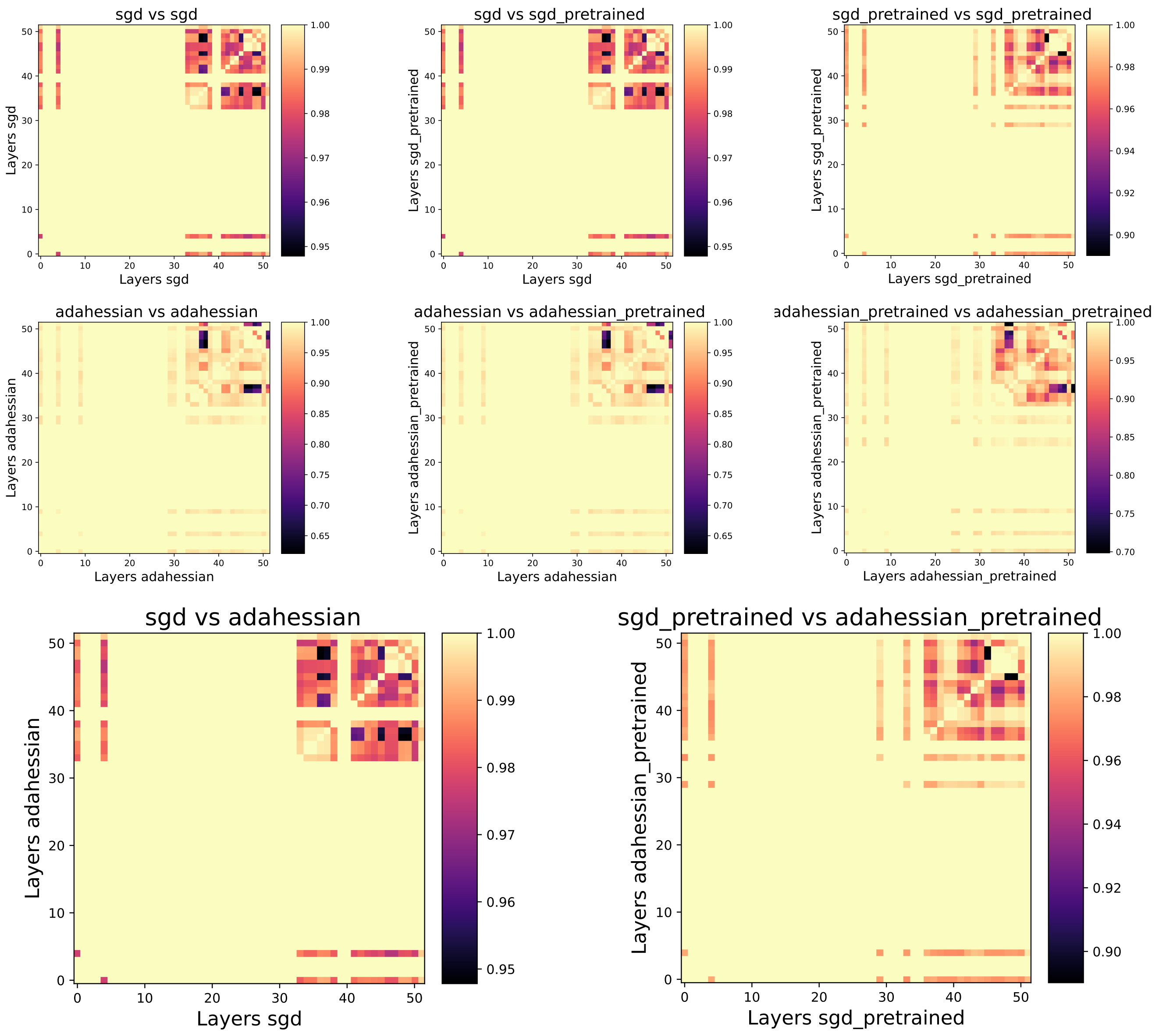


Fig. 7: Comparisons with SVCCA

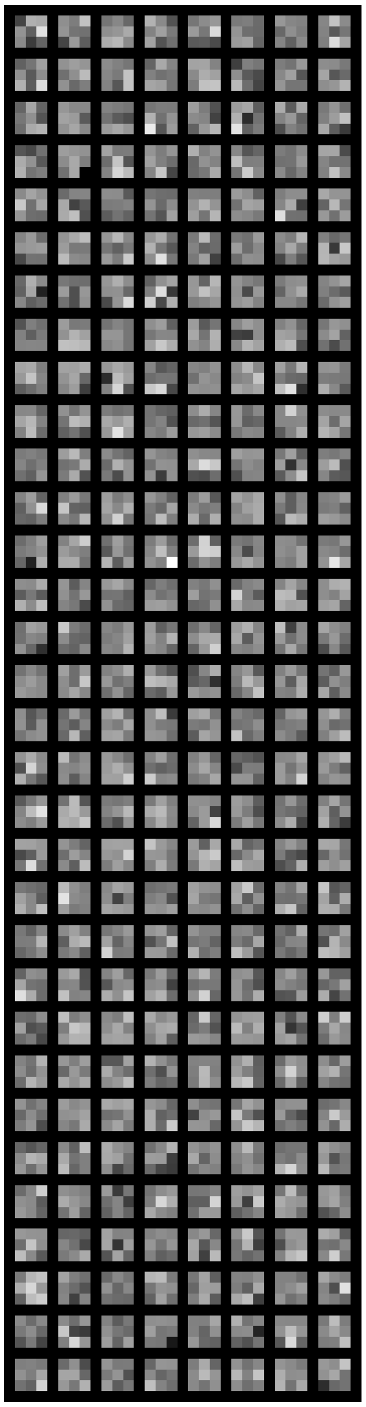


Fig. 8: Visualizations of convolution filters in layer3.1Conv2d trained with sgd

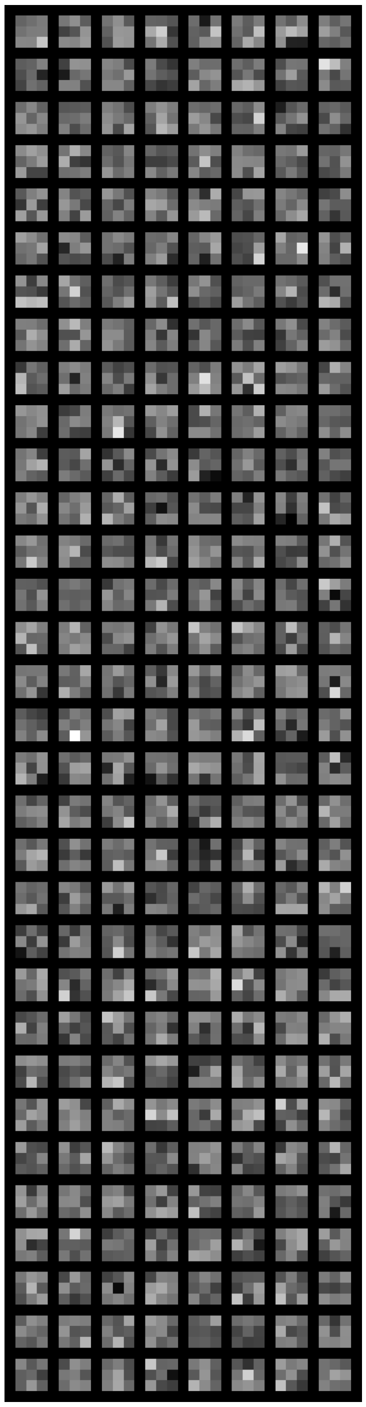


Fig. 9: Visualizations of convolution filters in layer3.1Conv2d trained with adahessian