A Proofs

Proposition A.1. *The gradient of the forward KL divergence with respect to the coreset* $\mathbf{u}$ *is*

$$\nabla_{\mathbf{u}} D_{\mathrm{KL}}[\nu_{\mathbf{x}} \| \nu_{\mathbf{u}}] = -\nabla_{\mathbf{u}} \mathbb{E}_{[\nu_{\mathbf{x}}]_{\mathbf{u}}}[\log p(\tilde{\mathbf{y}} \mid \mathbf{f}_{\mathbf{u}})] + \mathbb{E}_{[\nu_{\mathbf{u}}]_{\mathbf{u}}}[\nabla_{\mathbf{u}} \log p(\tilde{\mathbf{y}} \mid \mathbf{f}_{\mathbf{u}})], \tag{10}$$

where $[\nu_{\mathbf{x}}]_{\mathbf{u}}$ *and* $[\nu_{\mathbf{u}}]_{\mathbf{u}}$ *are finite-dimensional distributions of the stochastic processes* $\nu_{\mathbf{x}}$ *and* $\nu_{\mathbf{u}}$*, respectively,* $\mathbf{f}_{\mathbf{u}} := (f(u_j))_{j=1}^m$*, and* $p(\tilde{\mathbf{y}} \mid \mathbf{f}_{\mathbf{u}}) = \prod_{j=1}^m p(\tilde{y}_j \mid f(u_j))$*.*

Proof. We follow the arguments in de G. Matthews et al. [10], Rudner et al. [29]. The forward KL divergence is defined as,

$$D_{\mathrm{KL}}[\nu_{\mathbf{x}} \| \nu_{\mathbf{u}}] = \int \log \frac{\mathrm{d}\nu_{\mathbf{x}}}{\mathrm{d}\nu_{\mathbf{u}}}(f) \mathrm{d}\nu_{\mathbf{x}}(f). \tag{18}$$

By the chain rule for the Radon-Nikodym derivative, we have

$$D_{\mathrm{KL}}[\nu_{\mathbf{x}} \| \nu_{\mathbf{u}}] = \int \log \frac{\mathrm{d}\nu_{\mathbf{x}}}{\mathrm{d}\nu_0}(f) \mathrm{d}\nu_{\mathbf{x}}(f) - \int \log \frac{\mathrm{d}\nu_{\mathbf{u}}}{\mathrm{d}\nu_0}(f) \mathrm{d}\nu_{\mathbf{x}}(f). \tag{19}$$

The first term does not depend on $\mathbf{u}$, so we investigate the second term. By the measure theoretic Bayes' rule,

$$\frac{\mathrm{d}\nu_{\mathbf{u}}}{\mathrm{d}\nu_0}(f) = \frac{p(\tilde{\mathbf{y}} \mid f, \mathbf{u})}{p(\tilde{\mathbf{y}} \mid \mathbf{u})}, \tag{20}$$

where $p(\tilde{\mathbf{y}} \mid f, \mathbf{u}) := \prod_{j=1}^m p(\tilde{y}_j \mid u_j, f)$ and,

$$p(\tilde{\mathbf{y}} | \mathbf{u}) = \int p(\tilde{\mathbf{y}} | f, \mathbf{u}) \mathrm{d}\nu_0(f). \tag{21}$$

Now let $\rho_A : (\mathcal{X} \to \mathbb{R}^d) \to (A \to \mathbb{R}^d)$ be a projection function that takes a function f and returns its restriction on a set $A \subseteq \mathcal{X}$. Assuming that the likelihood depends only on the finite index set $\mathbf{u}$, we can write

$$\frac{\mathrm{d}\nu_{\mathbf{u}}}{\mathrm{d}\nu_0}(f) = \frac{\mathrm{d}[\nu_{\mathbf{u}}]_{\mathbf{u}}}{\mathrm{d}[\nu_0]_{\mathbf{u}}}(\rho_{\mathbf{u}}(f)) = \frac{p(\tilde{\mathbf{y}} \mid \mathbf{f}_{\mathbf{u}})}{p(\tilde{\mathbf{y}} \mid \mathbf{u})}, \tag{22}$$

where $[\cdot]_{\mathbf{u}}$ denotes the finite-dimensional distribution of stochastic process evalauted at $\mathbf{u}$ and $\rho_{\mathbf{u}}(f) := \mathbf{f}_{\mathbf{u}} := (f(u_j))_{j=1}^m$ are corresponding function values at $\mathbf{u}$. Putting this back into the above equation,

$$\begin{aligned} \int \log \frac{\mathrm{d}\nu_{\mathbf{u}}}{\mathrm{d}\nu_0}(f) \mathrm{d}\nu_{\mathbf{x}}(f) &= \int \log \frac{\mathrm{d}[\nu_{\mathbf{u}}]_{\mathbf{u}}}{\mathrm{d}[\nu_0]_{\mathbf{u}}}(\mathbf{f}_{\mathbf{u}}) \mathrm{d}[\nu_{\mathbf{x}}]_{\mathbf{u}}(\mathbf{f}_{\mathbf{u}}) \\ &= \int \log \frac{p(\tilde{\mathbf{y}} \mid \mathbf{f}_{\mathbf{u}})}{p(\tilde{\mathbf{y}} \mid \mathbf{u})} \mathrm{d}[\nu_{\mathbf{x}}]_{\mathbf{u}}(\mathbf{f}_{\mathbf{u}}) \\ &= \mathbb{E}_{[\nu_{\mathbf{x}}]_{\mathbf{u}}}[\log p(\tilde{\mathbf{y}} \mid \mathbf{f}_{\mathbf{u}})] - \log p(\tilde{\mathbf{y}} \mid \mathbf{u}). \end{aligned} \tag{23}$$

Now taking the gradient w.r.t. $\mathbf{u}$, we get

$$\nabla_{\mathbf{u}} D_{\mathrm{KL}}[\nu_{\mathbf{x}} \| \nu_{\mathbf{u}}] = -\nabla_{\mathbf{u}} \mathbb{E}_{[\nu_{\mathbf{x}}]_{\mathbf{u}}}[\log p(\tilde{\mathbf{y}} \mid \mathbf{f}_{\mathbf{u}})] + \nabla_{\mathbf{u}} \log p(\tilde{\mathbf{y}} \mid \mathbf{u}). \tag{24}$$

Note also that

$$\begin{aligned} \nabla_{\mathbf{u}} \log p(\tilde{\mathbf{y}} \mid \mathbf{u}) &= \nabla_{\mathbf{u}} \log \int p(\tilde{\mathbf{y}} \mid f, \mathbf{u}) \mathrm{d}\nu_0(f) \\ &= \int \frac{\nabla_{\mathbf{u}} p(\tilde{\mathbf{y}} \mid f, \mathbf{u})}{p(\tilde{\mathbf{y}} \mid \mathbf{u})} \mathrm{d}\nu_0(f) \\ &= \int \nabla_{\mathbf{u}} \log p(\tilde{\mathbf{y}} \mid f, \mathbf{u}) \frac{p(\tilde{\mathbf{y}} \mid f, \mathbf{u})}{p(\tilde{\mathbf{y}} \mid \mathbf{u})} \mathrm{d}\nu_0(f) \\ &= \int \nabla_{\mathbf{u}} \log p(\tilde{\mathbf{y}} \mid f, \mathbf{u}) \frac{\mathrm{d}\nu_{\mathbf{u}}}{\mathrm{d}\nu_0}(f) \mathrm{d}\nu_0(f) \\ &= \int \nabla_{\mathbf{u}} \log p(\tilde{\mathbf{y}} \mid f, \mathbf{u}) \mathrm{d}\nu_{\mathbf{u}}(f) \\ &= \int \nabla_{\mathbf{u}} \log p(\tilde{\mathbf{y}} \mid \mathbf{f}_{\mathbf{u}}) \mathrm{d}[\nu_{\mathbf{u}}]_{\mathbf{u}}(f) = \mathbb{E}_{[\nu_{\mathbf{u}}]_{\mathbf{u}}}[\nabla_{\mathbf{u}} \log p(\tilde{\mathbf{y}} \mid \mathbf{f}_{\mathbf{u}})]. \end{aligned} \tag{25}$$

As a result, we conclude that

$$\nabla_{\mathbf{u}} D_{\mathrm{KL}}[\nu_{\mathbf{x}} \| \nu_{\mathbf{u}}] = -\nabla_{\mathbf{u}} \mathbb{E}_{[\nu_{\mathbf{x}}]_{\mathbf{u}}}[\log p(\tilde{\mathbf{y}} \mid \mathbf{f}_{\mathbf{u}})] + \mathbb{E}_{[\nu_{\mathbf{u}}]_{\mathbf{u}}}[\nabla_{\mathbf{u}} \log p(\tilde{\mathbf{y}} \mid \mathbf{f}_{\mathbf{u}})]. \tag{26}$$

□

B Inducing points in Stochastic Variational Gaussian Processes

Stochastic Variational Gaussian Processes [SVGP; 12, 13] were introduced as a solution to address the significant computational complexity, characterized by a cubic computational complexity of $O(n^3)$ and memory cost of $O(n^2)$, associated with performing inference using Gaussian Processes. In this context, n represents the total number of training data points. SVGP effectively leverages a concept called *inducing points*, which serves to reduce the computational complexity to $O(m^3)$ and memory requirements to $O(m^2)$ during inference, while still providing a reliable approximation of the posterior distribution of the entire training dataset. Notably, m denotes the number of inducing points, which is typically much smaller than the total number of training data points i.e. $m \ll n$. The above description clearly shows that the inducing points have a similar purpose to FBPC. However, there are some differences in their learning objectives. In the context of SVGP, the process of optimizing the inducing points denoted as $Z = \{z_1, \ldots, z_m\}$ involves maximizing the ELBO in order to make a variational Gaussian distribution $q(\mathbf{f}_{\text{tr}}, \mathbf{f}_z)$ well approximate the posterior distribution $p(\mathbf{f}_{\text{tr}}, \mathbf{f}_z | \mathbf{y}_{\text{tr}})$. This variational distribution is composed of two parts: 1) $p(\mathbf{f}_{\text{tr}} | \mathbf{f}_z)$, which represents the Gaussian posterior distribution, and 2) $q(\mathbf{f}_z)$, which is the Gaussian variational distribution. During this optimization, we focus on training the inducing points Z as well as the mean and variance of the variational distribution $q(\mathbf{f}_z)$. The goal of this optimization is to create a good approximation of the posterior distribution $p(y_{\text{te}} | x_{\text{te}}, D_{\text{tr}})$ during the inference process, all while keeping the computational cost low. On the other hand, as outlined in Section 3.2, the formulation of FBPC involves directly minimizing the divergence between function space posterior, specifically $D_{\text{KL}}[\nu_{\mathbf{x}} \| \nu_{\mathbf{u}}]$. To sum up, while they do share some similarities in that they introduce a set of learnable pseudo data points, they are fundamentally different in their learning objectives. SVGP is interested in approximating the full data posterior through the inducing points, while ours aims to make the pseudocoreset posterior as close as possible to the full data posterior.

C Experimetal Details

The code for our experiments will be available soon.

C.1 Expert trajectory

For expert trajectory, we trained the network with the entire dataset and saved their snapshot parameters at every epoch, following the setup described in [7]. For training, we used an SGD optimizer with a learning rate of 0.01. We saved 100 training trajectories, with each trajectory consisting of 50 epochs.

C.2 Hyperparmeter setting

Training In our training procedure, we have some hyperparameters. Firstly, we sampled the MAP solution of $\mathbf{x}$, denoted as $\theta_{\mathbf{x}}$, from the later part of the expert trajectories. The decision of how many samples from the later part to utilize was treated as a hyperparameter for each experimental setting. We chose to use samples from T epoch onwards as the MAP solution samples. When obtaining the MAP solution $\theta_{\mathbf{u}}$, there are also several hyperparameters involved, the optimizer and convergence criteria for training the MAP solution from random parameters. We used an Adam optimizer with a learning rate of 0.001 to train the model until the training loss reached a threshold of γ or below.

Next, to approximate our Gaussian variational function posterior, we employed the empirical covariance of the function samples. During the process of drawing function samples, we performed an additional training steps. This step involved specifying the optimizer and the number of steps used. We used an SGD optimizer with learning rates of $\lambda_{\mathbf{x}}$ and $\lambda_{\mathbf{u}}$ for a total of 30 steps during the additional training step for drawing function samples for $\mathbf{x}$ and $\mathbf{u}$, respectively.

Lastly, we used the training iteration N, a learning rate of α for pseudocoresets, and set the batch size for pseudocoresets to B. The hyperparameters used in our paper are summarized in Table 4.

Evaluation To implement the SGHMC algorithm, as discussed in [9] and following the recommendations of [9], we employed the SGD with momentum along with an auxiliary noise term.

$$\begin{cases} \Delta\theta = v \\ \Delta v = -\eta\nabla\tilde{U}(x) - \alpha v + \mathcal{N}(0, 2d). \end{cases} \tag{27}$$

we set $\eta = 0.03$, $\alpha = 0.1$, and $d = 0.01/m$, where m represents the coreset size. We perform 1000 epochs of SGHMC and collect samples every 100 epochs.

D Additional Experiments

D.1 Cross-architecture generalization

In order to assess the cross-architecture generalization performance of FBPC, we trained pseudo-corests of sizes {1, 10, 50} using the ConvNet architecture and tested them on various architectures trained from scratch. In addition to the ConvNet architecture used during training, we also evaluated the performance on sophisticated architectures such as ResNet18, ResNet34, VGG, and AlexNet. The results are presented in Table 5. As evident from Table 5, FBPC demonstrates considerable performance even on architectures different from those used during training, highlighting its strong cross-architecture generalization capabilities.

D.2 Training FBPC on larger neural networks

To evaluate the scalability of our method to large networks, we trained FBPC with the ResNet18 architecture. Training coreset with larger networks, such as ResNet18, has proven to be challenging and has been explored in only a few previous works [38]. This is primarily due to the lack of scalable training methods and the tendency for overfitting when training large networks. As a result, even when evaluating ResNet after training on a smaller network like ConvNet, the performance tends to suffer. Furthermore, it has been reported that coreset training directly on ResNet initially yields lower performance compared to training on ConvNet [40, 35].

Our experiments also revealed a similar trend in our findings as shown in the first column of Table 6. Although FBPC exhibits excellent scalability, making it easy to train on ResNet18 and larger networks, its performance was observed to be lower compared to ConvNet. On the other hand, the second column, ResNet18 + ConvNet, refers to training both ResNet18 and ConvNet simultaneously using the FBPC-multi training approach. In this case, surprisingly, the test performance of ResNet18 actually improved when trained in conjunction with ConvNet using the FBPC-multi training approach. In this case, the ConvNet accuracy was recorded at 60.03, which did not significantly compromise the ConvNet's performance while enhancing the performance of ResNet18. This suggests that training ConvNet acted as a regularizer, preventing overfitting in ResNet18 and enabling it to achieve better performance.

Table 4: Hyperparameter for each experiment setting.

	ipc	T	γ	$\lambda_{\mathbf{x}}$	$\lambda_{\mathbf{u}}$	N	α	B
CIFAR10	1	1	0.01	0.05	0.01	1000	100	10
	10	2	0.1	0.05	0.01	1000	1000	100
	50	10	0.2	0.05	0.01	1000	1000	500
CIFAR100	1	2	0.1	0.1	0.1	5000	1000	100
	10	40	0.1	0.1	0.1	5000	1000	1000
	50	20	0.2	0.01	0.01	300	1000	5000
Tiny-ImageNet	1	2	0.1	0.1	3	5000	1000	100
	10	40	1	0.1	3	500	1000	100
	50	40	1	0.1	3	500	1000	100

Table 5: Averaged test accuracy and negative log-likelihoods for each architecture of the pseudocoreset trained using the ConvNet architecture with CIFAR10 dataset.

		ConvNet	ResNet18	ResNet34	VGG	AlexNet
1	Acc (↑)	35.71±0.90	27.7±0.96	22.46±0.12	26.33±0.88	21.05±0.43
	NLL (↓)	3.44±0.07	2.87±0.13	3.06±0.02	5.38±0.74	3.13±0.60
10	Acc (↑)	62.53±0.34	47.51±1.73	35.48±1.22	47.87±1.18	32.27±0.78
	NLL (↓)	1.31±0.01	1.82±0.02	2.41±0.01	3.72±0.09	2.96±0.04
50	Acc (↑)	71.20±0.36	62.02±1.55	47.97±3.37	58.24±1.63	52.42±1.30
	NLL (↓)	1.03±0.00	1.41±0.05	2.10±0.10	3.03±0.26	2.08±0.07

Table 6: Test performance of FBPC (CIFAR10, ipc 10) on ResNet18. FBPC is trained with ResNet18 and ResNet18 + ConvNet (multiple architecture training).

	ResNet18	ResNet18 + ConvNet
Acc (↑)	50.00	54.90
NLL (↓)	1.75	1.56

D.3 Computational cost

To compare how scalable our approach is compared to posterior matching in the weight space, we measured GPU memory usage corresponding to the number of parameters. As shown in Table 7 and Table 8, the memory usage for weight space BPC significantly increases as the number of parameters grows, while FBPC operates very efficiently. Additionally, the coreset ipc increases memory usage proportionally to its size. In terms of memory considerations, FBPC excels. However, as shown in Table 9, in terms of time, our method requires slightly more time because more SGD steps are needed to acquire the empirical covariance. However, when using FBPC-isotropic, these steps can be reduced, trading off a slight decrease in performance for time savings.

D.4 Differentiable siamese augmentation

Table 10 shows the result for BPC-fKL and FBPC without using DSA [37] and without any augmentation during training. Interestingly, for the ipc 1 case in BPC-fKL, performance improved when DSA was not applied. However, in all other cases, it is evident that not using DSA leads to an average performance drop of approximately 4.7%. Moreover, even when training BPC without augmentation, we observe that function space BPC outperforms weight space BPC.

D.5 Visualization

In this section, we provide visualizations of the pseudocoreset examples for each dataset.

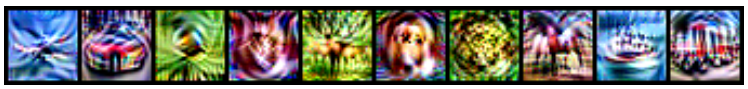

Figure 3: Example images of FBPC for CIFAR10 with ipc 1.

Table 7: GPU memory usage (GB) for training CIFAR10 FBPC with ipc 10.

	LeNet	ConvNet	ResNet18
# parameters	6.2×10^4	3.2×10^5	1.1×10^7
FBPC	0.02	0.32	2.56
BPC-fKL	0.11	3.17	12.18

Table 8: GPU memory usage (GB) for training CIFAR10 FBPC according to the ipc.

	1	10	50
FBPC	0.04	0.32	1.59
BPC-fKL	0.41	3.17	15.59

Table 9: Wall-clock time (sec) for 1 step update for training CIFAR10 pseudocoreset according to the ipc.

	1	10	50
BPC-fKL	$1.04_{\pm 0.10}$	$1.37_{\pm 0.13}$	$2.59_{\pm 0.86}$
FBPC	$1.5_{\pm 0.15}$	$3.29_{\pm 0.51}$	$8.38_{\pm 0.48}$

Table 10: BPC Performances with and without DSA.

	1	10	50
BPC-fKL (no DSA)	$37.26_{\pm 1.65}$	$50.48_{\pm 1.39}$	$60.75_{\pm 0.26}$
FBPC (no DSA)	$33.69_{\pm 2.73}$	$55.07_{\pm 1.30}$	$66.03_{\pm 0.21}$
FBPC (DSA)	$35.45_{\pm 0.31}$	$62.33_{\pm 0.34}$	$71.23_{\pm 0.17}$

(a) Examples images of FBPC for CIFAR10 with ipc 10. 1 image per class.

(b) Examples images of FBPC for CIFAR10 with ipc 50. 10 images per class.

Figure 4: Examples images of FBPCs for CIFAR10 dataset.

(a) Examples images of FBPC for CIFAR100 with ipc 1. 1 image per class.

(b) Examples images of FBPC for CIFAR100 with ipc 10. 1 image per class.

Figure 5: Examples images of FBPCs for CIFAR100 dataset.

Figure 6: Example images of FBPC for Tiny-ImageNet with ipc 1. 1 image per class.

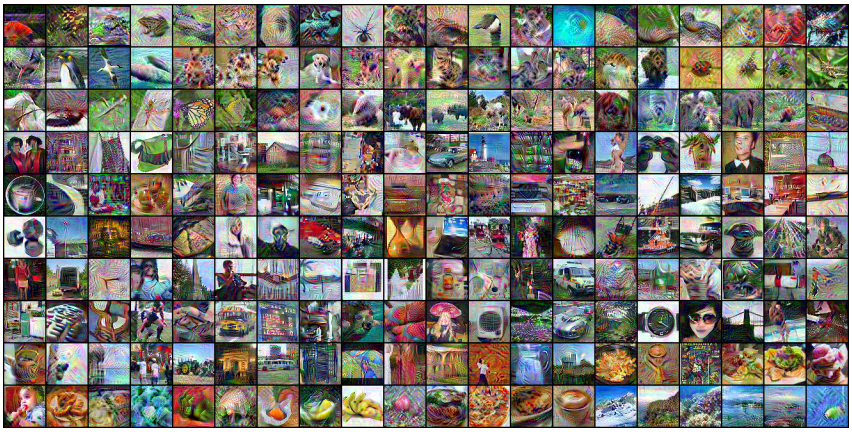

Figure 7: Example images of FBPC for Tiny-ImageNet with ipc 10. 1 image per class.