

Long-Tailed Recognition via Weight Balancing (Supplementary Material)

Shaden Alshammari* Yu-Xiong Wang[‡] Deva Ramanan^{‡,†} Shu Kong[‡]

*MIT [‡]UIUC [†]Argo AI [‡]CMU

shaden@mit.edu yxw@illinois.edu {deva, shuk}@andrew.cmu.edu

<https://github.com/ShadeAlsha/LTR-weight-balancing>

Outline

In this document, we first supplement the ablation study with more results to justify the use of regularizers for better learning for long-tailed recognition (LTR). We then present our open-source code in Jupyter Notebook as a self-explanatory tutorial. Lastly, we attach a video demo that shows how weights change during training with different regularizers.

1. Detailed Ablation Study

In Table 1, we list more results in addition to the ablation study presented in the main paper. Please refer to the caption for salient conclusions.

2. Open-Source Code

Description. We

- `demo1_first-stage-training.ipynb`

Running this file compares the first-stage training between a naive network (without weight decay) and a model with a tuned weight decay. It should achieve an overall accuracy $\sim 39\%$ and $\sim 46\%$ respectively on the CIFAR100-LT (imbalance factor 100).

- `demo2_second-stage-training.ipynb`

Running this file will compare various regularizers used in the second-stage training. It should achieve an overall accuracy $> 52\%$.

Why Jupyter Notebook? We prefer to release the code using Jupyter Notebook (<https://jupyter.org>) because it allows for interactive demonstration for education purposes. In case the reader would like to run python script, using the following command can convert a Jupyter Notebook file XXX.ipynb into a Python script:

```
jupyter nbconvert --to script XXX.ipynb
```

Table 1. **Ablation study** on CIFAR100-LT (IF=100) w.r.t top-1 accuracy (%). “CE”: cross-entropy loss; “CB”: class-balanced loss [1]; “WD”: weight decay; “Max”: MaxNorm constraint; “default-WD”: using the weight decay tuned for the first-stage training; “ τ -norm”: τ -normalization [2]; “+”: fine-tuning the last layer(s) as the second-stage training. Here are salient conclusions. (1) Learning with a properly tuned WD boosts performance from 38.38% to 46.08%, that is +8% increase. (2) Re-training the last layer with CB and WD gives another boost (+6%) to 52.42%. (3) Based on the above, applying additional MaxNorm yields a slight improvement +1% (53.35%); finetuning the last two layers achieves 53.55%. (4) Finetuning more layers performs worse, presumably because CB induces modified gradients that affect feature learning, and so hurt the final LTR performance.

model	many	median	few	avg
on the last layer (classifier)				
WD=0 (w/ CE)	64.05	35.80	11.43	38.38
+ τ -norm ($\tau=1.0$)	59.54	38.23	25.93	42.00
WD tuned (w/ CE)	76.94	44.28	12.17	46.08
+ τ -norm ($\tau=1.9$)	73.11	47.69	30.10	51.31
+ L2norm	76.09	47.74	20.87	49.60
+ CE & L2norm	76.37	48.11	21.00	49.87
+ CE & WD	76.97	45.94	14.00	47.22
+ CE & Max	76.80	47.26	15.10	47.95
+ CE & Max & default-WD	76.89	47.06	13.90	47.55
+ CE & Max & WD	76.80	47.51	14.40	47.83
+ CB	77.00	45.89	13.60	47.09
+ CB & L2norm	76.43	48.20	21.60	50.10
+ CB & WD	72.77	49.74	31.80	52.42
+ CB & Max	76.49	49.23	20.67	50.20
+ CB & Max & default-WD	76.20	48.91	21.50	50.24
+ CB & WD & Max	72.60	51.86	32.63	53.35
on the last two layers				
+ CE & WD & Max	76.34	48.46	21.17	50.03
+ CB & WD & Max	71.37	51.17	35.53	53.55
on the last five layers				
+ CE & WD & Max	76.03	48.14	20.87	49.72
+ CB & WD & Max	74.37	49.80	26.63	51.45

Requirement. Running our code requires some common packages. We installed Python and most packages through Anaconda. A few other packages might not be in-

stalled automatically, such as Pandas, torchvision, and PyTorch, which are required to run our code. Below are the versions of Python and PyTorch used in our work.

- Python version: 3.7.4 [GCC 7.3.0]
- PyTorch version: 1.7.1

We suggest assigning >1GB space to run all the files. The code will save checkpoints after every training epoch.

License. We release open-source code under the MIT License to foster future research in this field.

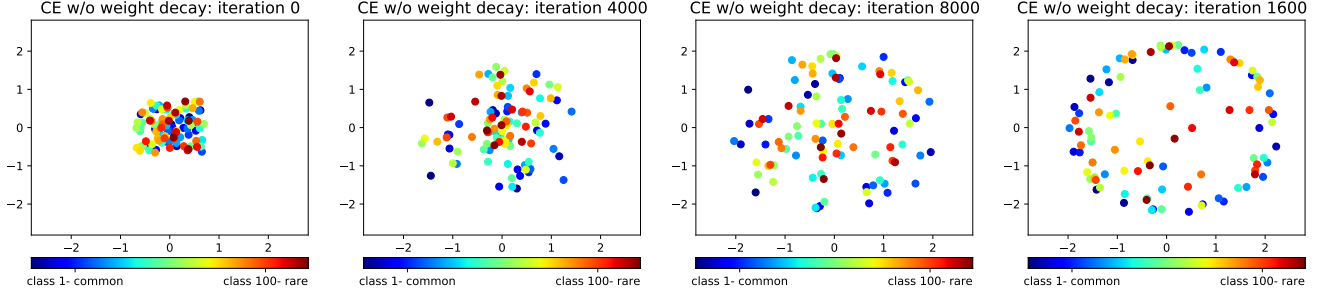
3. Video Demo

The goal of this section is to demonstrate how weights' norms evolve during training. For demonstration, we train models on the CIFAR100-LT dataset with an imbalance factor 100. To do so, we modify the ResNet34 network architecture by inserting an additional 2-dim pre-logit layer. This layer has weights $\mathbf{W} = [\mathbf{w}_{ij}] \in \mathbb{R}^{2 \times K}$ that project 2-dim pre-logit features to K -dim logits. At the logit layer, each filter weight \mathbf{w}_i (i.e., a row of \mathbf{W}) is class-specific. Therefore, we can plot the K 2-dim class-specific weights as K points on a 2D plane. The MaxNorm constraint upper bounds the norm of each class-specific weight, i.e., $\|\mathbf{w}_i\|_2 < \delta$. Fig. 1 plots per-class weights after three different training iterations. For better visualization, we suggest the reader to watch our video demo `demo2D_weight_evolution.mp4` in our github repository https://github.com/ShadeAlsha/LTR-weight-balancing/blob/master/demo2D_weight_evolution.mp4.

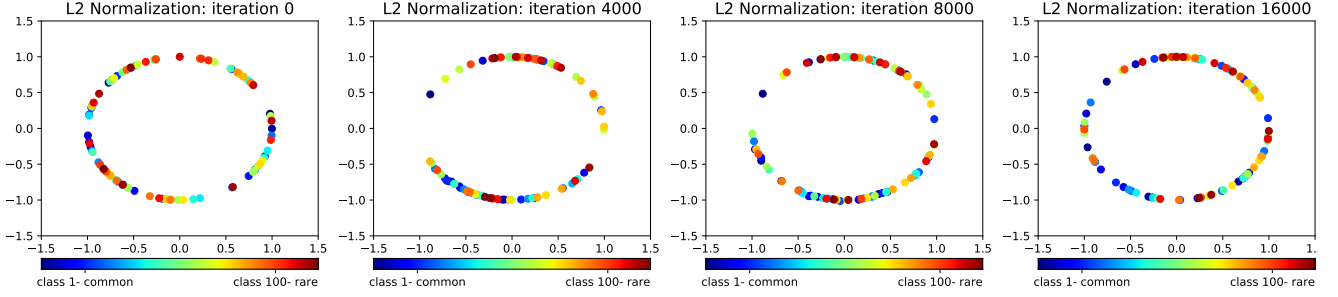
References

- [1] Yin Cui, Menglin Jia, Tsung-Yi Lin, Yang Song, and Serge Belongie. Class-balanced loss based on effective number of samples. In *CVPR*, 2019. 1
- [2] Bingyi Kang, Saining Xie, Marcus Rohrbach, Zhicheng Yan, Albert Gordo, Jiashi Feng, and Yannis Kalantidis. Decoupling representation and classifier for long-tailed recognition. In *ICLR*, 2020. 1

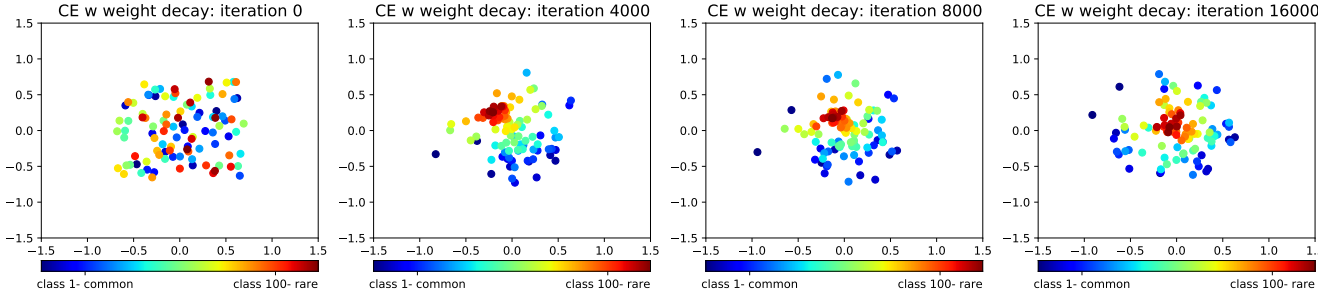
(a) Naively trained network without weight decay



(b) Network trained with L2-normalization



(c) Network trained with weight decay



(d) MaxNorm constrained network

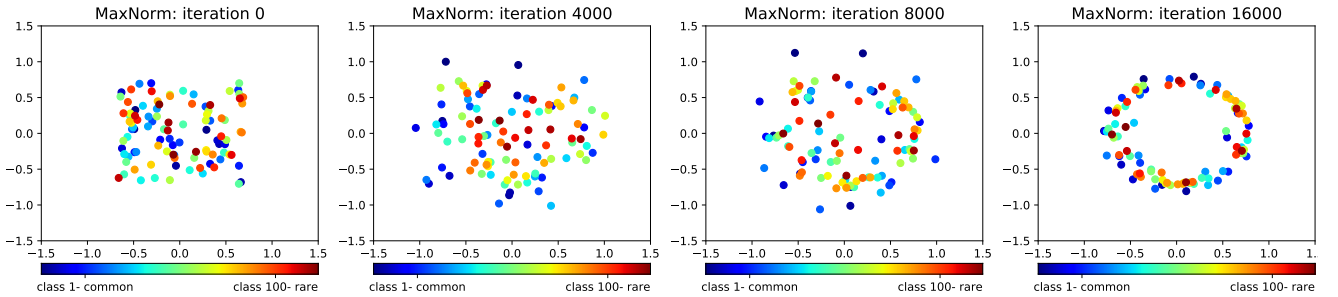


Figure 1. We plot per-class filter weights of the classifier as 2D points from **(a)** a naively trained network without weight decay, **(b)** a classifier with L2-normalization, **(c)** a classifier with weight decay, and **(d)** a classifier constrained by MaxNorm. All the networks are trained on the CIFAR100-LT dataset with imbalance factor as 100. The four columns denote training iterations: iteration-0 as random initialization, iteration-4000, iteration-8000, and iteration-16000. The naively trained network learns “imbalanced” weights, i.e., large weights for the common classes and small weights for the rare classes. The model trained with L2-normalization has constant weight norms. When trained with weight decay, the network has smaller yet more balanced weights for all the classes. The MaxNorm constrained network caps weight norms, encouraging small weights (from both common and rare classes) to grow, approaching to the surface of the norm ball. We refer the reader to the video demo `demo2D_weight_evolution.mp4` for better visualization.