Population Normalization for Federated Learning

Supplementary Material

In this appendix, we first present the details of how to train a neural network associated with our normalization methods using stochastic optimization algorithms. Then we provide the experimental configuration of this paper and the experimental results omitted in the main text due to the space limitation.

A. Details of The Training Process

A.1. Train Deep Neural Networks Associated with PN

Actually, the training problem of a DNN associated with PN, i.e., Equation 6 in the main paper, can be solved by various stochastic optimization methods. In the following, we take the stochastic Lagrangian multipliers as an example to show how to train such models. For simplicity, we give the steps of regular training, which can be adapted to FL training easily according to FedAVG.

By introducing the Lagrangian multipliers $(\lambda_k^{(\ell)}, \xi_k^{(\ell)})$ with $k \in [m^{(\ell)}]$ and $\ell \in \{0\} \cup [L]$, the constrained minimization problem (Equation 7 of the main paper) can then be transformed into the following unconstrained maximin problem:

$$\max_{\lambda,\xi} \min_{\theta,\mu,\gamma} J(\theta, \mu, \gamma, \lambda, \xi), \tag{7}$$

with

$$\begin{aligned} J(\theta, \mu, \gamma, \lambda, \xi) = & \frac{1}{|\mathcal{D}|} \sum_{(\boldsymbol{x},y)\in\mathcal{D}} \phi(f(\theta; \boldsymbol{x}), y) \\ & + \sum_{\ell=0}^{L} \sum_{k=1}^{m^{(\ell)}} \lambda_k^{(\ell)} \left(\frac{1}{|\mathcal{D}|} \sum_{(\boldsymbol{x},y)\in\mathcal{D}} \left(\gamma_k^{(\ell)} \left(f_k^{(\ell)}(\theta; \boldsymbol{x}) - \mu_k^{(\ell)} \right) \right)^2 - 1 \right) \\ & + \sum_{\ell=0}^{L} \sum_{k=1}^{m^{(\ell)}} \xi_k^{(\ell)} \left(\frac{1}{|\mathcal{D}|} \sum_{(\boldsymbol{x},y)\in\mathcal{D}} f_k^{(\ell)}(\theta; \boldsymbol{x}) - \mu_k^{(\ell)} \right) \end{aligned} \tag{8}$$

Here, we separate γ and μ from θ to highlight them. By changing the order of summation operators in the second and third items in $J(\theta, \mu, \gamma, \lambda, \xi)$, We can then rewrite it as follows:

$$\begin{aligned} J(\theta, \mu, \gamma, \lambda, \xi) = & \frac{1}{|\mathcal{D}|} \sum_{(\boldsymbol{x},y)\in\mathcal{D}} \Bigg[\phi(f(\theta; \boldsymbol{x}), y) \\ & + \sum_{\ell=0}^{L} \sum_{k=1}^{m^{(\ell)}} \lambda_k^{(\ell)} \left(\left(\gamma_k^{(\ell)} \left(f_k^{(\ell)}(\theta; \boldsymbol{x}) - \mu_k^{(\ell)} \right) \right)^2 - 1 \right) \\ & + \sum_{\ell=0}^{L} \sum_{k=1}^{m^{(\ell)}} \xi_k^{(\ell)} \left(f_k^{(\ell)}(\theta; \boldsymbol{x}) - \mu_k^{(\ell)} \right) \Bigg] \end{aligned} \tag{9}$$

Note that $J(\theta, \mu, \gamma, \lambda, \xi)$ has been written as the sum over the global training dataset $\mathcal{D} = \cup_{m=1}^{M} \mathcal{D}_m$, thus the problem (7) can be solved by alternatively applying stochastic gradient ascent on λ, ξ and stochastic gradient descent on θ with mini-batches as follows:

$$\begin{aligned} \theta_{t+1} &\leftarrow \theta_t - \eta_t \nabla_\theta \hat{J}(\theta_t, \mu_t, \gamma_t, \lambda_t, \xi_t), \\ \mu_{t+1} &\leftarrow \mu_t - \eta_t \nabla_\mu \hat{J}(\theta_t, \mu_t, \gamma_t, \lambda_t, \xi_t), \\ \gamma_{t+1} &\leftarrow \gamma_t - \eta_t \nabla_\gamma \hat{J}(\theta_t, \mu_t, \gamma_t, \lambda_t, \xi_t), \\ \lambda_{t+1} &\leftarrow \lambda_t + \eta_t \nabla_\lambda \hat{J}(\theta_t, \mu_t, \gamma_t, \lambda_t, \xi_t), \\ \xi_{t+1} &\leftarrow \xi_t + \eta_t \nabla_\xi \hat{J}(\theta_t, \mu_t, \gamma_t, \lambda_t, \xi_t), \end{aligned}$$

where η_t is the learning rate and $\nabla_\theta \hat{J}, \nabla_\mu \hat{J}, \nabla_\gamma \hat{J}, \nabla_\lambda \hat{J}$ and $\nabla_\xi \hat{J}$ are the stochastic gradients calculated on a random sampled mini-batch $\mathcal{B}$. It is obvious that we can solve Equation 6 in the main paper as above with arbitrary batch size by choosing a proper learning rate.

A.2. Train Deep Neural Networks Associated with NPN

The training process of DNN associated with NPN is similar to that associated with PN. The only difference is that in each forward propagation we first sample data independent noises n_μ and n_γ and then inject them into γ and μ, respectively, and we treat the noises as untrainable parameters in the back propagation.

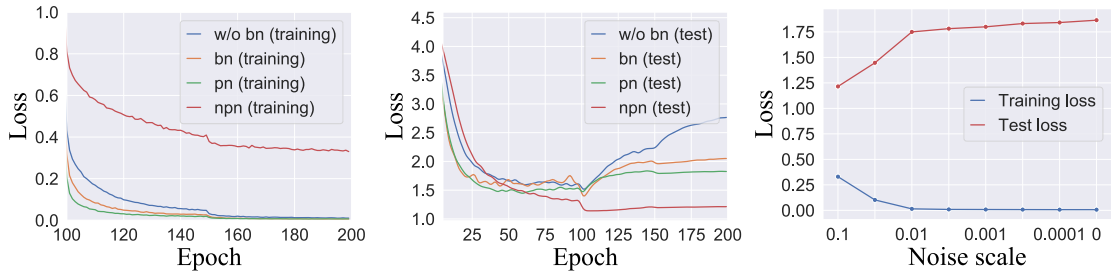
B. Experimental Configuration

In this section, we present the experimental settings and implementation details.

Firstly, our code is based on PyTorch framework and all the experiments on CIFAR-10 and CIFAR-100 are conducted on one RTX 2080-Ti with 11GB memory.

For federated learning with small batch sizes, we comply with the following setting: for local training, the networks are trained using SGD optimizer with momentum and the weight decay is set as 1e-4. The local learning rate for batch size 8 is set to the best among $\{0.01, 0.015, 0.02, 0.025\}$ for each normalization technique. We use step decay learning rate scheduler and reduce learning rate by $\frac{1}{10}$ at 100^{th} and 150^{th} global communication rounds. We apply random crop and random horizontal flip for data augmentation.

For experiments of training with large batch sizes, the local learning rate is set to 0.1, the local batch size is set to 2048, and the other local hyper-parameters are set to be the same as above. In each training step, the artificially injected



(a) (b)

Figure 3. (a) The training (left) and testing (right) loss of VGG-16 associated with BN, PN and NPN over the training process on CIFAR-100 dataset; (b) The sensitivity of NPN to noise level on CIFAR-100 dataset.

noises for each channel are sampled independently from the distributions of $\mathcal{N}(0,\ 0.1)$

For all experiments, the statistics μ and γ are initialized to satisfy the constraints of Equation 6 in the main paper and c and u are initialized with 0 and 1, respectively. The learning rates of the Lagrangian multipiers $(\lambda_k^{(l)},\ \xi_k^{(l)})$ are set to 0.01 with weight decay of 1 to smoothen the training phase.

C. Additional Experimental Results

In this section, we provide more experimental results, which are omitted in the main text due to the space limitation.

C.1. Results on CIFAR-100

Below, we give the detailed experimental results on CIFAR-100 to show the superiority of our proposed method over the baselines. Here, the batch size is set to 2048.

We visualize the curves of VGG-16 associated with different normalization methods in Figure 3(a). We can see that the testing loss of NPN is significantly smaller than all the baselines at the end of the training process and the gap becomes larger as the training goes on, while its training loss is much larger than the other baselines over the whole training process. This result demonstrates that our NPN can prevent the model from being over-fitting effectively.

To show the sensitivity of our NPN to the noise level, we train VGG-16 on CIFAR-100 with NPN under various noise levels, i.e., the variance of the noise varies in $(0, 0.1)$. The results are reported in Fig.3(b). We can observe that the training (resp. testing) loss increases (resp. decreases) slowly when the noise becomes larger. Therefore, in real applications, we can find a proper noise level without expending considerable efforts.

C.2. NPN with Different Kinds of Noise

Since in all the experiments above, we use Gaussian Noise in NPN. A natural question to our NPN is that can we inject other kinds of noise into PN to improve its generalization? To answer this question, we conduct a simple experiment on CIFAR-10 with VGG-16. To be precise, we compare the performances of NPN with two different kinds of noises, i.e., Gaussian noise $\mathcal{N}(0, 0.1)$ and Uniform noise $U(-\sqrt{3/10}, \sqrt{3/10})$. These two kinds of noises have the same variance 0.1.

Table 6. The performances of our NPN with different kinds of noise on VGG-16.

Noise	Loss	Top-1	Top-2	Top-3
Gaussian	0.28	93.66	98.05	99.12
Uniform	0.28	93.68	98.12	99.06

The results are given in Table 6. It shows that NPNs with Gaussian noise and Uniform noise can achieve comparable performances. Therefore, we can use different kinds of noise in NPN and the only thing we need to do is to choose a proper noise level. This result implies that our idea of injecting data independent noise into normalization methods is a very general technique.