A Extra Federated Learning Results

In this section, we present additional results under various system and training settings to further evaluate the robustness of our approach.

A.1 Different Local Training Epochs

Tuning the number of local epochs affects accuracy-communication trade-off for most federated learning algorithms. Prior studies [32,51,60] attempt to reduce the number of local training epochs to mitigate the disparity of local models. The default number of local training epochs is set as $E = 10$ in the main manuscript. We further test different numbers of local training epochs $E = \{1, 5\}$ in Tab. 5. When E is set as 1, we increase the total number of rounds to 500 to ensure convergence. As is seen from Tab. 5, our method consistently improves the base algorithms with various numbers of local training epochs.

Table 5. Accuracy (%) with different number of local training epochs.

Model	**Method**	$\boldsymbol{E = 1}$	$\boldsymbol{E = 5}$
MobileNetV2 ($\alpha = 0.5$)	FedAvg	70.51	68.40
	+ CCVR	71.33 ($\uparrow$0.82)	68.93 ($\uparrow$0.53)
	+ BABU	71.46 ($\uparrow$0.95)	69.10 ($\uparrow$0.70)
	+ Ours	**73.26 ($\uparrow$2.75)**	**72.02 ($\uparrow$3.62)**
VGG13 ($\alpha = 0.1$)	FedNova	60.53	57.46
	+ CCVR	61.31 ($\uparrow$0.78)	57.77 ($\uparrow$0.31)
	+ BABU	62.75 ($\uparrow$2.22)	58.50 ($\uparrow$1.04)
	+ Ours	**64.75 ($\uparrow$4.22)**	**62.06 ($\uparrow$4.60)**

A.2 Different Client Numbers

The default number of clients K is set as 10 following prior work [41,48]. We further test the performance of our methods when the system contains more clients. In Tab. 6, we partition CIFAR-100 training set to $K = 100$ clients according to a Dirichlet distribution with a concentration parameter $\alpha = 0.5$. During federated training, the central server randomly selects 10% clients to participate each round [32,43,54,60]. For each method, we set the number of rounds as 500. According to results in Tab. 6,

A.3 Different Learning Rate Scheduling Strategies

Besides adjusting the learning rate at each round according to a cosine annealing schedule[4], we further experiment another widely used learning rate scheduling

[4] See `torch.optim.lr_scheduler.CosineAnnealingLR`

Table 6. Accuracy (%) with $K = 100$ clients.

Model	Method	Accuracy	Model	Method	Accuracy
MobileNetV2	FedAvg	68.26	VGG13	FedNova	49.04
	+ CCVR	69.20 ($\uparrow$0.94)		+ CCVR	50.45 ($\uparrow$1.41)
	+ BABU	69.14 ($\uparrow$0.88)		+ BABU	51.27 ($\uparrow$2.23)
	+ Ours	**71.38 ($\uparrow$5.12)**		+ Ours	**55.23 ($\uparrow$6.19)**

strategy (*e.g.*, multi-step scheduling[5]) to verify the robustness of our approaches. Specifically, we decay the learning rate by 0.1 every 40 epochs. Empirical results in Tab. 7 indicate that our methods are able to bring state-of-the-art accuracy gain with different learning rate scheduling strategies.

Table 7. Accuracy (%) with different learning rate (LR) schedulings.

LR	method	Accuracy	LR	Method	Accuracy
Cosine	FedAvg	68.78	Multi-step	FedAvg	68.60
	+ CCVR	69.14 ($\uparrow$0.36)		+ CCVR	69.05 ($\uparrow$0.45)
	+ BABU	69.35 ($\uparrow$0.57)		+ BABU	69.43 ($\uparrow$0.83)
	+ Ours	**71.85 ($\uparrow$3.07)**		+ Ours	**71.06 ($\uparrow$2.46)**

B Personalization Performance Comparison

We investigate the performance gain brought by SphereFed for personalized federated learning. Following the setup in [29,54,77], we first combine the training and testing sets of CIFAR-100 to a single dataset which contains 60,000 samples in total. Then, the combined dataset is partitioned into 10 clients according to a Dirichlet distribution with a concentration parameter α. On each client, we use 15% its local data as local testing set and the other as local training set. Each personalized local model is evaluated on its corresponding local testing set. The overall performance of personalized federated learning is evaluated by calculating the mean and standard deviation of local testing accuracies across all clients [29,54,77].

We consider four recent personalized federated learning baselines for comparison in Tab. 8. For instance, LG-FedAvg [44] jointly updates feature extractors and classifiers during local training and only aggregates classifiers on the server in order to learn compact local representations. In FedRep [15], local feature extractors and classifiers are updated sequentially and the servers aggregates updated feature extractors at each round. The state-of-the-art pFL method, BABU [54], is also included in the comparison.

For our methods, we use SphereFed during federated training. Since it is not necessary to get the optimal global classifier for pFL, we skip the FFC for

[5] See `torch.optim.lr_scheduler.MultiStepLR`

the global classifier. Instead, we keep the learnt global feature extractor fixed and conduct personalization fine-tuning for the local classifier on each client. We consider two manners for the personalization fine-tunings. The first manner (duded 'Ours (SGD)' in Tab. 8) is to optimize the classifier on the local training set with SGD optimizer like prior arts [29,54,77]. The second manner (duded 'Ours (FFC)' in Tab. 8) is to adapt our FFC to compute the closed-form optimal classifier on the local training set (according to Eq. (7)).

Empirical results in Tab. 8 indicate that our proposed methods are able to improve pFL as well with hyperspherical features which are better aligned and less biased.

Table 8. Accuracy (%) comparison of different personalized federated learning (pFL) methods. The numbers 'A ± B' are the 'mean ± standard deviation' of personalized accuracies across clients.

Data	FedAvg	LG-FedAvg	FedRep	BABU	Ours (FFC)	Ours (SGD)
$\boldsymbol{\alpha = 0.5}$	69.95 ± 3.96	71.67 ± 4.33	62.39 ± 3.91	72.34 ± 3.84	75.63 ± 3.13	**75.71±3.32**
$\boldsymbol{\alpha = 0.1}$	80.16 ± 3.77	81.95 ± 3.69	73.63 ± 3.74	82.12 ± 3.63	84.06 ± 3.10	**84.12±3.29**

C Extra Calibration Results

C.1 Adapting An ℓ_2 Penalty in the Closed-Form Solution

In Sec. 4.3, we formulate the calibrating the classifier $\mathbf{W}$ as a least square problem in Eq. (6). Theoretically, an ℓ_2 penalty of $\mathbf{W}$ can be added and the objective of calibrating the classifier is,

$$\arg\min_{\mathbf{W}} \ \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}} \left[\mathcal{L}_{\text{MSE}}\left(\mathbf{W}\mathbf{z},\ y\right)\right] + \lambda\|\mathbf{W}\|_2^2 \quad , \tag{11}$$

where λ is a hyper-parameter used to control the penalty intensity. As a result, the closed-form weights optimum (*i.e.*, Eq. (10)) becomes,

$$\mathbf{W}^* = \left(\sum_{k=1}^{K} \mathbf{V}^k + \lambda\mathbf{I}\right)^{-1} \left(\sum_{k=1}^{K} \mathbf{U}^k\right) , \tag{12}$$

where $\mathbf{I} \in \mathbb{R}^{l\times l}$ is the identity matrix. In practise, we test different values for $\lambda \in \{0,\ 10^{-3},\ 10^{-2},\ 10^{-1},\ 10^{0},\ 10^{1}\}$ and find that the difference among resulted accuracies is less than 0.27% (10^{-1} results in the better accuracy 71.96%). Therefore, we keep $\lambda = 0$ in our main experiments.

C.2 Calibrate Weights One or Multiple Times?

Theoretically, we can conduct the Fast Federated Calibration (FFC) multiple times during the federated training. In practise, we attempt to calibrate the classifier every 10 rounds and get a final accuracy 71.88% which is quite close to the accuracy of one-time calibration (71.85%). This empirical result suggests that the pre-defined orthogonal hyperspherical $\mathbf{W}$ serves as a high-quality feature learning target (as discussed in Sec. 4.1) against a calibrated one. Moreover, conducting calibration multiple times will introduce extra communication and computation overheads. In this regard, we conduct FFC once after federated training in our main experiments.

C.3 Applying FFC with Features Trained with CE

The derivation of Fast Federated Calibration (FFC) relies on the mean square error (MSE) loss. However, in this section, we show that FFC can be used upon features trained by other losses (*e.g.*, cross entropy loss) because the training of the feature extractor and the calibration of the classifier are decoupled. In addition, both CE and MSE have similar optimization goal, *i.e.*, encouraging the feature extractor to make features of the i-th class close to $\mathbf{w}_i$. For instance, we train the feature extractor with cross entropy loss (CE) and then calibrate the classifier with our proposed FFC in Tab. 9. Experiential results verify that FFC is able to improve the classifier on features trained with CE.

Table 9. Applying FFC for the classifier on features trained with CE.

	FedAvg	**+ Fix (R)**	**+ Fix (R) + FFC**
CE	63.90	64.91 ($\uparrow$1.01)	65.36 ($\uparrow$1.46)

D Different Ways to Generate Orthogonal Classifier Initialization

We experiment two representative methods to generate the row-orthogonal weight matrix for classifier's weight initialization.

– QR-decomposition method. In linear algebra, a QR decomposition is to decompose a random matrix $\mathbf{A}$ into a product $\mathbf{A} = \mathbf{QR}$ of an orthogonal matrix $\mathbf{Q}$ and an upper triangular matrix $\mathbf{R}$ [1,58,73], in which $\mathbf{Q}$ is the matrix of our interest.
– Tammes method. To distribute C two-dimensional vectors on an unit circle as uniformly as possible, one can randomly place the first vector and then put the next vector by shifting the previous vector for an angle of $\frac{2\pi}{C}$. However, when the dimension is larger than two, no such optimal separation algorithm exists, which is known as the Tammes problem [71]. To maximize the separation for any vector dimension, Mettes *et al.* [52] optimize an objective which encourages large cosine similarity of any pair of vectors with

gradient decent. Following [52], we learn the vectors using SGD optimizer with 0.1 learning rate and 0.9 momentum for 10^4 steps.

Table 10. Accuracy (%) of different initialization methods for the classifier.

FedAvg+SphereFed	Init.	Rep. dim. (l)	#Classes (C)	Time	Accuracy
(MobileNetv2 on CIFAR-100)	QR	1280	100	0.02 s	71.85
	Tammes	1280	100	13.1 s	71.36

Tab. 10 shows the comparison of above two kinds of initialization methods for the orthogonal classifier weight matrix. QR-decomposition initialization achieves slightly better accuracy than the Tammes initialization. We also provide the wall time of the two methods which is measured on the machine with one NVIDIA GeForce GTX 1080 Ti GPU.

In all the other experiments of this work, QR method is used for SphereFed due to its efficiency and effectiveness.

E Details of Hardware Experiments

We evaluate the hardware performance of different on an embedded DNN training accelerator [91] based on the Xilinx VC707 FPGA evaluation board [78]. A 32 by 64 systolic array is used to perform the tensor operations during the forward and backpropagation. Each systolic cell consists of a Multiply-Accumulate (MAC) unit which can perform a floating-point multiplication and addition within a clock cycle, and a special unit is implemented to perform the operations for the rest layers (*e.g.*, group/batch normalization, ReLU). The hardware system runs at 100 MHz.

F Details of Implementation

F.1 Model Architectures

We provide the detailed information of ConvNet, MobileNetV2, ResNet18, VGG13 and SENet18 in Tabs. 11 to 15. For the 'NormLayer' after each convolution layer, two kinds of normalization layers are experimented. For experiments with MobileNetV2 and CIFAR-100, we instance the normalization layer as batch normalization. For other experiments, we use group normalization following prior arts [32,43,48,66,76,84,87].

F.2 Hyper-parameters

SphereFed and FFC do not introduce any extra hyper-parameter to base federated learning algorithms. Since we change the loss function from cross entropy to mean square error and these two loss functions have different magnitude, we tune the learning rate for both baselines and our methods using grid search from the limited candidate set $\{0.005, 0.01, 0.05, 0.1, 0.5, 0.8, 1.0\}$. Detailed default hyper-parameters are summarized in Tab. 16.

Table 11. Architecture of ConvNet

Block	**Layers**	**Repetition**
	Conv(3, 32, k=3, s=1), NormLayer(32), ReLU()	1
	Conv(32, 64, k=3, s=2), NormLayer(64), ReLU()	1
	Conv(64, 64, k=3, s=2), NormLayer(64), ReLU()	1
	Conv(64, 64, k=3, s=1), NormLayer(64), ReLU()	1
	Conv(64, 128, k=3, s=2), NormLayer(128), ReLU()	1
	Conv(128, 128, k=3, s=1), NormLayer(128), ReLU()	1
	Conv(128, 256, k=3, s=2), NormLayer(256), ReLU()	1
	Flatten()	1
	FeatureNorm() `if use` SphereFed	1
	FC(1024, 100, bias=False)	1

Table 12. Architecture of VGG13

Block	**Layers**	**Repetition**
	Conv(3, 64, k=3, s=1, p=1), NormLayer(64), ReLU()	1
	Conv(64, 64, k=3, s=1, p=1), NormLayer(64), ReLU()	1
	MaxPool2d(k=2, s=2)	1
	Conv(64, 128, k=3, s=1, p=1), NormLayer(128), ReLU()	1
	Conv(128, 128, k=3, s=1, p=1), NormLayer(128), ReLU()	1
	MaxPool2d(k=2, s=2)	1
	Conv(128, 256, k=3, s=1, p=1), NormLayer(256), ReLU()	1
	Conv(256, 256, k=3, s=1, p=1), NormLayer(256), ReLU()	1
	MaxPool2d(k=2, s=2)	1
	Conv(256, 512, k=3, s=1, p=1), NormLayer(256), ReLU()	1
	Conv(512, 512, k=3, s=1, p=1), NormLayer(256), ReLU()	1
	MaxPool2d(k=2, s=2)	1
	Conv(512, 512, k=3, s=1, p=1), NormLayer(256), ReLU()	2
	MaxPool2d(k=**TIN_S**, s=**TIN_S**)	1
	AvgPool2d(k=1, s=1)	
	Flatten()	1
	FeatureNorm() `if use` SphereFed	1
	FC(512, 100, bias=False)	1

*: **TIN_S**=1 if dataset is CIFAR-100 and **TIN_S**=2 if dataset is TinyImageNet.

Table 13. Architecture of SENet18

Block	Layers	Repetition
	Conv(3, 64, k=3, s=1, p=1), NormLayer(64), ReLU()	1

B1	Conv(64, 64, k=3, s=**TIN_S**, p=1), NormLayer(64), ReLU()
	Conv(64, 64, k=3, s=1, p=1), NormLayer(64), ReLU()
	SquzzeExcitationModule()
		1

	Conv(64, 64, k=3, s=1, p=1), NormLayer(64), ReLU()
	Conv(64, 64, k=3, s=1, p=1), NormLayer(64), ReLU()
	SquzzeExcitationModule()
		1

B2	Conv(64, 128, k=3, s=2, p=1), NormLayer(128), ReLU()
	Conv(128, 128, k=3, s=1, p=1), NormLayer(128), ReLU()
	SquzzeExcitationModule()
		1

	Conv(128, 128, k=3, s=1, p=1), NormLayer(128), ReLU()
	Conv(128, 128, k=3, s=1, p=1), NormLayer(128), ReLU()
	SquzzeExcitationModule()
		1

B3	Conv(128, 256, k=3, s=2, p=1), NormLayer(256), ReLU()
	Conv(256, 256, k=3, s=1, p=1), NormLayer(256), ReLU()
	SquzzeExcitationModule()
		1

	Conv(256, 256, k=3, s=1, p=1), NormLayer(256), ReLU()
	Conv(256, 256, k=3, s=1, p=1), NormLayer(256), ReLU()
	SquzzeExcitationModule()
		1

B4	Conv(256, 512, k=3, s=2, p=1), NormLayer(512), ReLU()
	Conv(512, 512, k=3, s=1, p=1), NormLayer(256), ReLU()
	SquzzeExcitationModule()
		1

	Conv(512, 512, k=3, s=1, p=1), NormLayer(512), ReLU()
	Conv(512, 512, k=3, s=1, p=1), NormLayer(512), ReLU()
	SquzzeExcitationModule()
		1

	AvgPool2d(k=4, s=4)
	Flatten()
	FeatureNorm() `if use` SphereFed
	FC(512, 100, bias=False)
		1
		1
		1
		1

*: **TIN_S**=1 if dataset is CIFAR-100 and **TIN_S**=2 if dataset is TinyImageNet.

Table 14. Architecture of MobileNetV2.

Block	Layers	Repetition
	Conv(3, 32, k=3, s=1), NormLayer(32), ReLU()	1

B1	Conv(32, 32, k=1, s=1), NormLayer(32), ReLU()
	Conv(32, 32, k=3, s=1, p=1, g=32), NormLayer(32), ReLU()
	Conv(32, 16, k=1, s=1), NormLayer(16), ReLU()
		1

B2	Conv(16, 96, k=1, s=1), NormLayer(96), ReLU()
	Conv(96, 96, k=3, s=**TIN_S**, p=1, g=96), NormLayer(96), ReLU()
	Conv(96, 24, k=1, s=1), NormLayer(24), ReLU()
		1

	Conv(24, 144, k=1, s=1), NormLayer(144), ReLU()
	Conv(144, 144, k=3, s=1, p=1, g=144), NormLayer(144), ReLU()
	Conv(144, 24, k=1, s=1), NormLayer(24), ReLU()
		1

B3	Conv(24, 144, k=1, s=1), NormLayer(144), ReLU()
	Conv(144, 144, k=3, s=2, p=1, g=144), NormLayer(144), ReLU()
	Conv(144, 32, k=1, s=1), NormLayer(32), ReLU()
		1

	Conv(32, 192, k=1, s=1), NormLayer(192), ReLU()
	Conv(192, 192, k=3, s=1, p=1, g=192), NormLayer(192), ReLU()
	Conv(192, 32, k=1, s=1), NormLayer(32), ReLU()
		2

B4	Conv(32, 192, k=1, s=1), NormLayer(192), ReLU()
	Conv(192, 192, k=3, s=2, p=1, g=192), NormLayer(192), ReLU()
	Conv(192, 64, k=1, s=1), NormLayer(64), ReLU()
		1

	Conv(64, 384, k=1, s=1), NormLayer(384), ReLU()
	Conv(384, 384, k=3, s=1, p=1, g=384), NormLayer(384), ReLU()
	Conv(384, 64, k=1, s=1), NormLayer(64), ReLU()
		3

B5	Conv(64, 384, k=1, s=1), NormLayer(384), ReLU()
	Conv(384, 384, k=3, s=1, p=1, g=384), NormLayer(384), ReLU()
	Conv(384, 96, k=1, s=1), NormLayer(96), ReLU()
		1

	Conv(96, 576, k=1, s=1), NormLayer(576), ReLU()
	Conv(576, 576, k=3, s=1, p=1, g=576), NormLayer(576), ReLU()
	Conv(576, 96, k=1, s=1), NormLayer(96), ReLU()
		2

B6	Conv(96, 576, k=1, s=1), NormLayer(576), ReLU()
	Conv(576, 576, k=3, s=2, p=1, g=576), NormLayer(576), ReLU()
	Conv(576, 160, k=1, s=1), NormLayer(160), ReLU()
		1

	Conv(160, 960, k=1, s=1), NormLayer(960), ReLU()
	Conv(960, 960, k=3, s=1, p=1, g=960), NormLayer(960), ReLU()
	Conv(960, 160, k=1, s=1), NormLayer(160), ReLU()
		2

B7	Conv(160, 960, k=1, s=1), NormLayer(960), ReLU()
	Conv(960, 960, k=3, s=1, p=1, g=960), NormLayer(960), ReLU()
	Conv(960, 320, k=1, s=1), NormLayer(320), ReLU()
		1

	Conv(320, 1280, k=1, s=1), NormLayer(1280), ReLU()	1
	AvgPool2d(k=4, s=4)	1
	Flatten()	1
	FeatureNorm() `if use` SphereFed	1
	FC(1280, 100, bias=False)	1

*: **TIN_S**=1 if dataset is CIFAR-100 and **TIN_S**=2 if dataset is TinyImageNet.

Table 15. Architecture of ResNet18

Block	**Layers**	**Repetition**
	Conv(3, 64, k=3, s=1, p=1), NormLayer(64), ReLU()	1

B1	Conv(64, 64, k=3, s=**TIN_S**, p=1), NormLayer(64), ReLU()
	Conv(64, 64, k=3, s=1, p=1), NormLayer(64), ReLU()
		1

	Conv(64, 64, k=3, s=1, p=1), NormLayer(64), ReLU() Conv(64, 64, k=3, s=1, p=1), NormLayer(64), ReLU()	1

B2	Conv(64, 128, k=3, s=2, p=1), NormLayer(128), ReLU()
	Conv(128, 128, k=3, s=1, p=1), NormLayer(128), ReLU()
		1

	Conv(128, 128, k=3, s=1, p=1), NormLayer(128), ReLU() Conv(128, 128, k=3, s=1, p=1), NormLayer(128), ReLU()	1

B3	Conv(128, 256, k=3, s=2, p=1), NormLayer(256), ReLU()
	Conv(256, 256, k=3, s=1, p=1), NormLayer(256), ReLU()
		1

	Conv(256, 256, k=3, s=1, p=1), NormLayer(256), ReLU() Conv(256, 256, k=3, s=1, p=1), NormLayer(256), ReLU()	1

B4	Conv(256, 512, k=3, s=2, p=1), NormLayer(512), ReLU()
	Conv(512, 512, k=3, s=1, p=1), NormLayer(256), ReLU()
		1

	Conv(512, 512, k=3, s=1, p=1), NormLayer(512), ReLU() Conv(512, 512, k=3, s=1, p=1), NormLayer(512), ReLU()	1
	AvgPool2d(k=4, s=4)	1
	Flatten()	1
	FeatureNorm() `if use` SphereFed	1
	FC(512, 100, bias=False)	1

*: **TIN_S**=1 if dataset is CIFAR-100 and **TIN_S**=2 if dataset is TinyImageNet.

Table 16. Summary of default hyper-parameters.

Method	Hyper-parameters	IID	$\alpha = 0.5$	$\alpha = 0.1$	TinyImageNet
FedAvg (MobileNetV2)	Rounds	100			
	Optimizer	SGD			
	Weights decay	0.00001			
	Momentum	0.9			
	Local epochs	10			
	Local batch size	64			
	Learning rate	0.1			
+ CCVR	# virtual features per class [48]	500	500	500	1000
	fine-tuning learning rate [48]	0.00001	0.00001	0.00001	0.00001
+ BABU	learning rate	0.1	0.1	0.1	0.01
+ Ours	learning rate	0.5	0.5	1.0	0.5
FedProx (ResNet18)	Rounds	100			
	Optimizer	SGD			
	Weights decay	0.00001			
	Momentum	0.9			
	Local epochs	10			
	Local batch size	64			
	Learning rate	0.1			
	μ	0.001			
+ CCVR	# virtual features per class [48]	500	500	500	1000
	fine-tuning learning rate [48]	0.00001	0.00001	0.00001	0.00001
+ BABU	learning rate	0.1	0.1	0.1	0.1
	μ	0.001	0.001	0.001	0.001
+ Ours	learning rate	0.5	0.5	0.5	0.5
	μ	0.0001	0.0001	0.0001	0.001
FedNova (VGG13)	Rounds	100			
	Optimizer	SGD			
	Weights decay	0.00001			
	Momentum	0.9			
	Local epochs	10			
	Local batch size	64			
	Learning rate	0.01			
+ CCVR	# virtual features per class [48]	500	500	500	1000
	fine-tuning learning rate [48]	0.00001	0.00001	0.00001	0.00001
+ BABU	learning rate	0.01	0.01	0.01	0.001
+ Ours	learning rate	0.1	0.1	0.1	0.1
FedOpt (SENet18)	Rounds	100			
	Optimizer	SGD			
	Weights decay	0.00001			
	Momentum	0.9			
	Local epochs	10			
	Local batch size	64			
	Local learning rate	0.01			
	On-server optimizer [60]	SGD			
	On-server learning rate [60]	1.0			
	On-server momentum [60]	0.3			
+ CCVR	# virtual features per class [48]	500	500	500	1000
	fine-tuning learning rate [48]	0.00001	0.00001	0.00001	0.00001
+ BABU	Local learning rate	0.01	0.01	0.01	0.01
+ Ours	Local learning rate	0.5	0.5	0.5	0.5