# Behavior and generalisation capabilities of various optimising techniques for Image Classification using Convolution Neural Networks

Baffou Jeremy, Lozhkina Arina, Francis Clément
*CS-439 Optimization for Machine Learning - Project*

***Abstract*—In this project, we implemented 4 optimizers algorithms: SGD, SGD with momentum, RMSprop and ADAM and investigated the impact of them as well as its learning rate for solving a classification problem on multiple datasets: CIFAR10, CIFAR100, MNIST, FashionMNIST. We implemented the testing on the augmented data by Gaussian and salt and pepper noise and affine translation in order to demonstrate the generalization capacities of optimizers. The best results: VALUE accuracy on validation, were obtained with ALGORITHM with learning rate. As a result of our experiments, we also found THAT....**

## I. Introduction

Defining the most suitable optimising method among several different algorithms for a specific problem can be identified as optimisation benchmarking. Such benchmarking can be very helpful for model designers as well as model users in a wide range of real-world applications. Indeed, the choice of the optimizer and its tuning is critical in the final performances and generalization capabilities of the model. Thus being able to find the best optimization algorithm for a given task is a highly relevant topic.

However, benchmarking can be difficult to produce properly in order to have meaningful results outside of the specific task studied. Several questions arise from the definition of generalization to the reproducibility of the pipeline. As a matter of fact, the method selection will depend on the test settings, the computing environment and the performance aspect considered to be the most important (speed, memory or accuracy...).

In this paper, we studied the generalization performances of four optimizers on four different reference data sets in image classification and investigate the behaviors of the different algorithms to try to explain these differences.

## II. Background

### A. Datasets

We based our analysis on reference datasets in the field of image classification: MNIST, FashionMNIST, CIFAR10 and CIFAR100. Each of them consist of 60000 images of dimension $28 \times 28$ pixels. MNIST has grayscale digits from 0 to 9 and FashionMNIST images are grayscale fashion objects. The CIFAR datasets are composed of various real world RGB colored images such as animals or vehicles. The datasets do not solely differ in their content but also in their number of output classes, even though they all have perfect class balance. MNIST, FashionMNIST and CIFAR10 have 10 classes, where as CIFAR100 has 100 classes. The training and testing repartition for the different datasets can be found in table II.

We augmented the tests using the following data augmentations operations:

- Random Vertical and horizontal flipping: a probability of 0.5 to apply each flip independently.
- Gaussian noise with a kernel $3 \times 3$ and standard deviation $\sigma = 0.2$.
- Linear transformations: random rotation of $\pm 20$ degrees, random translation of $\pm 20\%$ of the width and/or height, scaling of $\pm 15\%$.

For each dataset, the resulting test sets are composed of: the original one, and one additional set of the original size for each augmentation type.

### B. Generalization Capability

We decided to base our analysis on the generalization capabilities of the different optimizers. By this we mean the ability of an optimizer to train models with good accuracy on images similar to what it has been trained on, but that it has never seen previously. Thus we used the test sets of the different data sets for this purpose, but also we augmented the test datasets to see if the filters and features learned by our model were robust to some pertubations such as noise or affine transforms. Note that we do not have augmented our training datasets because we wanted especially to see if the choice of the optimizer could results in a more robust learning, without being exposed to such transformations.

### C. Model Architecture

The model used as a root for the study is a Visual Geometry Group network (VGGNet). It takes the $28 \times 28 \times C$ images and has $N$ outputs units, where $C$ is the number of channel and $N$ the number of classes. Each output unit $O_i$ codes the probability that the image belongs to class $i$.

The model is composed of a successions of Convolutional, Batch Normalization and Rectified Linear Unit (ReLU) layers with 5 Maxpool Layers uniformly distributed between them. The output obtained after the last MaxPooling layer is given as input to a fully connected layer combined with a softmax output unit.

The intuition behind the VGGNet architecture is that the succession of Convolutionnal layers learn features that are then fed to the linear layer that acts as a standard classifier.

The model architecture is represented in 1 where the filters used are of size $3 \times 3$ (with a padding of 1) for the convolution layers and $2 \times 2$ (no padding) for the MaxPool layers.

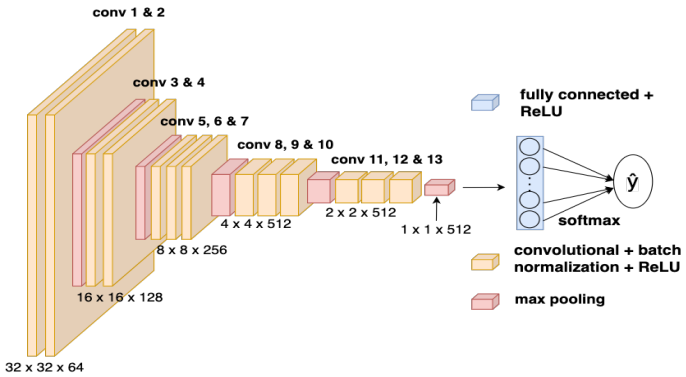


Figure 1: Illustration of the VGG architecture

### D. Training Criterion

Regarding the training part of the model we used BackPropagation on the CrossEntropy loss, which is presented in 1.

$$l(x, y) = \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n} \mathbf{1}\{y_n \neq \text{ignore\_index}\}} l_n \in \mathbb{R}^N \quad (1)$$

With

$$l_n = -w_{y_n} \log \frac{\exp x_{n,y_n}}{\sum_{c=1}^{C} \exp x_{n,c}} \mathbf{1}\{y_n \neq ignore\_index\} \quad (2)$$

Where $x_n$ is the input vector, $y_n$ the target one-hot output vector, $N$ is the batch size (1024 in our case) and $C$ is the number of classes.

### E. Selected Machine Learning Optimizers

In this study we compared four different optimizers in their generalization capabilities and behaviors. We define the cost function as $J(\theta)$, where $\boldsymbol{\theta}$ is the weight vector.

*1) Stochastic Gradient Descent (SGD):* The Stochastic Gradient Descent algorithm aims to solve the main problem of the classical Gradient Descent algorithm which is that it uses the whole training set to compute the gradient. This structure makes it very slow for large training sets. In contrast, SGD randomly picks a subset of the training set whose size is determined by the batch size. On the other hand, this stochastic characteristic will induce random fluctuations in the loss minimization process.
The SGD core step can be expressed as in 1 where $i$ is a randomly selected subset, $\nabla_{\boldsymbol{\theta}} J^i(\boldsymbol{\theta}))$ is average gradient of $J(\boldsymbol{\theta})$ over the datapoints in $i$.

**Algorithm 1** SGD algorithm

1: $\theta \leftarrow \theta - \eta \nabla_\theta J^i(\boldsymbol{\theta})$

Therefore, the optimizer has one hyper-parameter:

- The learning rate $\eta$

*2) Momentum Stochastic Gradient Descent (MSGD):* This optimizer is a variant of the Gradient Descent method. MSGD uses previous gradients to smooth the fluctuations. This mechanism acts as an exponential filter on the gradient steps, where $\rho$ controls the strength of the averaging. The core of the momentum algorithm is presented in 2.

**Algorithm 2** Momentum algorithm

1: $\mathbf{m} \leftarrow \rho \mathbf{m} + \nabla_\theta J^i(\boldsymbol{\theta})$
2: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \mathbf{m}$

Therefore, this optimizer has 2 hyper-parameters:

- The learning rate $\eta$
- The momentum $\rho$

*3) Root Mean Squared Propagation (RMSProp):* The idea of RMSProp is to scale down the gradient vector along the steepest dimensions in order to correct its direction earlier to point to the optimum. As seen in the first line of 3, the algorithm accumulates the squares of the most recent iteration gradients; it does so by using exponential decay. The second step of the algorithm is almost identical to the Gradient Descent, but with one big difference: the gradient vector is scaled down by a factor of $\sqrt{\mathbf{s} + \epsilon}$. The symbol $\oslash$ represents the element-wise division and the $\otimes$ the element wise multiplication.

**Algorithm 3** RMSProp algorithm

1: $\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_\theta J(\boldsymbol{\theta}) \otimes \nabla_\theta J(\boldsymbol{\theta})$
2: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_\theta J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \epsilon}$

Therefore, the optimizer has 3 different hyper-parameters:

- The learning rate $\eta$
- The exponential decay factor $\beta$
- The smoothing term $\epsilon$ to avoid division by zero

*4) Adaptive Moment Estimation (ADAM):* This algorithm combines to ideas of the momentum optimization and the RMSProp. Just like momentum optimization, it keeps track of an exponentially decaying average of past gradients; and like RMSProp, it keeps track of an exponentially decaying average of past squared gradients. In algorithm 4, $t$ represents the iteration number. By looking at steps 1, 2 and 5; the close similarity to both momentum optimization and RMSProp is clearly noticeable. Steps 3 and 4 are of implementation detail: since $\mathbf{m}$ and $\mathbf{s}$ are initialized at 0, they will be biased toward 0 at the beginning of training. These two steps help boosting $\mathbf{m}$ and $\mathbf{s}$ at the beginning of the training.

**Algorithm 4** Adam algorithm

1: $\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1 - \beta_1) \nabla_\theta J(\boldsymbol{\theta})$
2: $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_\theta J(\boldsymbol{\theta}) \otimes \nabla_\theta J(\boldsymbol{\theta})$
3: $\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$
4: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$
5: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \epsilon}$

Therefore, the Adam optimizer has 4 different hyper-parameters:

- The learning rate $\eta$
- The exponential decay factors $\beta_1$ and $\beta_2$
- The smoothing term $\epsilon$ to avoid division by zero

## III. Methods and Implementation

In this section we develop the ideas and methods that we used to have consistency in the training in order to have reliable comparisons, and to perform our analysis of generalization and behavior.

*1) Implementation Choice:* Due to the large number of combination of datasets and optimizer ($4 \times 4 = 16$), we decided to only investigate the effect of learning rate, and to fix the other parameters of the various algorithms (decaying parameter = 0.9, damping coefficient = 1e-8, momentum = 0.9, ADAM decaying parameters = (0.9, 0.999)). We tested for 7 different learning rates ranging from $10^{-5}$ to 1, which makes a total of $16 \times 7 = 112$ trainings. This choice has impacted the performance of the algorithm and we will come back at it in the discussion section.

### A. Pipeline Description

In order have reliable results we apply a common pipeline to each pair of dataset-optimizer.

First, to ensure reproducibility, we assign a seed to all the random number generator used during model initialization and training (c.f. the github for extensive description). Then we split the training set in a validation and real training set with ratio 10%,90% respectively. Then we run a training with a fixed learning rate during 100 epochs for the CIFAR datasets and 70 epochs for the MNIST datasets. We have less epochs in the second case as, in general, the 30 extra epochs did not improve the best validation accuracy. The batch size was fixed at 1024. Furthermore every fourth training steps, we collect the weights of the first CNN layer and the linear layer in order to proceed to the behavior analysis.

When the maximum epoch is reached, we save the weights which have given the best validation accuracy.

### B. Benchmarking and Comparison Criteria

In order to rank the different optimization algorithms, we used the accuracy as our metric.

$$Accuracy = \frac{NumberofCorrectPredictions}{TotalNumberofPredictions}$$

For further investigation, confusion matrices could be studied in order to have a finer resolution of the results of our models.

### C. Behavior Analysis

The use of accuracy as the only factor to discriminate the different algorithms, in a generalization perspective, is weak. Thus we performed extensive analysis of the impact of the choice of the optimizer on both the weights trajectories and the frequency content of the parameters evolution. This will help us to gain extra insight in the generalization capabilities of the optimizer under study.

*1) Weights Trajectory:* The weights of our network lie in a high dimensional space, which makes direct visualization impossible. Thus dimensionnality reduction techniques must be performed in order to graphically inspect the parameters trajectories over time. For computational resources purpose, we separated our analysis on the first CNN layer and the linear linear of our model. We first perform PCA to reduce to 50 components the layer weights-evolution matrix $W^l_{evol}$, which is $M \times N$, where $M$ is the number of times the weights have been collected and $N$ the weights'layer dimensionnality. Then we applied T-SNE on the PCA result to finally map the weights into a 2d and 3d embedded space, following the guidelines in [1].

*2) Frequency Analysis:* High fluctuations in our learning process can result in sub-optimal solutions or slow convergence. The analysis of the frequency of changes in the parameters can partly characterize the ability of given algorithm to have a smooth learning. Based on the layer weights-evolution matrix, we performed Fast Fourier Transform (FFT) for each single weight over time. Then we aggregated the weights fourier coefficients by taking their module and sum all of them for each time frequency. It gives an approximation of the amplitude of the frequency content of the layer at each frequency. We then inspected this frequency content plot, but also computed the ratio of high frequency defined as $\frac{\sum_{i=10}^{N} A_i}{\sum_{j=1}^{N} A_j}$, where $A_i$ is the aggregated fourier coefficient module and the threshold of 10 has been selected empirically.

## IV. Results and Discussion

### A. Results

The different accuracies reached with the best models can be seen in tables I, IV, V, VI. The optimizer with the best accuracy over all the test sets is MSGD. We can see in figure 2 an example on training vs validation accuracy and indeed MSGD achieves the best validation accuracy. Thus, based on the accuracies, the ranking would be MSGD, SGD, ADAM and finally RMSProp.

| Dataset | SGD | MSGD | RMSProp | ADAM |
|---|---|---|---|---|
| CIFAR10 | 83.020 | **84.890** | 81.480 | 84.440 |
| CIFAR100 | 42.690 | **51.730** | 39.780 | 38.140 |
| MNIST | 99.470 | **99.570** | 99.520 | 92.570 |
| FashionMNIST | 93.330 | 93.120 | 92.220 | **99.300** |

Table I: Accuracy on the standard test set

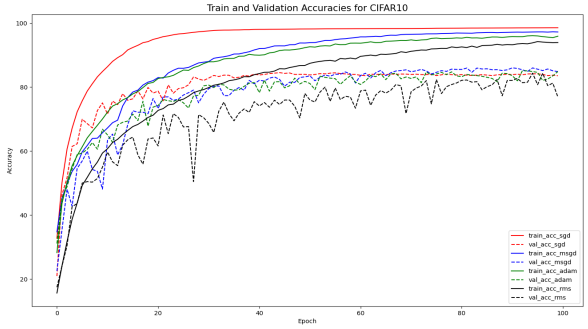


Figure 2: Training vs validation accuracy on CIFAR10

### B. Discussion

Although we have sought to reduce the stochastic factors from our analysis of the work, as well as conducting enough image classification experiments, it would be incorrect to generalise unambiguously about our results. First, due to computational capacity constraints, we were not able to perform experiments with a wide variety of models and optimizer parameters. Second, technical problems led to the fact that fixing the initialization of weights was not always implemented successfully. As a result, the work mainly demonstrates the impossibility of a universal algorithm for each problem, as well as the regularities between the learning rate and the model performance.

The main problems we identify are sub-optimal hyper-parameters other than the learning rate, which are likely to have affected the learning results, and inaccuracies in the augmentation, such as flips for MNIST data, which affected the semantics of the image and resulted in reduced accuracy not because of the optimiser, but because of the meaning of the image

## V. Further Work

For further work on the subject we would first try to tune more precisely all the hyperparameters to have a more realistic tuning of the optimizers. We implemented but did not effectively try to start from different initial weights to see which optimizer was the least prone to weight initialization and thus in a way generalize better.

## VI. Conclusion

## VII. Appendix

### A. Tables

| | Training | Validation | Testing | Augmented Testing |
|---|---|---|---|---|
| MNIST | 54000 | 6000 | 10000 | 40000 |
| FashionMNIST | 54000 | 6000 | 10000 | 40000 |
| CIFAR10 | 54000 | 6000 | 6000 | 24000 |
| CIFAR100 | 54000 | 6000 | 6000 | 24000 |

Table II: Data-Set Splitting

| Dataset | SGD | MSGD | RMSProp | ADAM |
|---|---|---|---|---|
| CIFAR10 | 3e-1 | 2e-1 | 1e-3 | 1e-3 |
| CIFAR100 | 5e-1 | 3e-2 | 1e-4 | 1e-4 |
| MNIST | 5e-1 | 1e-1 | 1e-3 | 1e-3 |
| FashionMNIST | 5e-1 | 1e-1 | 1e-3 | 1e-4 |

Table III: Best learning rates

| Dataset | SGD | MSGD | RMSProp | ADAM |
|---|---|---|---|---|
| CIFAR10 | **67.860** | 58.560 | 56.140 | 61.670 |
| CIFAR100 | 32.330 | **33.220** | 32.070 | 29.280 |
| MNIST | 96.040 | 98.570 | 98.770 | **98.960** |
| FashionMNIST | **87.340** | 86.160 | 83.520 | 86.170 |

Table IV: Accuracy on the noisy test set

| Dataset | SGD | MSGD | RMSProp | ADAM |
|---|---|---|---|---|
| CIFAR10 | 53.040 | **56.610** | 50.080 | 56.330 |
| CIFAR100 | 18.510 | **22.440** | 14.960 | 14.360 |
| MNIST | 72.200 | 74.610 | 75.120 | **79.800** |
| FashionMNIST | 47.180 | 46.480 | **52.630** | 49.050 |

Table V: Accuracy on the affine transformed test set

| Dataset | SGD | MSGD | RMSProp | ADAM |
|---|---|---|---|---|
| CIFAR10 | 58.810 | **60.560** | 60.500 | 59.790 |
| CIFAR100 | 28.300 | **35.180** | 26.540 | 25.840 |
| MNIST | 56.290 | 56.460 | 56.990 | **57.460** |
| FashionMNIST | 54.010 | 51.430 | **54.210** | 53.920 |

Table VI: Accuracy on the flipped test set

| Dataset | SGD | MSGD | RMSProp | ADAM |
|---|---|---|---|---|
| CIFAR10 | **35.639** | 66.074 | 49.029 | 45.108 |
| CIFAR100 | 41.010 | 66.669 | 12.079 | **11.956** |
| MNIST | **9.161** | 15.309 | 9.214 | 35.411 |
| FashionMNIST | **36.045** | 46.210 | 42.942 | 39.111 |

Table VII: High frequencies ratio of the first CNN layer

| Dataset | SGD | MSGD | RMSProp | ADAM |
|---|---|---|---|---|
| CIFAR10 | **47.097** | 63.258 | 75.339 | 65.985 |
| CIFAR100 | **56.435** | 72.653 | 65.361 | 59.662 |
| MNIST | 43.117 | 40.617 | **39.049** | 54.584 |
| FashionMNIST | 49.976 | **49.430** | 75.184 | 58.569 |

Table VIII: High frequencies ratio of the linear layer

### B. Figures

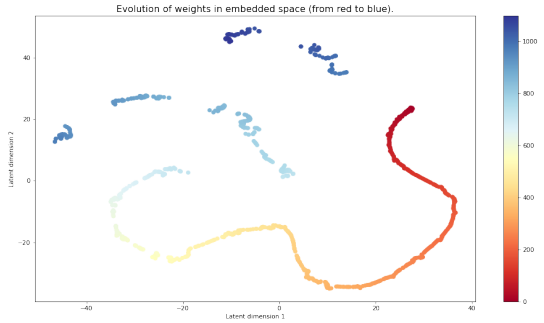


Figure 3: Weights trajectory in the first CNN layer on CIFAR10 with SGD

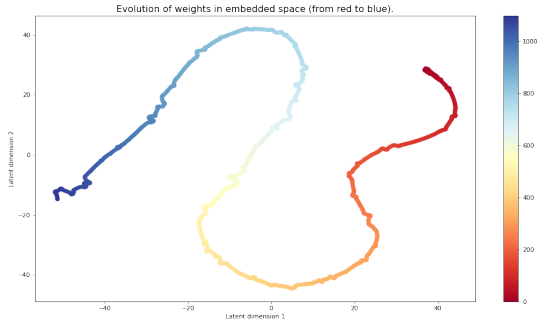


Figure 4: Weights trajectory in the first CNN layer on CIFAR10 with MSGD

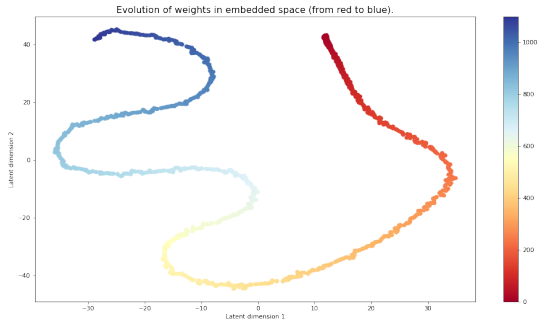


Figure 5: Weights trajectory in the first CNN layer on CIFAR10 with RMS

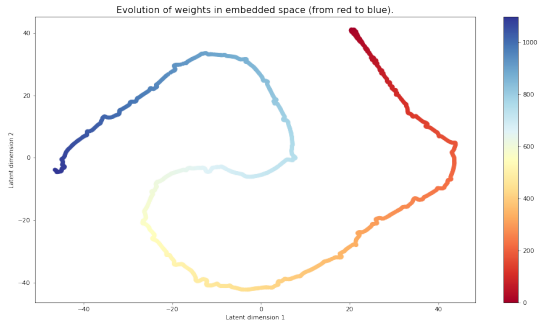


Figure 6: Weights trajectory in the first CNN layer on CIFAR10 with ADAM

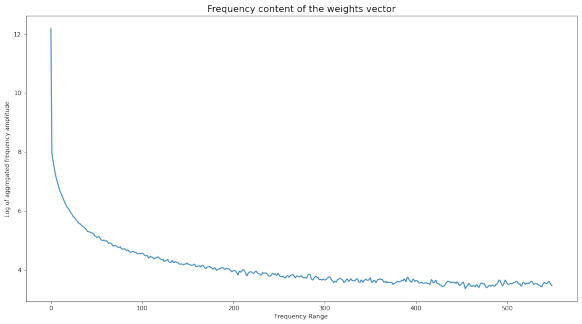


Figure 7: Frequency content in the first CNN layer on CIFAR10 with SGD

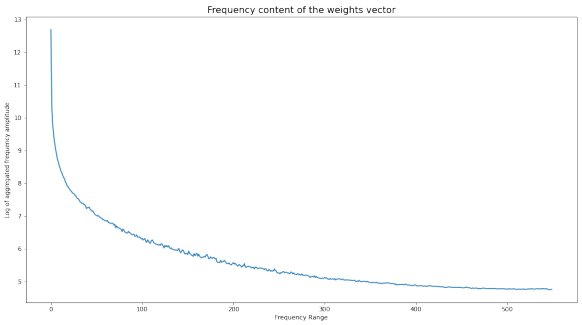


Figure 8: Frequency content in the first CNN layer on CIFAR10 with MSGD

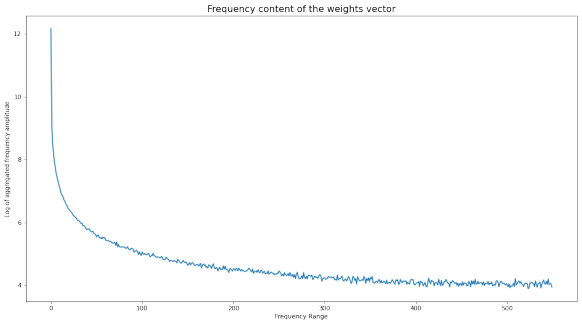


Figure 9: Frequency content in the first CNN layer on CIFAR10 with RMS

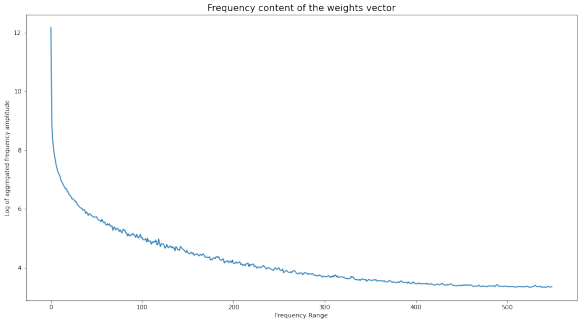


Figure 10: Frequency content in the first CNN layer on CIFAR10 with ADAM

## REFERENCES

[1] L. van der Maaten. [Online]. Available: https://lvdmaaten.github.io/tsne/