# Mixture of Experts: a Bayesian Architecture

**Daoyuan GUO**
The Hong Kong University of Science and Technology (Guangzhou)
dguo294@connect.hkust-gz.edu.cn

## Abstract

Sparse Mixture-of-Experts (MoE) architectures scale model capacity by routing each input through a small subset of experts, but their routers are typically deterministic and tuned with ad-hoc auxiliary losses, leaving open questions about calibration, data-awareness, and behavior under distribution shift. We revisit the router as a Bayesian component. Concretely, we place a factorized Gaussian posterior over the router's linear logits, derive closed-form per-input moments, and support two training modes: (i) an *Expected* mode that uses the logistic-normal approximation to obtain deterministic, low-variance gate probabilities, and (ii) an *MC* mode that optimizes a full ELBO with reparameterized samples and a control variate. To analyze routing, we introduce entropy-based metrics that capture alignment between expert usage and label structure: an Entropy Alignment Score (EAS) and a Data-Aware Routing Score (DARS) combining EAS with mutual information. On CIFAR-10/100 with SVHN as OOD, the Bayesian router matches or slightly outperforms strong deterministic MoE baselines in accuracy, while achieving higher DARS and more desirable ID/OOD confidence gaps, at essentially the same architecture and similar compute. Our results suggest that even lightweight Bayesian treatment of the router can make sparse MoEs more specialized, better calibrated, and more robust to distribution shift.

## 1 Introduction

Sparse Mixture-of-Experts (MoE) architectures have become a standard way to scale deep networks by routing each input through a small subset of experts rather than the full model [11, 2]. Recent systems work has demonstrated that such models can be trained at massive scale via sharding and efficient dispatch mechanisms [7], while algorithmic advances in routing and balanced assignment [8, 13] improve load balancing and utilization. However, most existing routers are deterministic linear maps trained with auxiliary balance losses and temperature tricks. This leaves several questions open: how well do router decisions reflect data structure, how calibrated are the resulting confidences, and how robust is routing under distribution shift?

In this work we study these questions through the lens of *Bayesian routing*. Rather than modifying the backbone or experts, we treat the router as the only stochastic component and place a factorized Gaussian posterior over its logits. Given input features, this posterior induces a Gaussian distribution over expert scores, for which we derive closed-form means and variances. We then instantiate two training modes: (i) an *Expected* mode that uses the standard logistic-normal approximation [3] to compute deterministic gate probabilities without sampling, and (ii) an *MC* mode that optimizes a variational ELBO with reparameterized samples and a control variate based on the Expected path. Both modes plug into a standard sparse MoE with top-$k$ routing and capacity constraints, and share all non-router hyperparameters.

To quantify how "data-aware" routing is, we further propose two entropy-based metrics. The Entropy Alignment Score (EAS) measures how closely the entropy of marginal expert usage tracks the entropy of the label distribution, and the Data-Aware Routing Score (DARS) combines EAS with the mutual information between expert index and label. High DARS corresponds, as we show empirically and in a simple analysis, to regimes where experts specialize on meaningful subsets of classes without collapsing to a few experts or spreading uniformly.

We evaluate the resulting Bayesian router on CIFAR-10 and CIFAR-100, using SVHN as an out-of-distribution (OOD) probe. Compared to strong deterministic MoE baselines—including Shazeer-style MoE with auxiliary losses, Expert Choice routing, auxiliary-loss-free balancing, and BASE-style balanced assignment—our Bayesian router: (i) matches or slightly improves test accuracy, (ii) achieves systematically higher DARS, and (iii) produces larger and more coherent gaps between ID and OOD confidence, while remaining efficient in Expected mode. A small ablation over KL weight and warm-up shows that a simple configuration (moderate KL, short warm-up, Expected routing) already gives a favorable accuracy–throughput trade-off; MC-ELBO training can be used as a slower, more Bayesian refinement that pushes DARS further on the harder CIFAR-100 task.

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

Overall, our contributions are three-fold: **(i)** We introduce a Bayesian router for sparse MoEs that models router uncertainty via a factorized Gaussian posterior over logits, with both Expected and MC-ELBO training modes, and no changes to backbone or experts; **(ii)** We propose EAS and DARS as compact, entropy-based metrics of data-aware routing, and relate high DARS to desirable specialization behavior under converged training; **(iii)** We provide an empirical study on CIFAR-10/100 with SVHN OOD showing that Bayesian routing improves routing quality and OOD confidence behavior at comparable accuracy and compute to strong deterministic MoE baselines.

## 2 Related Work

**Scaling and balancing sparse MoE.** Mixture-of-experts and hierarchical mixtures-of-experts [5, 6, 12, 9] combine local predictors via learned gating and form the classical basis for modern sparse MoE. Sparsely-gated MoE [11] introduced token-wise top-$k$ routing and auxiliary importance/load losses, enabling very large-capacity models at near-dense cost. Subsequent systems work scaled MoE via sharding and efficient dispatch/communication [7], while Switch Transformers [2] simplified routing to top-1, and V-MoE showed similar gains in vision [10]. Assignment-style formulations sharpen load balancing: BASE Layers [8] recast routing as a balanced assignment problem, and *Expert Choice* routing [13] inverts selection (experts pick tokens), guaranteeing hard balancing and improving convergence. Across these designs, the router is still a *deterministic* score function; balancing is enforced by auxiliary losses, capacity factors, or architectural constraints.

**Dispatch, capacity, and deterministic routing.** Most sparse MoE systems use a shared dispatch pattern: strict top-$k$ selection, renormalization within the selected set, partial expert execution, and per-expert capacity limits to prevent congestion. Auxiliary importance/load losses from early work [11, 7] are combined with temperature tuning and capacity factors to trade off utilization, overflow, and variance. These mechanisms control *how* gates are applied given scores, but leave the scores themselves as fixed logits, without an explicit notion of posterior uncertainty or calibration.

**Uncertainty, Bayesian MoE, and gating.** Early Bayesian treatments of mixtures-of-experts [4, 1] used fully probabilistic gating and variational inference in small-scale hierarchical MoE, but did not consider modern sparse top-$k$ routing, large expert counts, or OOD robustness. Classical variational analyses of logistic models [3] derive a quadratic bound and the well-known $\pi/8$ shrink for logistic-normal expectations, yielding a deterministic surrogate that reduces overconfidence and gradient variance. Related Bayesian ideas are widely used for classifiers and calibration, but are rarely integrated into sparse MoE routers, where logits are still treated as point estimates even when distribution shift is a concern. We bring this variational view directly to sparse MoE routing: the router computes per-input logit moments, uses the logistic-normal surrogate for low-variance expected routing, and admits an MC-ELBO training path when a more faithful Bayesian treatment is desired.

**Positioning.** Relative to deterministic routers with auxiliary balancing and system-level capacity control [11, 7], our approach *internalizes* uncertainty in the router without changing experts or backbones. Compared to assignment-based balancing [8, 13], we retain simple token-wise top-$k$ routing but gain calibrated probabilities, variance control, and a bridge between deterministic (Expected) and stochastic (MC-ELBO) modes. The result is a minimal, drop-in router that targets calibration, load controllability, and training variance via Bayesian modeling of the gating logits rather than via additional balancing losses alone.

## 3 Methodology

**Overview.** We use a sparse Mixture-of-Experts (MoE) with a shared backbone and experts, and replace the usual deterministic router with a Bayesian router that places a factorized Gaussian posterior over gating logits. The rest of the system (experts, backbone, dispatch) stays standard; all changes are localized in the router and its training objective. (Full definitions in App. A.)

**Architecture.** A deterministic backbone $f_\theta$ maps input $x$ to features $h \in \mathbb{R}^F$. We keep $E$ experts $\{f_e\}_{e=1}^E$, each outputting logits $f_e(h) \in \mathbb{R}^C$. A router produces expert scores, selects a small set of top-$k$ experts per token, and aggregates their outputs into the final prediction (App. A.5).

**Bayesian router.** For each expert $e$, the router logit is linear in features, $m_e = w_e^\top h + b_e$, with a factorized Gaussian posterior over $(w_e, b_e)$. Given $h$, this induces a Gaussian distribution over logits with closed-form mean and diagonal variance (App. A, Eqs. (4)–(5)). The prior is an isotropic Gaussian over router parameters.

**Training modes.** We support two ways to use this posterior: (i) *Expected* mode, which uses the standard $\pi/8$ logistic-normal approximation to compute deterministic gate probabilities without sampling (App. A.3, Eq. (6)); and (ii) *MC* mode, which samples logits via reparameterization and optimizes a full ELBO with a KL term and an unbiased control variate based on the Expected path (App. A.4). Both modes share the same architecture and differ only in how the router is trained.

**Sparse routing and capacity.** In both modes, routing follows the usual sparse MoE recipe: we compute softmax probabilities over experts (with temperature $T$), take `top-`$k$ scores, renormalize within the selected set, and execute only those experts (App. A.5, Eq. (12)). A capacity factor $\phi$ sets a per-expert token cap $C = \lfloor \phi N/E \rfloor$; overflowed tokens are dropped and counted as overflow (App. A.6, Eq. (13)).

| Dataset | Model | Mode | Acc | ECE | DARS | ID $\mu_{\max}$ | OOD $\mu_{\max}$ | $\Delta\mu$ |
|---|---|---|---|---|---|---|---|---|
| CIFAR-10 | Bayes | expected | 0.849 | 0.079 | 0.742 | 0.926 | 0.573 | 0.353 |
| | Bayes | mc | 0.853 | 0.073 | 0.727 | 0.924 | 0.438 | 0.486 |
| | Simple | – | 0.809 | 0.072 | 0.691 | 0.877 | 0.566 | 0.311 |
| | EC | – | 0.425 | 0.032 | 0.500 | 0.456 | 0.330 | 0.125 |
| | AuxFree | – | 0.841 | 0.073 | 0.712 | 0.914 | 0.558 | 0.356 |
| | BASE | – | 0.854 | 0.064 | 0.702 | 0.918 | 0.739 | 0.179 |
| CIFAR-100 | Bayes | expected | 0.528 | 0.249 | 0.612 | 0.776 | 0.355 | 0.421 |
| | Bayes | mc | 0.522 | 0.236 | 0.623 | 0.773 | 0.362 | 0.411 |
| | Simple | – | 0.489 | 0.272 | 0.529 | 0.724 | 0.381 | 0.343 |
| | EC | – | 0.460 | 0.056 | 0.501 | 0.714 | 0.571 | 0.143 |
| | AuxFree | – | 0.516 | 0.247 | 0.577 | 0.754 | 0.364 | 0.390 |
| | BASE | – | 0.523 | 0.249 | 0.576 | 0.665 | 0.432 | 0.232 |

Table 1: CIFAR-10/100 test performance and confidence statistics with SVHN as OOD. Acc: test accuracy; ECE: ID calibration; DARS: Data-Aware Routing Score; $\mu_{\max}$: mean max-softmax confidence; $\Delta\mu = \mu_{\max}^{\mathrm{ID}} - \mu_{\max}^{\mathrm{OOD}}$.

**Loss and inference.** The training loss combines: (i) cross-entropy on the MoE outputs (under Expected or MC routing), (ii) a KL term between router posterior and prior with a linear warm-up schedule, and (iii) an optional light $\mathrm{CV}^2$ penalty on expert loads for compatibility with classic MoE balancing (App. A.7, Eq. (14)). At test time we default to Expected routing, which is deterministic, fast, and better calibrated.

**Routing-quality metrics.** To summarize routing behavior, we measure: normalized entropies of expert usage and labels ($H_E^{\mathrm{norm}}$, $H_Y^{\mathrm{norm}}$), their entropy alignment $\mathrm{EAS} = 1 - |H_E^{\mathrm{norm}} - H_Y^{\mathrm{norm}}|$, and the Data-Aware Routing Score $\mathrm{DARS} = \frac{1}{2}(\mathrm{EAS} + \mathrm{NMI}(\mathrm{expert}; \mathrm{label}))$. Formal definitions and basic lemmas are given in App. B.

**Implementation details.** All variants share the same backbone, experts, optimizer, and data pipeline. We use standard CIFAR-style augmentation (random crop/flip, per-channel normalization), and tune only router-specific hyperparameters such as $k$, temperature $T$, capacity factor $\phi$, KL weight, and (for MC) the number of samples. Exact values and search ranges are summarized in the experimental appendix (App. A.10, C).

# 4 Experiments

We empirically study two questions: (i) how KL weight, KL warm-up and router mode (Expected vs. MC) affect accuracy, calibration, routing quality and efficiency (RQ1); (ii) with a tuned Bayesian router, how our MoE compares to standard non-Bayesian MoEs in terms of ID performance and OOD confidence (RQ2).

## 4.1 Setup

All models share a ResNet-18 backbone and a sparse MoE head with $E = 16$ experts, top-$k = 2$ routing and a capacity factor around 1.0. We train on CIFAR-10 or CIFAR-100 as in-distribution (ID) data and use SVHN as out-of-distribution (OOD). Optimization, data augmentation, router hyperparameter ranges and metric definitions (ECE, DARS, confidence statistics) follow Sec. 3; full details, ablation grids and plots are deferred to App. A.10 and App. C. App. C.4 further extends the main runs to 200 epochs and shows that 100 epochs are sufficient for convergence: all models change by at most a few tenths of a percentage point in test accuracy and sometimes slightly overfit.

## 4.2 RQ1: Bayesian Router Ablation

We first ablate the Bayesian router on CIFAR-100 with SVHN OOD, varying the KL weight $\beta_{\mathrm{KL}} \in \{10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}\}$, whether KL warm-up is used, and the router training mode (Expected vs. MC); App. C.1 reports full curves and summary tables. For the Expected router, increasing $\beta_{\mathrm{KL}}$ without warm-up slightly improves calibration but hurts accuracy when the KL becomes too strong; adding a short warm-up flattens the dependence on $\beta_{\mathrm{KL}}$, giving an accuracy band around $0.51$–$0.52$ with moderate ECE. For the MC router, no-warm-up runs are very well calibrated but clearly underfit and are slow, while warm-up raises accuracy into the $0.50$–$0.51$ range, with moderate ECE and roughly doubled throughput. Across all runs, Expected mode dominates the accuracy–throughput Pareto front, reaching $\approx 0.51$–$0.52$ accuracy at $\approx 2500$–$2750$ img/s, whereas MC mode trades speed for a more faithful Bayesian treatment and slightly higher DARS on harder settings. These trends motivate the recipe used in RQ2: a moderate KL ($\beta_{\mathrm{KL}} \in [10^{-6}, 10^{-5}]$) with a short warm-up, Expected routing as the default, and MC routing as an optional refinement.

## 4.3 RQ2: Comparison with Non-Bayesian MoE Baselines

We now fix tuned Bayesian routers (Expected and MC) and compare them against four sparse MoE baselines: Simple_Moe (Shazeer-style auxiliary balance losses), Expert Choice (EC), Aux_Free_Moe (aux-free balancing), and BASE_Moe (balanced assignment). We report test accuracy, ID calibration (ECE), routing quality (DARS), and ID/OOD mean max-softmax confidence, denoted by $\mu_{\max}^{\mathrm{ID}}$, $\mu_{\max}^{\mathrm{OOD}}$ and their gap $\Delta\mu = \mu_{\max}^{\mathrm{ID}} - \mu_{\max}^{\mathrm{OOD}}$. All results are taken from the final training epoch; additional visualizations are provided in App. C.2–C.3.

On CIFAR-10 (Tab. 1, top block), the Bayesian routers reach the top-accuracy cluster together with BASE ($\approx 0.85$), while achieving higher DARS than all non-Bayesian baselines. EC attains very low ECE but suffers from low accuracy and weak data-aware routing (DARS $\approx 0.50$). Expected-Bayes and MC-Bayes both produce substantially lower OOD mean confidence than BASE; MC-Bayes yields the largest ID–OOD confidence gap ($\Delta\mu \approx 0.49$), indicating cautious OOD predictions without sacrificing ID performance.

CIFAR-100 is considerably harder, but the pattern is similar (Tab. 1, bottom block). Expected-Bayes attains the highest test accuracy ($0.528$), slightly above BASE and Aux_Free_Moe, and both Bayesian variants achieve the highest DARS, indicating the strongest alignment between expert usage and label distribution. They also maintain clear ID–OOD confidence gaps ($\Delta\mu \approx 0.41$–$0.42$), whereas BASE remains more confident on OOD samples and EC again trades accuracy and DARS for low ECE. Overall, tuned Bayesian routers deliver MoE models that are competitive in ID accuracy yet more data-aware (high DARS) and more robust in OOD confidence than standard deterministic MoEs, while requiring only a local change to the router.

# 5 Discussion

Our results show that explicitly modeling router uncertainty is a practical way to improve sparse MoE behavior without changing the backbone or experts. Compared with deterministic routers trained only with auxiliary balance losses, the Bayesian router exposes a more structured trade-off between fit, calibration, and efficiency, and produces expert assignments that align more closely with the label structure.

On both CIFAR-10 and CIFAR-100, the Bayesian router matches or slightly outperforms strong non-Bayesian baselines such as BASE in test accuracy (Tab. 1), while consistently achieving higher DARS and more favorable ID/OOD confidence behavior. In the accuracy–DARS plots (App. C.2–C.3), Expected and MC variants occupy the upper-right region, indicating that entropy-aligned, class-aware routing is not only interpretable but predictively useful. The larger ID–OOD confidence gaps of the Bayesian models further show that router uncertainty can make the overall system more cautious under distribution shift, even though the experts remain deterministic.

Our entropy-based metrics give a compact, data-aware view of routing. EAS measures how closely expert-usage entropy tracks label entropy, and DARS combines EAS with $\mathrm{NMI}(\text{expert}; \text{label})$ into a single score. Empirically, high-DARS regimes correspond to models where experts specialize on meaningful subsets of classes while avoiding both collapse (few experts used) and overspread (near-uniform usage that ignores labels), matching the patterns observed in per-expert confusion matrices and routing-load histograms (App. C.2–C.3). Our analysis suggests that, under converged training and finite capacity, high EAS/DARS correspond to a data-aware fixed point where marginal expert usage approximates the label distribution and further specialization yields diminishing returns on the supervised loss.

The ablations in Sec. 4.2 also yield simple design guidelines. A moderate KL weight ($\beta_{\mathrm{KL}} \in [10^{-6}, 10^{-5}]$) with a short warm-up is sufficient to stabilize training and is robust across datasets. Expected-mode training provides a strong accuracy–throughput Pareto frontier by using logistic-normal expectations to obtain calibrated logits and high DARS at a compute cost comparable to a deterministic router. MC-ELBO training is slower and more sensitive to warm-up and sampling, but can be used as an optional refinement when one desires a more faithful Bayesian treatment or slightly higher DARS on harder tasks.

Several limitations remain. Our experiments are restricted to moderate-scale vision benchmarks (CIFAR-10/100) and a single OOD dataset (SVHN); it is unclear how the same trade-offs behave in very large language or multimodal MoEs with hundreds of experts and stricter latency constraints. We use a factorized Gaussian posterior; richer posteriors (e.g. structured covariances, sparsity-inducing priors) and more flexible routing schemes (e.g. dynamic-$k$, expert pruning, learned capacity factors) are not explored. Finally, our theoretical analysis focuses on EAS/DARS and expected routing under converged training and does not yet provide PAC-style guarantees for expert specialization or OOD detection, leaving a gap between empirical behavior and formal guarantees.

# 6 Conclusion

We introduced a Bayesian router for sparse Mixture-of-Experts that replaces ad-hoc regularization with explicit posterior modeling over router logits, and proposed entropy-based metrics (EAS, DARS) to quantify how data-aware a routing policy is. On CIFAR-10/100 with SVHN OOD, the Bayesian router delivers competitive or better accuracy than strong deterministic baselines, while achieving higher DARS and more desirable ID/OOD confidence patterns. Our ablations show that a simple configuration—factorized Gaussian posterior, medium KL with short warm-up, and Expected-mode training—already offers a favorable accuracy–throughput trade-off, with MC-ELBO updates as an optional, more Bayesian refinement.

These findings suggest that even lightweight Bayesian modeling of the router, without modifying experts or backbones, can make sparse MoEs more specialized, better calibrated, and more robust to distribution shift, and that EAS/DARS provide a useful lens for comparing routing strategies across architectures. Future work includes scaling Bayesian routing to larger MoE systems, exploring richer router posteriors and adaptive capacities, and tightening the connection between high DARS, Bayesian uncertainty, and downstream risk bounds in tasks such as selective prediction and OOD-aware decision-making.

## References


[1] C. M. Bishop and M. Svensén. Bayesian hierarchical mixtures of experts. In *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence (UAI)*, 2003.

[2] W. Fedus, B. Zoph, and N. Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022. URL `https://jmlr.org/papers/v23/21-0998.html`.

[3] T. S. Jaakkola and M. I. Jordan. A variational approach to Bayesian logistic regression models and their extensions. In D. Madigan and P. Smyth, editors, *Proceedings of the Sixth International Workshop on Artificial Intelligence and Statistics (AISTATS)*, volume R1 of *Proceedings of Machine Learning Research*, pages 283–294. PMLR, Jan 1997. URL `https://proceedings.mlr.press/r1/jaakkola97a.html`.

[4] R. A. Jacobs. Bayesian inference for hierarchical mixtures-of-experts with hidden markov model extensions. In *Proceedings of the IEEE*, 1996.

[5] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87, 1991. doi: 10.1162/neco.1991.3.1.79.

[6] M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural Computation*, 6(2):181–214, 1994. doi: 10.1162/neco.1994.6.2.181.

[7] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. In *9th International Conference on Learning Representations (ICLR)*, Virtual Event, 2021. OpenReview.net. URL `https://openreview.net/forum?id=qrwe7XHTmYb`.

[8] M. Lewis, S. Narang, S. Dharamshi, P. Shyam, A. Gulati, A. C. Li, R. Anil, J. Du, E. W. Yang, R. Liu, N. Goyal, D. Valter, V. Sharma, M. Shoeybi, J. Uszkoreit, T. Cohen, W. Xia, A. Vaswani, L. Zettlemoyer, B. Catanzaro, and A. Aghajanyan. Base layers: Simplifying training of large, sparse models. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, volume 139 of *Proceedings of Machine Learning Research*, pages 6265–6274. PMLR, 2021. URL `https://proceedings.mlr.press/v139/lewis21a.html`.

[9] S. Masoudnia and R. Ebrahimpour. Mixture of experts: a literature survey. *Artificial Intelligence Review*, 42(2):275–293, 2012. doi: 10.1007/s10462-012-9338-y.

[10] C. Riquelme, J. Puigcerver, B. Mustafa, M. Neumann, A. S. Pinto, D. Keysers, and N. Houlsby. Scaling vision with sparse mixture of experts. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[11] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. V. Le, G. E. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *5th International Conference on Learning Representations (ICLR)*, Toulon, France, 2017. OpenReview.net. URL `https://openreview.net/forum?id=B1ckMDqlg`.

[12] S. E. Yüksel, J. N. Wilson, and P. D. Gader. Twenty years of mixture of experts. *IEEE Transactions on Neural Networks and Learning Systems*, 23(8):1177–1193, 2012. doi: 10.1109/TNNLS.2012.2200299.

[13] Y. Zhou, T. Lei, H. Liu, N. Du, Y. Huang, V. Y. Zhao, A. M. Dai, Z. Chen, Q. V. Le, and J. Laudon. Mixture-of-experts with expert choice routing. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, 2022.

# Contents

# A Model and Training Details

## A.1 Notation and Architecture

Each training example $(x, y)$ consists of an input $x$ and a class label $y \in \{1, \dots, C\}$; we denote the corresponding one-hot vector by $\mathbf{e}_y \in \mathbb{R}^C$. A deterministic backbone network $f_\theta$ maps the input into a $F$-dimensional feature vector

$$h = f_\theta(x) \in \mathbb{R}^F,$$

which serves as the common representation fed to all experts and to the router.

We maintain $E$ experts $\{f_e\}_{e=1}^E$, each expert being a small classifier that maps the shared feature $h$ to class logits in $\mathbb{R}^C$:

$$f_e : \mathbb{R}^F \to \mathbb{R}^C, \qquad z_e = f_e(h).$$

All experts see the same feature dimension $F$, but can differ in their internal parameters. The MoE head then forms a sparse mixture of expert predictions: a router produces an expert-score vector $m = (m_1, \dots, m_E) \in \mathbb{R}^E$ and, after dispatch and capacity handling (App. A.5–A.6), yields a sparse mixture weight vector $\alpha = (\alpha_1, \dots, \alpha_E)$ with $\alpha_e \geq 0$ and $\sum_e \alpha_e = 1$. The final prediction is a weighted average of expert logits

$$\hat{y} = \sum_{e=1}^{E} \alpha_e f_e(h) \in \mathbb{R}^C.$$

The backbone and experts are shared across all routing variants; only the router is made Bayesian in our approach.

## A.2 Router Posterior and Logit Moments

For each expert $e$, the router logit is a linear function of the features:

$$m_e = w_e^\top h + b_e, \tag{1}$$

with weight vector $w_e \in \mathbb{R}^F$ and bias $b_e \in \mathbb{R}$. Collecting parameters across experts gives a weight matrix $W \in \mathbb{R}^{E \times F}$ (whose $e$-th row is $w_e^\top$) and a bias vector $b \in \mathbb{R}^E$. In a standard deterministic MoE, $(W, b)$ are trained by gradient descent with no notion of uncertainty. In contrast, we treat these parameters as latent random variables and place a factorized Gaussian posterior over $(w_e, b_e)$:

$$q(w_e) = \mathcal{N}\big(\mu_{w_e}, \mathrm{diag}(\sigma^2_{w_e})\big), \tag{2}$$

$$q(b_e) = \mathcal{N}(\mu_{b_e}, \sigma^2_{b_e}), \tag{3}$$

where $\mu_{w_e} \in \mathbb{R}^F$, $\sigma^2_{w_e} \in \mathbb{R}^F$ and $\mu_{b_e}, \sigma^2_{b_e} \in \mathbb{R}$. In implementation we parameterize the log-variances $\log \sigma^2_{w_e}, \log \sigma^2_{b_e}$ for numerical stability.

The prior is an isotropic Gaussian

$$p(w_e, b_e) = \mathcal{N}(0, \sigma_p^2 I),$$

with fixed variance $\sigma_p^2$ shared across all router parameters. This choice is deliberately simple: it encourages weights to remain close to zero unless strongly supported by data, and leads to a closed-form KL term.

Stacking posterior parameters across experts, we obtain $W_\mu \in \mathbb{R}^{E \times F}$ and $W_{\sigma^2} \in \mathbb{R}^{E \times F}$, whose $e$-th rows contain $\mu_{w_e}$ and $\sigma^2_{w_e}$, respectively, and similarly $b_\mu, b_{\sigma^2} \in \mathbb{R}^E$. Given a feature vector $h$, the induced mean and variance of the logit vector $m$ under $q$ have closed forms:

$$\mu_m(h) = W_\mu h + b_\mu, \tag{4}$$

$$\sigma^2_m(h) = (W_{\sigma^2} \odot W_{\sigma^2}) h^{\odot 2} + b^{\odot 2}_{\sigma^2}, \tag{5}$$

where $\odot$ denotes elementwise multiplication and $h^{\odot 2}$ is the elementwise square of $h$. Intuitively, the router variance is larger when a feature dimension has both high posterior variance and large magnitude; conversely, features with small or well-determined weights contribute little to uncertainty.

Thus, for each token, the router induces a diagonal Gaussian logit distribution

$$q(m \mid h) = \mathcal{N}\big(\mu_m(h), \mathrm{diag}(\sigma^2_m(h))\big),$$

which we exploit in both Expected and MC training modes.

**KL divergence.** Because both prior and posterior are Gaussian and factorized, the KL term decomposes over parameters and can be computed in closed form. For a single scalar parameter with posterior $q(\theta) = \mathcal{N}(\mu, \sigma^2)$ and prior $p(\theta) = \mathcal{N}(0, \sigma_p^2)$, the KL is

$$\mathrm{KL}(q \| p) = \frac{1}{2}\left(\frac{\sigma^2}{\sigma_p^2} + \frac{\mu^2}{\sigma_p^2} - 1 - \log \frac{\sigma^2}{\sigma_p^2}\right).$$

Summing over all dimensions of $w_e$ and $b_e$ and across experts yields $\mathrm{KL}\big(q(\Theta) \| p(\Theta)\big)$ in Eq. (14). This makes optimization straightforward and allows us to control the strength of Bayesian regularization through the scalar weight $\beta_t$ in Eq. (15).

### A.3 Expected-Mode Routing

Let $\sigma(z) = 1/(1+\exp(-z))$ denote the logistic function. For a scalar Gaussian random variable $z \sim \mathcal{N}(\mu, \sigma^2)$, a classical variational bound [3] implies the approximation

$$\mathbb{E}[\sigma(z)] \approx \sigma\left(\frac{\mu}{\sqrt{1+\frac{\pi}{8}\sigma^2}}\right). \tag{6}$$

This formula can be interpreted as a *variance-dependent shrinkage* of the mean: when $\sigma^2$ is small, the denominator is close to 1 and the expression reduces to $\sigma(\mu)$; when $\sigma^2$ is large, the effective logit magnitude is reduced, pushing the probability closer to $1/2$ and encouraging caution.

In our router, each component $m_e$ of the logit vector $m$ is Gaussian under $q(m \mid h)$. We apply Eq. (6) elementwise to obtain

$$\mathbb{E}_q[\sigma(m_e)] \approx \sigma\left(\frac{\mu_{m,e}(h)}{\sqrt{1+\frac{\pi}{8}\sigma^2_{m,e}(h)}}\right),$$

and then use these "shrunken" logits as inputs to a softmax. Formally, we define

$$\tilde{m}_e(h) = \frac{\mu_{m,e}(h)}{\sqrt{1+\frac{\pi}{8}\sigma^2_{m,e}(h)}}, \quad p_e(h) = \frac{\exp(\tilde{m}_e(h)/T)}{\sum_{j=1}^{E}\exp(\tilde{m}_j(h)/T)}, \tag{7}$$

where $T > 0$ is a temperature hyperparameter. Lower $T$ sharpens the distribution and encourages more decisive routing, while higher $T$ makes the router more uniform.

Sparse top-$k$ routing is then applied to $p(h)$ as in App. A.5. In Expected mode, both training and inference use these deterministic probabilities. This yields a router that is *aware* of its own uncertainty through the shrinkage in $\tilde{m}_e(h)$, but incurs no Monte Carlo noise and is computationally comparable to a deterministic linear gate.

### A.4 MC-ELBO Routing and Control Variate

Expected-mode routing already captures much of the effect of logit uncertainty, but it remains an approximation to the true logistic-normal expectation. To obtain a more faithful Bayesian treatment, we also consider a Monte Carlo ("MC") mode that optimizes a variational ELBO using reparameterized samples from $q(m \mid h)$.

In MC mode, we draw logit samples from $q(m \mid h)$ via the reparameterization trick:

$$m(h, \epsilon) = \mu_m(h) + \sigma_m(h) \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I). \tag{8}$$

For each sample, we apply softmax and sparse top-$k$ routing to obtain stochastic gates $p(h, \epsilon)$, compute the expert mixture, and obtain a sampled prediction $\hat{y}(h, \epsilon)$. Given a dataset $\mathcal{D} = \{(x_i, y_i)\}$, the ELBO objective is

$$\mathcal{L}_{\text{ELBO}} = \mathbb{E}_{q(\Theta)}\Big[-\sum_{(x,y)\in\mathcal{D}} \log p(y \mid x, \Theta)\Big] + \beta_{\text{KL}}\,\mathrm{KL}\big(q(\Theta)\,\|\,p(\Theta)\big), \tag{9}$$

where $\Theta$ collects all router parameters and $\beta_{\text{KL}}$ is a tunable KL weight. Gradients of the expectation are estimated by sampling $\epsilon$ and backpropagating through $m(h, \epsilon)$.

Pure MC estimates, however, can suffer from high variance. To alleviate this, we use the Expected-mode loss as a control variate. Concretely, let $\ell_{\text{MC}}(\epsilon)$ denote the per-batch loss under sampled routing (for a given $\epsilon$) and $\ell_{\text{Expected}}$ the loss under Expected routing. We write the MC objective as

$$\mathcal{L}_{\text{MC}} = \mathcal{L}_{\text{Expected}} + \mathbb{E}_\epsilon\big[\ell_{\text{MC}}(\epsilon) - \ell_{\text{Expected}}\big] + \beta_{\text{KL}}\,\mathrm{KL}\big(q(\Theta)\,\|\,p(\Theta)\big), \tag{10}$$

where $\mathcal{L}_{\text{Expected}}$ denotes the Expected-mode loss used in Sec. A.3. Under mild regularity conditions, we have $\mathbb{E}_\epsilon[\ell_{\text{MC}}(\epsilon)] = \mathbb{E}_\epsilon[\ell_{\text{Expected}}]$, so the difference term has zero mean and acts as an unbiased control variate: it preserves the expected gradient but reduces sample-to-sample variance. In practice, we approximate the expectation with a small number of MC samples per batch (often one or two), which we found sufficient to stabilize training.

### A.5 Dispatch and Mixing

Given a score vector $r \in \mathbb{R}^E$ (either the Expected logits $\tilde{m}(h)$ or a sampled logits vector $m(h, \epsilon)$), we first compute a temperature-scaled softmax:

$$p_e = \frac{\exp(r_e/T)}{\sum_{j=1}^{E}\exp(r_j/T)}. \tag{11}$$

We then select the `top-`$k$ experts $\mathcal{S}(h) = \mathrm{TopK}(r, k)$ by value of $r_e$. This selection is based on raw scores rather than probabilities so that ties and relative ordering are determined consistently across temperatures.

To obtain mixture weights, we renormalize within the selected set:

$$\alpha_e = \begin{cases} \dfrac{p_e}{\sum_{j\in\mathcal{S}(h)} p_j}, & e \in \mathcal{S}(h), \\ 0, & \text{otherwise}, \end{cases} \qquad \hat{y} = \sum_{e\in\mathcal{S}(h)} \alpha_e f_e(h). \tag{12}$$

Only experts in $\mathcal{S}(h)$ are executed, which allows the model to scale to large numbers of experts without linearly increasing compute per token. The overall complexity per token scales as $O(EF)$ for router computation, $O(E \log k)$ to identify the top-$k$ scores (or $O(Ek)$ with a partial selection algorithm), and $O(kC)$ for expert computation.

### A.6 Capacity Factor and Overflow

Without capacity constraints, popular experts can become congested while others are under-utilized, harming both efficiency and specialization. We therefore impose a per-expert capacity controlled by a *capacity factor* $\phi > 0$. Let $N$ be the number of tokens in a batch. The capacity $C$ is defined as

$$C = \left\lfloor \phi \frac{N}{E} \right\rfloor . \tag{13}$$

This corresponds to allowing each expert to handle at most a factor-$\phi$ multiple of its fair share of tokens.

During dispatch, for each expert $e$ we keep at most $C$ tokens with the highest routing weights $\alpha_e$ and drop any additional tokens routed to $e$, counting them as overflow. The stable matching between tokens and capacities is implemented via batched indexed scatter operations. We track the overflow rate (fraction of tokens dropped) as a diagnostic metric in experiments: very low overflow can indicate under-utilization of capacity, while very high overflow means that a few experts are repeatedly over-subscribed, which may hurt performance.

### A.7 Loss and KL Warm-up

Let $\ell_{\mathrm{CE}}$ denote the cross-entropy loss between the MoE output and the true label, computed under either Expected or MC routing. The overall loss for a batch is

$$\mathcal{L} = \ell_{\mathrm{CE}} + \beta_t \, \mathrm{KL}\big(q(\Theta) \,\|\, p(\Theta)\big) + \lambda_{\mathrm{CV}} \, \mathrm{CV}^2(\text{importance/load}), \tag{14}$$

where $\beta_t$ is a time-varying KL weight, $\lambda_{\mathrm{CV}}$ is a small coefficient (often zero in our main runs), and $\mathrm{CV}^2$ is the squared coefficient of variation over per-expert token counts (either based on gating "importance" or realized load). The optional $\mathrm{CV}^2$ term is included only to match classic MoE balancing schemes; our Bayesian router does not rely on it.

We use a simple linear warm-up schedule for the KL weight:

$$\beta_t = \beta_{\max} \cdot \min\left( \frac{t}{T_{\text{warm}}}, 1 \right), \tag{15}$$

where $t$ is the current epoch, $T_{\text{warm}}$ is the warm-up length, and $\beta_{\max}$ is the target KL weight. Intuitively, the model is allowed to fit the data with relatively weak Bayesian regularization during the early epochs; as training proceeds, $\beta_t$ increases and encourages the posterior to stay closer to the prior, preventing over-confident router weights and improving calibration. Our ablations indicate that a short warm-up (e.g., 10–20 epochs) with a moderate $\beta_{\max}$ is sufficient to stabilize both Expected and MC training.

### A.8 Data Preprocessing

For CIFAR-10/100, we apply standard data augmentation: random horizontal flips and random crops with padding around the original $32 \times 32$ images, followed by per-channel mean–std normalization. Augmentation is applied only to the training set; validation and test images are center-cropped (trivial for $32 \times 32$) and normalized in the same way. We do not use more aggressive augmentations (e.g., CutMix, Mixup) in order to keep the focus on routing rather than data augmentation.

SVHN is used only as an OOD dataset and is normalized with the same CIFAR statistics for fair comparison of softmax outputs. This ensures that differences in ID/OOD confidence reflect the behavior of the MoE and router, rather than trivial differences in input scaling.

### A.9 Computational Complexity

The router computation per token is dominated by two matrix–vector multiplications: one to compute the mean logits $W_\mu h$ and one to compute the variance term in Eq. (5). Both are $O(EF)$ operations. For the scale of our experiments ($E = 16$ experts and moderate feature dimension $F$), this cost is small relative to a single forward pass through the ResNet backbone.

Sparse dispatch executes only $k$ experts per token, yielding an overall expert cost of $O(kC)$ per token, where $C$ is the per-expert capacity. In Expected mode, routing is deterministic and requires a single forward pass through the router per batch. In MC mode, each additional sample incurs an extra router forward pass and a sparse mixture computation, roughly multiplying the router overhead by the number of samples. In practice, we use at most two MC samples per batch, so the total overhead remains modest. Our throughput measurements in App. C.1 confirm that Expected mode yields the best accuracy–speed Pareto frontier, while MC mode is slower but more Bayesian.

### A.10 Hyperparameters

Key hyperparameters for the Bayesian router include: the number of experts $E$ and active experts $k$, the temperature $T$, capacity factor $\phi$, the prior variance $\sigma_p^2$, the maximum KL weight $\beta_{\max}$, the warm-up length

$T_{\text{warm}}$, and (for MC mode) the number of ELBO samples per batch. We also share optimization hyperparameters (learning rate, weight decay, batch size, training epochs) across all MoE variants to ensure a fair comparison.

In our ablations, we explore $\beta_{\max} \in \{10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}\}$, with and without KL warm-up, in both Expected and MC modes, as well as a small grid over temperatures and capacity factors. The main text reports only the final selected configurations for each dataset and router mode; full grids and numerical results are provided in the experimental appendix (App. C–C.1). This separation keeps the main exposition focused while allowing the interested reader to inspect the detailed behavior of the Bayesian router under different settings.

# B Entropy-Based Routing Metrics

To quantify how data-aware the routing policy is, we define two entropy-based metrics: Entropy Alignment Score (EAS) and the Data-Aware Routing Score (DARS).

## B.1 Definitions

Consider a trained MoE model and a held-out dataset with label random variable $Y$ and expert-index random variable $E$ (top-1 routed expert). Let $H(\cdot)$ denote Shannon entropy in nats and $H(\cdot \mid \cdot)$ the conditional entropy. We define the normalized entropies

$$H_Y^{\mathrm{norm}} = \frac{H(Y)}{\log C}, \qquad H_E^{\mathrm{norm}} = \frac{H(E)}{\log E}, \tag{16}$$

which both lie in $[0, 1]$. The Entropy Alignment Score is

$$\mathrm{EAS} = 1 - \left| H_E^{\mathrm{norm}} - H_Y^{\mathrm{norm}} \right|. \tag{17}$$

EAS is maximal when the router uses experts with an entropy level comparable to the label entropy and decreases as the two diverge.

We also consider the mutual information between experts and labels,

$$\mathrm{MI}(E; Y) = H(E) - H(E \mid Y), \tag{18}$$

and its normalized version

$$\mathrm{NMI}(E; Y) = \frac{\mathrm{MI}(E; Y)}{\sqrt{H(E)\, H(Y)}}, \tag{19}$$

which lies in $[0, 1]$. The Data-Aware Routing Score is then defined as

$$\mathrm{DARS} = \frac{1}{2}\Big(\mathrm{EAS} + \mathrm{NMI}(E; Y)\Big). \tag{20}$$

## B.2 Basic Properties

We collect a few simple properties that justify EAS and DARS as routing metrics.

**Lemma 1.** *For any routing policy and label distribution, EAS satisfies* $0 \le \mathrm{EAS} \le 1$. *Moreover,* $\mathrm{EAS} = 1$ *if and only if* $H_E^{\mathrm{norm}} = H_Y^{\mathrm{norm}}$.

*Proof.* By definition $H_E^{\mathrm{norm}}, H_Y^{\mathrm{norm}} \in [0, 1]$, so $\left| H_E^{\mathrm{norm}} - H_Y^{\mathrm{norm}} \right| \in [0, 1]$ and hence $\mathrm{EAS} = 1 - |\cdot| \in [0, 1]$. The equality $\mathrm{EAS} = 1$ holds iff the absolute difference is zero, i.e., $H_E^{\mathrm{norm}} = H_Y^{\mathrm{norm}}$. □

**Lemma 2.** *For any routing policy and label distribution, DARS satisfies* $0 \le \mathrm{DARS} \le 1$. *If* $\mathrm{NMI}(E; Y) = 0$ *and* $H_E^{\mathrm{norm}}$ *is far from* $H_Y^{\mathrm{norm}}$*, then DARS is small. If* $\mathrm{NMI}(E; Y)$ *is large and* $H_E^{\mathrm{norm}} \approx H_Y^{\mathrm{norm}}$*, then DARS is close to* 1.

*Proof.* By construction, both EAS and NMI lie in $[0, 1]$, so their average lies in $[0, 1]$. If $\mathrm{NMI}(E; Y) = 0$ and $|H_E^{\mathrm{norm}} - H_Y^{\mathrm{norm}}|$ is large, then EAS is small and thus DARS is small. If $\mathrm{NMI}(E; Y)$ is close to 1 and the normalized entropies match, Lemma 1 gives $\mathrm{EAS} \approx 1$ and hence $\mathrm{DARS} \approx 1$. □

## B.3 Connection to Specialized yet Balanced Routing

Intuitively, an MoE router is desirable if it is both (i) *specialized* (experts focus on different subsets of labels) and (ii) *balanced* (expert usage reflects label frequencies and avoids collapse). We now give a simplified statement that connects high DARS to this desideratum under idealized assumptions.

**Lemma 3.** *Suppose the label distribution is uniform over* $\{1, \ldots, C\}$ *and the router implements an injective mapping from labels to experts, so that each label is routed to a disjoint subset of experts and the marginal expert usage is uniform. Then* $H_Y^{\mathrm{norm}} = H_E^{\mathrm{norm}} = 1$, $\mathrm{NMI}(E; Y) = 1$, *and hence* $\mathrm{DARS} = 1$.

*Proof.* Uniform labels give $H(Y) = \log C$ and thus $H_Y^{\mathrm{norm}} = 1$. If the mapping from labels to experts is injective and yields a uniform marginal over experts, then $H(E) = \log E$ and $H_E^{\mathrm{norm}} = 1$, so $\mathrm{EAS} = 1$. Under a deterministic injective mapping, the mutual information reaches its maximum and $\mathrm{NMI}(E; Y) = 1$. Therefore $\mathrm{DARS} = \frac{1}{2}(1 + 1) = 1$. □

In practice, real datasets and routers deviate from these idealized assumptions, but our experiments show that regimes with high DARS tend to exhibit (i) non-trivial specialization patterns in per-expert confusion matrices and (ii) expert-usage distributions whose entropy closely tracks the label entropy. This supports DARS as a compact measure of "specialized yet balanced" routing behavior, which we use throughout Sec. 4.

# C Additional Experimental Results

This appendix complements Sec. 4 with detailed ablation results and visualizations of routing behavior. All plots use the same trained models and hyperparameters as in the main tables.

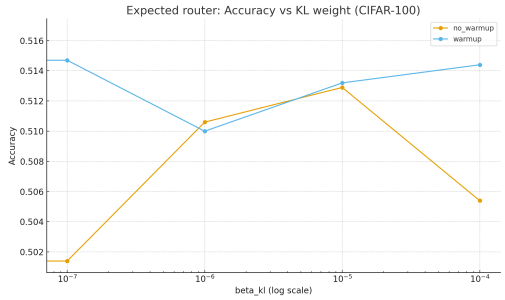
## C.1 Bayesian Router Ablations on CIFAR-100

Table 2 summarizes, for each router mode and warm-up configuration, the KL weight that best balances accuracy and calibration on CIFAR-100 with SVHN OOD.

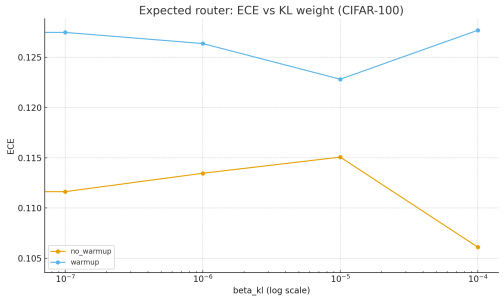
| Mode | Warm-up | Best $\beta_{\mathrm{KL}}$ | Acc | ECE | Throughput (img/s) |
|---|---|---|---|---|---|
| Expected | ✗ | $10^{-5}$ | 0.513 | 0.106 | $\approx 2600$ |
| Expected | ✓ | $10^{-6}$–$10^{-5}$ | 0.516 | 0.125 | $\approx 2600$ |
| MC | ✗ | $10^{-5}$ | 0.433 | 0.047 | $\approx 520$ |
| MC | ✓ | $10^{-6}$–$10^{-5}$ | 0.507 | 0.089 | $\approx 1250$ |

Table 2: CIFAR-100 ablation summary for the Bayesian router. For each mode and warm-up setting we show the KL weight with the best accuracy–calibration compromise, its test accuracy / ECE, and a representative throughput.

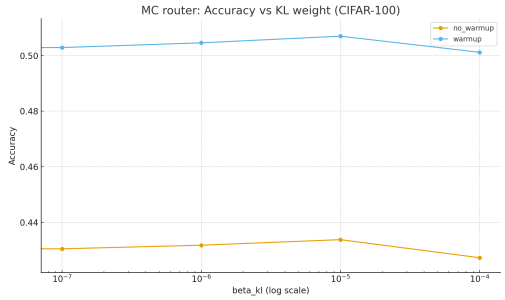
Figure 1 shows how KL weight and warm-up affect test accuracy and ECE for Expected and MC routers.



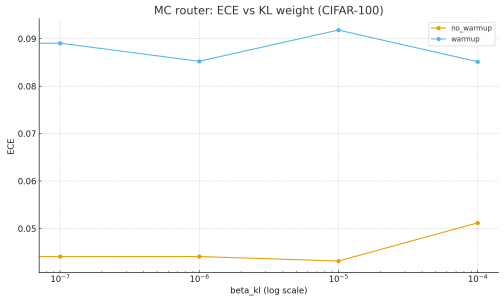

(a) Expected router: accuracy vs. $\beta_{\mathrm{KL}}$.




(b) Expected router: ECE vs. $\beta_{\mathrm{KL}}$.




(c) MC router: accuracy vs. $\beta_{\mathrm{KL}}$.




(d) MC router: ECE vs. $\beta_{\mathrm{KL}}$.

Figure 1: CIFAR-100 ablation curves for KL weight and warm-up under Expected and MC router modes.

For the Expected router (top row), accuracy without warm-up increases as $\beta_{\mathrm{KL}}$ moves from $10^{-7}$ to $10^{-5}$ and then decreases at $10^{-4}$, while ECE monotonically improves with larger KL. Warm-up shifts the accuracy curve upwards and flattens it across $\beta_{\mathrm{KL}}$, trading slightly worse calibration for robustness to the KL choice.

For the MC router (bottom row), no-warm-up configurations have similar, relatively low accuracy ($\approx 0.43$) but very small ECE, whereas warm-up boosts accuracy into the $0.50$–$0.51$ range and increases ECE to a moderate level. This matches the summary in Table 2.

Figure 2 plots the corresponding accuracy–throughput trade-off.

Expected-mode runs form a dense high-throughput cluster ($\approx 2500$–$2750$ images/s) with accuracy around $0.51$–$0.52$, plus a few slower but still competitive settings. MC-mode runs split into a low-throughput, low-accuracy cluster (no warm-up) and a mid-throughput, higher-accuracy cluster (with warm-up), illustrating the trade-off between a more Bayesian treatment of uncertainty and computational efficiency.

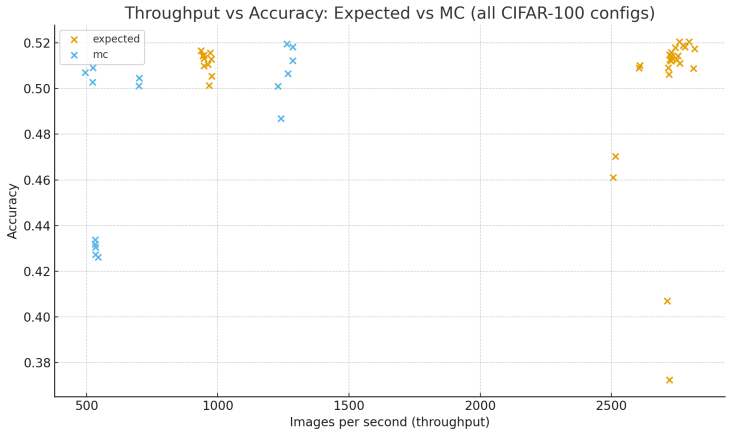
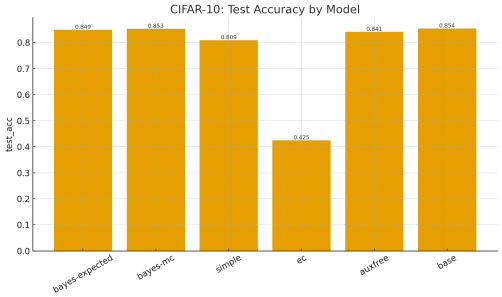


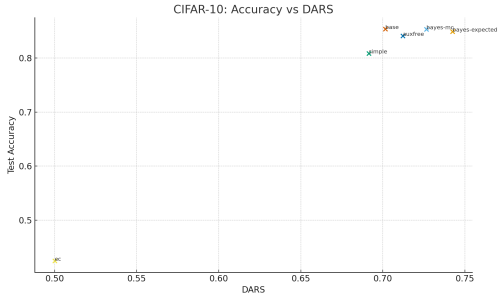

Figure 2: CIFAR-100 throughput vs. test accuracy for Expected vs. MC routers (each point is one hyperparameter setting).

## C.2 Model Comparison Plots: CIFAR-10

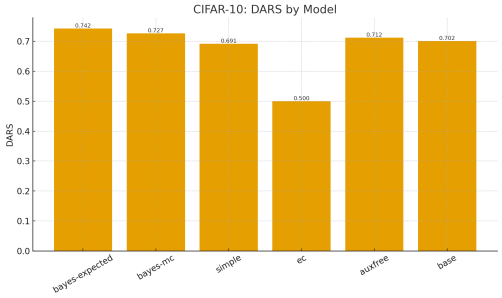
Figure 3 visualizes the CIFAR-10 comparison in terms of accuracy, DARS, and OOD confidence for all MoE variants.



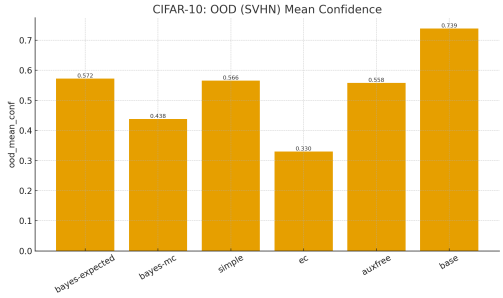

(a) CIFAR-10 test accuracy by model.




(b) CIFAR-10 accuracy vs. DARS.




(c) CIFAR-10 DARS by model.




(d) Mean max-softmax confidence on SVHN (OOD).

Figure 3: CIFAR-10 comparison of accuracy, routing quality (DARS), and OOD confidence across MoE architectures.

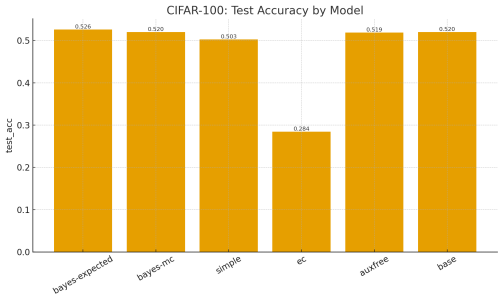
## C.3 Model Comparison Plots: CIFAR-100

Figure 4 shows the analogous plots for CIFAR-100.

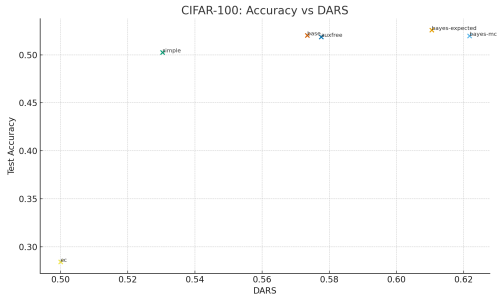
## C.4 Extended Training to 200 Epochs on CIFAR-100

To check whether our 100-epoch schedule is sufficient, we extend training of all MoE variants on CIFAR-100 (with SVHN as OOD) to 200 epochs using the same hyperparameters as in the main comparison. Table 3 reports the final test metrics at epoch 200.

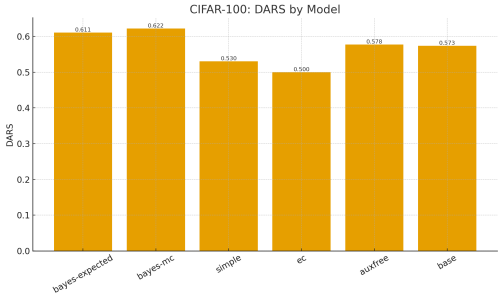
Compared with the 100-epoch results in Table 1, all models show very similar test accuracy at epoch 200: changes are within a few tenths of a percentage point, and in several cases the 200-epoch accuracy is slightly *lower* than the 100-epoch value (e.g. Bayes-Expected: $0.528 \rightarrow 0.526$, Bayes-MC: $0.522 \rightarrow 0.520$). At the



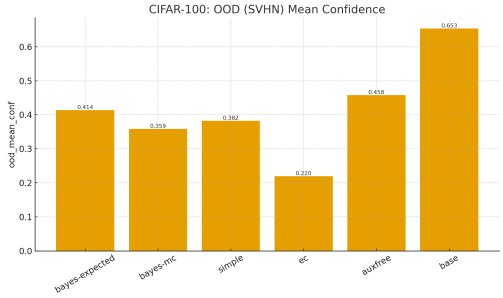

(a) CIFAR-100 test accuracy by model.




(b) CIFAR-100 accuracy vs. DARS.




(c) CIFAR-100 DARS by model.




(d) Mean max-softmax confidence on SVHN (OOD).

Figure 4: CIFAR-100 comparison of accuracy, routing quality (DARS), and OOD confidence across MoE architectures.

same time, calibration (ECE) and confidence statistics move only slightly, indicating that models have effectively converged by epoch 100 and further training mainly risks mild overfitting rather than bringing systematic gains. The relative ordering across Bayesian and non-Bayesian models is unchanged.

| Model | Bayes mode | Acc | NLL | ECE | DARS | ID $\mu_{\max}$ | OOD $\mu_{\max}$ |
|---|---|---|---|---|---|---|---|
| Bayes (Expected) | expected | 0.526 | 3.539 | 0.308 | 0.611 | 0.834 | 0.414 |
| Bayes (MC) | mc | 0.520 | 3.138 | 0.279 | 0.622 | 0.799 | 0.359 |
| Simple | – | 0.503 | 2.994 | 0.253 | 0.530 | 0.755 | 0.382 |
| EC | – | 0.284 | 3.369 | 0.086 | 0.500 | 0.370 | 0.220 |
| AuxFree | – | 0.519 | 2.860 | 0.268 | 0.578 | 0.785 | 0.458 |
| BASE | – | 0.520 | 2.881 | 0.270 | 0.573 | 0.790 | 0.653 |

Table 3: CIFAR-100 200-epoch results with SVHN as OOD. We report core performance, calibration (ECE), data-aware routing (DARS), and mean max-softmax confidence on ID/OOD.

## D Code example (including hyperparmeters)

Below is the code example of the RQ2 experiment, which includes training, testing, and logging based on set data workflow and model setup. Refer to `https://github.com/Pseudonymous-gdy/project` for detailed investigations.

---

```
"""
Moe_Compare_Models_LinearLog.py

Compare multiple MoE variants on CIFAR-10 / CIFAR-100 with SVHN as OOD.

Models:
  - Bayesian_NN_Moe (expected router, mc router; 4 presets total)
  - Simple_Moe      (Shazeer-style aux-loss MoE)
  - Expert_Choice   (EC MoE)
  - Aux_Free_Moe    (auxiliary-loss-free balancing)
  - BASE_Moe        (balanced assignment MoE)

Datasets (ID + OOD):
  - ID: cifar10, cifar100
  - OOD: svhn (via cifar100.get_ood_dataloader)

Metrics:
  - test_acc, test_nll, ECE
  - routing CV (per_expert_counts), NMI(expert; label), overflow rate
  - routing_entropy
  - ID / OOD max-softmax confidence stats (mean, std)

Bayes-aware extra metrics:
  - H_E_norm : normalized entropy of expert usage
  - H_Y_norm : normalized entropy of label distribution
  - EAS      : entropy alignment score 1 - |H_E_norm - H_Y_norm|
  - DARS     : Data-aware Routing Score = 0.5 * EAS + 0.5 * NMI

All training / evaluation progress is written linearly to:
  - stdout
  - moe_compare_train.txt
"""

import os
import sys
import math
import time
from typing import List, Dict, Any, Tuple, Optional

import numpy as np
import pandas as pd

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader

# ============================================================================
# Linear logging: everything goes to moe_compare_train.txt in chronological order
# ============================================================================

LOG_PATH = "moe_compare_train.txt"
_LOG_FH = None # lazy open, so that file is created only when first used


def log(msg: str) -> None:
    """
    Simple linear logger:
      - prepends timestamp
```

```
      - prints to stdout
      - appends to moe_compare_train.txt
    """
    global _LOG_FH
    if _LOG_FH is None:
        _LOG_FH = open(LOG_PATH, "a", encoding="utf-8")

    ts = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())
    line = f"{ts} | {msg}"
    print(line)
    _LOG_FH.write(line + "\n")
    _LOG_FH.flush()


# initial marker
log("==== Moe_Compare_Models (linear txt logging) started ====")

# ============================================================================
# Repo & model imports
# ============================================================================

# add repo root so that cifar10 / cifar100 / Moe.* can be imported
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

import cifar10
import cifar100

from Moe.Bayesian_NN_Moe import Bayesian_NN_Moe
from Moe.Simple_Moe import Simple_Moe
from Moe.Expert_Choice_Moe import Expert_Choice
from Moe.Aux_Free_Moe import Aux_Free_Moe
from Moe.BASE_Moe import BASE_Moe

# ============================================================================
# Global run configuration
# ============================================================================

NUM_EXPERTS = 16

# single global seed & single GPU
GLOBAL_SEED = 0

# default optimizer settings for non-Bayes models
DEFAULT_LR = 5e-4
DEFAULT_WEIGHT_DECAY = 2e-2

# main training epochs (warm-up epochs are extra)
EPOCHS = 200
BATCH_SIZE = 64
NUM_WORKERS = 2

# result file
RESULTS_CSV = "moe_compare_results.csv"

# which ID datasets & models to run
RUN_DATASETS = ["cifar100"]
RUN_SPECS = [
    dict(model="bayes", router_mode="expected"),
    dict(model="bayes", router_mode="mc"),
    dict(model="simple"),
    dict(model="ec"),
    dict(model="auxfree"),
    dict(model="base"),
]
```

```python
# ============================================================================
# Four Bayes presets (per dataset, per router_mode)
#   - 2 for CIFAR-10 (expected / mc)
#   - 2 for CIFAR-100 (expected / mc)
# ============================================================================

BAYES_PRESETS: Dict[str, Dict[str, Dict[str, Any]]] = {
    # ================= CIFAR-10 =================
    "cifar10": {
        # expected router: stable & fast
        "expected": dict(
            router_mode="expected",
            beta_kl=1e-6,
            tau=1.2,
            top_k=2,
            capacity=1.0,
            elbo_samples=2,
            use_control_variate=True,
            kl_anneal=5000,
            w_importance=0.0,
            w_load=0.0,
            warmup_epochs=5,
            lr=5e-4,
            weight_decay=1e-3,
        ),
        # mc ELBO: more Bayesian, needs stronger warm-up
        "mc": dict(
            router_mode="mc",
            beta_kl=1e-6,
            tau=1.2,
            top_k=2,
            capacity=1.0,
            elbo_samples=4,
            use_control_variate=True,
            kl_anneal=5000,
            w_importance=0.0,
            w_load=0.0,
            warmup_epochs=10,
            lr=5e-4,
            weight_decay=1e-3,
        ),
    },
    # ================= CIFAR-100 =================
    "cifar100": {
        # expected router preset from ECE / ACC / throughput analysis
        "expected": dict(
            router_mode="expected",
            beta_kl=1e-6,
            tau=1.2,
            top_k=2,
            capacity=1.0,
            elbo_samples=2,
            use_control_variate=True,
            kl_anneal=5000,
            w_importance=0.0,
            w_load=0.0,
            warmup_epochs=10,
            lr=5e-4,
            weight_decay=1e-3,
        ),
        # mc preset (needs warm-up)
        "mc": dict(
            router_mode="mc",
            beta_kl=1e-6,
            tau=1.2,
```

```
            top_k=2,
            capacity=1.0,
            elbo_samples=4,
            use_control_variate=True,
            kl_anneal=5000,
            w_importance=0.0,
            w_load=0.0,
            warmup_epochs=20,
            lr=5e-4,
            weight_decay=1e-3,
        ),
    },
}

# ============================================================================
# Utils: seeding, ECE, NMI, entropy alignment
# ============================================================================

def set_seed(seed: int) -> None:
    import random
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    log(f"[Seed] Set global seed to {seed}")


@torch.no_grad()
def ece_score(logits: torch.Tensor, y: torch.Tensor, n_bins: int = 15) -> float:
    probs = torch.softmax(logits, dim=1)
    conf, pred = probs.max(dim=1)
    acc = (pred == y).float()

    bins = torch.linspace(0, 1, n_bins + 1, device=logits.device)
    ece = torch.zeros(1, device=logits.device)
    for i in range(n_bins):
        m = (conf > bins[i]) & (conf <= bins[i + 1])
        if m.any():
            ece += torch.abs(acc[m].mean() - conf[m].mean()) * m.float().mean()
    return float(ece.item())


def specialization_nmi(hard_assign: np.ndarray,
                       labels: np.ndarray,
                       num_experts: int) -> float:
    x = np.asarray(hard_assign, dtype=int)
    y = np.asarray(labels, dtype=int)
    if x.size == 0:
        return float("nan")

    E = num_experts
    classes = np.unique(y)
    eps = 1e-12

    Px = np.array([(x == e).mean() for e in range(E)], dtype=np.float64) + eps
    Py = np.array([(y == c).mean() for c in classes], dtype=np.float64) + eps

    Hx = float(-(Px * np.log(Px)).sum())
    Hy = float(-(Py * np.log(Py)).sum())

    I = 0.0
    for i, e in enumerate(range(E)):
        for j, c in enumerate(classes):
```

```
            Pxy = float(((x == e) & (y == c)).mean()) + eps
            I += Pxy * (math.log(Pxy) - math.log(Px[i]) - math.log(Py[j]))
    return float(I / max(Hx, Hy))


def _entropy(p: np.ndarray) -> float:
    p = np.asarray(p, dtype=np.float64) + 1e-12
    return float(-(p * np.log(p)).sum())


def entropy_alignment_metrics(counts_np: np.ndarray,
                              labels_np: np.ndarray,
                              num_experts: int,
                              num_classes: int) -> Tuple[float, float, float]:
    if counts_np.sum() <= 0 or labels_np.size == 0:
        return float("nan"), float("nan"), float("nan")

    p_e = counts_np.astype(np.float64) / counts_np.sum()
    H_E = _entropy(p_e)
    H_E_norm = H_E / max(math.log(num_experts + 1e-12), 1e-12)

    label_counts = np.bincount(labels_np.astype(int),
                               minlength=num_classes).astype(np.float64)
    if label_counts.sum() <= 0:
        return H_E_norm, float("nan"), float("nan")
    p_y = label_counts / label_counts.sum()
    H_Y = _entropy(p_y)
    H_Y_norm = H_Y / max(math.log(num_classes + 1e-12), 1e-12)

    eas = 1.0 - abs(H_E_norm - H_Y_norm)
    eas = float(max(0.0, min(1.0, eas)))
    return H_E_norm, H_Y_norm, eas


# ============================================================================
# Dataset builders (ID + OOD)
# ============================================================================

def build_id_loaders(dataset_name: str,
                     batch_size: int,
                     num_workers: int,
                     download: bool = True) -> Tuple[DataLoader, DataLoader, int]:
    dataset_name = dataset_name.lower()
    log(f"[Data] Building ID dataloaders for {dataset_name}")
    if dataset_name == "cifar10":
        train_loader, test_loader, _ = cifar10.get_dataloaders(
            "2", batch_size=batch_size, num_workers=num_workers,
            data_dir="./data", download=download
        )
        num_classes = 10
    elif dataset_name == "cifar100":
        train_loader, test_loader, _ = cifar100.get_dataloaders(
            "2", batch_size=batch_size, num_workers=num_workers,
            data_dir="./data", download=download
        )
        num_classes = 100
    else:
        raise ValueError(f"Unsupported ID dataset: {dataset_name}")
    return train_loader, test_loader, num_classes


def build_ood_loader_svhn(batch_size: int,
                          num_workers: int,
                          download: bool = True) -> DataLoader:
    log("[Data] Building OOD dataloader: SVHN")
```

```
    return cifar100.get_ood_dataloader(
        "svhn", batch_size=batch_size, num_workers=num_workers,
        data_dir="./data", download=download
    )


# ================================================================================
# Model builders
# ================================================================================

def build_bayesian_moe(num_classes: int,
                       device: torch.device,
                       cfg: Dict[str, Any]) -> nn.Module:
    base = dict(
        num_experts=NUM_EXPERTS,
        num_features=32,
        hidden_size=64,
        top_k=2,
        tau=1.2,
        capacity=1.0,
        router_mode="expected",
        elbo_samples=2,
        use_control_variate=True,
        prior_var=10.0,
        beta_kl=1e-6,
        kl_anneal=5000,
        w_importance=0.0,
        w_load=0.0,
    )
    model_cfg = {**base, **cfg}

    log(
        "[Build] Bayesian_NN_Moe | "
        f"router_mode={model_cfg['router_mode']} beta_kl={model_cfg['beta_kl']} "
        f"tau={model_cfg['tau']} top_k={model_cfg['top_k']} "
        f"capacity={model_cfg['capacity']} elbo_samples={model_cfg['elbo_samples']}"
    )

    model = Bayesian_NN_Moe(
        num_experts=model_cfg["num_experts"],
        num_features=model_cfg["num_features"],
        output_size=num_classes,
        top_k=model_cfg["top_k"],

        router_prior_var=model_cfg.get("prior_var", 10.0),
        beta_kl=model_cfg.get("beta_kl", 1e-6),
        kl_anneal_steps=model_cfg.get("kl_anneal", 5000),
        router_temperature=model_cfg.get("tau", 1.2),

        capacity_factor=model_cfg.get("capacity", 1.0),
        overflow_strategy="drop",

        router_mode=model_cfg.get("router_mode", "expected"),
        elbo_samples=model_cfg.get("elbo_samples", 2),
        use_control_variate=model_cfg.get("use_control_variate", True),

        backbone_structure="resnet18",
        backbone_pretrained=False,
        hidden_size=model_cfg["hidden_size"],

        w_importance=model_cfg.get("w_importance", 0.0),
        w_load=model_cfg.get("w_load", 0.0),
    ).to(device)
    return model
```

```
def build_simple_moe(num_classes: int, device: torch.device) -> nn.Module:
    log("[Build] Simple_Moe")
    model = Simple_Moe(
        num_experts=NUM_EXPERTS,
        top_k=2,
        aux_loss_weight=0.0,
        backbone_structure="resnet18",
        backbone_pretrained=False,
        num_features=32,
        hidden_size=64,
        output_size=num_classes,
        per_token_noise=True,
        min_noise_scale=1e-2,
        w_importance=0.01,
        w_load=0.01,
        capacity_factor=1.5,
        overflow_strategy="drop",
        router_temperature=1.0,
    ).to(device)
    return model


def build_expert_choice(num_classes: int, device: torch.device) -> nn.Module:
    log("[Build] Expert_Choice_Moe")
    model = Expert_Choice(
        num_experts=NUM_EXPERTS,
        backbone_structure="resnet18",
        backbone_pretrained=False,
        num_features=32,
        hidden_size=64,
        output_size=num_classes,
        capacity_factor=1.0,
        use_noisy_scores=False,
    ).to(device)
    return model


def build_aux_free(num_classes: int, device: torch.device) -> nn.Module:
    log("[Build] Aux_Free_Moe")
    model = Aux_Free_Moe(
        num_experts=NUM_EXPERTS,
        top_k=2,
        backbone_structure="resnet18",
        backbone_pretrained=False,
        num_features=32,
        hidden_size=64,
        output_size=num_classes,
        per_token_noise=True,
        min_noise_scale=1e-3,
        router_temperature=1.2,
        capacity_factor=1.5,
        overflow_strategy="drop",
        bias_lr=0.1,
        ema_decay=0.9,
        bias_clip=2.0,
        update_bias_in_eval=False,
    ).to(device)
    return model


def build_base_moe(num_classes: int, device: torch.device) -> nn.Module:
    log("[Build] BASE_Moe")
    model = BASE_Moe(
        num_experts=NUM_EXPERTS,
```

```
        backbone_structure="resnet18",
        backbone_pretrained=False,
        num_features=32,
        hidden_size=64,
        output_size=num_classes,
        assign_mode="hungarian",
        use_noisy_scores=False,
        min_noise_scale=1e-2,
    ).to(device)
    return model


# ============================================================================
# Training & evaluation (with linear logging)
# ============================================================================

def _run_bayes_warmup(
    model: nn.Module,
    train_loader: DataLoader,
    device: torch.device,
    optimizer: torch.optim.Optimizer,
    warmup_epochs: int,
) -> None:
    if warmup_epochs <= 0:
        return

    log(f"[Bayes] Warm-up for {warmup_epochs} epoch(s)")

    if hasattr(model, "train_one_epoch_warmup") and callable(
        getattr(model, "train_one_epoch_warmup")
    ):
        for ep in range(1, warmup_epochs + 1):
            t0 = time.time()
            avg_loss, train_acc = model.train_one_epoch_warmup(
                loader=train_loader,
                optimizer=optimizer,
                device=device,
                criterion=None,
                max_batches=None,
            )
            dt = time.time() - t0
            log(
                f"[Warmup explicit] epoch={ep:03d} "
                f"loss={avg_loss:.4f} acc={train_acc:.4f} time={dt:.1f}s"
            )
        return

    old_router_mode = getattr(model, "router_mode", None)
    old_beta_kl = getattr(model, "beta_kl", None)
    old_kl_anneal = getattr(model, "kl_anneal_steps", None)

    if hasattr(model, "beta_kl"):
        model.beta_kl = 0.0
    if hasattr(model, "kl_anneal_steps"):
        model.kl_anneal_steps = 0
    if hasattr(model, "router_mode"):
        model.router_mode = "expected"

    log("[Bayes] Using implicit warm-up: beta_kl=0, router_mode='expected'")

    for ep in range(1, warmup_epochs + 1):
        t0 = time.time()
        avg_loss, train_acc = model.train_one_epoch(
            loader=train_loader,
            optimizer=optimizer,
```

```
            device=device,
            criterion=None,
            max_batches=None,
        )
        dt = time.time() - t0
        log(
            f"[Warmup implicit] epoch={ep:03d} "
            f"loss={avg_loss:.4f} acc={train_acc:.4f} time={dt:.1f}s"
        )

    if old_router_mode is not None and hasattr(model, "router_mode"):
        model.router_mode = old_router_mode
    if old_beta_kl is not None and hasattr(model, "beta_kl"):
        model.beta_kl = old_beta_kl
    if old_kl_anneal is not None and hasattr(model, "kl_anneal_steps"):
        model.kl_anneal_steps = old_kl_anneal


def train_model(model: nn.Module,
                train_loader: DataLoader,
                device: torch.device,
                epochs: int,
                lr: float,
                weight_decay: float,
                model_name: str,
                bayes_warmup_epochs: int = 0) -> None:
    optimizer = torch.optim.AdamW(model.parameters(), lr=lr,
         weight_decay=weight_decay)
    log(
        f"[Train] model={model_name} epochs(main)={epochs} "
        f"warmup_epochs={bayes_warmup_epochs} lr={lr} wd={weight_decay}"
    )

    if model_name == "bayes" and bayes_warmup_epochs > 0:
        _run_bayes_warmup(
            model=model,
            train_loader=train_loader,
            device=device,
            optimizer=optimizer,
            warmup_epochs=bayes_warmup_epochs,
        )

    for ep in range(1, epochs + 1):
        t0 = time.time()
        avg_loss, train_acc = model.train_one_epoch(
            loader=train_loader,
            optimizer=optimizer,
            device=device,
            criterion=None,
            max_batches=None,
        )
        dt = time.time() - t0
        log(
            f"[Train-main] epoch={ep:03d} "
            f"loss={avg_loss:.4f} acc={train_acc:.4f} time={dt:.1f}s"
        )


@torch.no_grad()
def evaluate_moe(model: nn.Module,
                 test_loader: DataLoader,
                 device: torch.device,
                 num_experts: int,
                 num_classes: int) -> Dict[str, Any]:
    log("[Eval] Evaluating on ID test set")
```

```
model.eval()

logits_all: List[torch.Tensor] = []
labels_all: List[torch.Tensor] = []

counts = torch.zeros(num_experts, dtype=torch.float64)
dropped = torch.zeros(num_experts, dtype=torch.float64)
entropies: List[float] = []

expert_assign_list: List[np.ndarray] = []
label_list: List[np.ndarray] = []

for xb, yb in test_loader:
    xb = xb.to(device, non_blocking=True)
    yb = yb.to(device, non_blocking=True)

    logits, aux = model(xb, return_aux=True)

    logits_all.append(logits.detach().cpu())
    labels_all.append(yb.detach().cpu())

    pec = aux.get("per_expert_counts", None)
    ofd = aux.get("overflow_dropped", None)
    if pec is not None:
        counts += pec.detach().cpu().to(torch.float64)
    if ofd is not None:
        dropped += ofd.detach().cpu().to(torch.float64)

    if "routing_entropy" in aux:
        entropies.append(float(aux["routing_entropy"].detach().cpu().item()))

    tki = aux.get("topk_idx", None)
    if tki is not None:
        ha = tki[:, 0].detach().cpu().numpy()
        expert_assign_list.append(ha)
        label_list.append(yb.detach().cpu().numpy())

if not logits_all:
    log("[Eval] Empty logits list; returning NaNs")
    return {k: float("nan") for k in
            ["acc", "nll", "ece", "cv", "nmi", "overflow",
             "routing_entropy", "H_E_norm", "H_Y_norm", "EAS", "DARS"]}

logits = torch.cat(logits_all, dim=0)
labels = torch.cat(labels_all, dim=0)

acc = (logits.argmax(dim=1) == labels).float().mean().item()
nll = F.cross_entropy(logits, labels).item()
ece = ece_score(logits, labels)

counts_np = counts.numpy()
total_kept = counts_np.sum()
total_dropped = dropped.numpy().sum()
total_attempt = total_kept + total_dropped

if total_kept > 0:
    cv = float(counts_np.std() / (counts_np.mean() + 1e-8))
    overflow_rate = float(total_dropped / (total_attempt + 1e-8))
else:
    cv = float("nan")
    overflow_rate = float("nan")

routing_entropy = float(np.mean(entropies)) if entropies else float("nan")

if expert_assign_list and label_list:
```

```
        ha = np.concatenate(expert_assign_list, axis=0)
        lb = np.concatenate(label_list, axis=0)
        nmi = specialization_nmi(ha, lb, num_experts)
    else:
        nmi = float("nan")

    H_E_norm, H_Y_norm, EAS = entropy_alignment_metrics(
        counts_np=counts_np,
        labels_np=labels.numpy(),
        num_experts=num_experts,
        num_classes=num_classes,
    )

    if not (math.isnan(EAS) or math.isnan(nmi)):
        DARS = 0.5 * EAS + 0.5 * nmi
    else:
        DARS = float("nan")

    log(
        "[Eval-ID] acc={:.4f} nll={:.3f} ece={:.4f} "
        "cv={:.3f} nmi={:.3f} overflow={:.4f} H_E_norm={:.3f} "
        "H_Y_norm={:.3f} EAS={:.3f} DARS={:.3f}".format(
            acc, nll, ece, cv, nmi, overflow_rate,
            H_E_norm, H_Y_norm, EAS, DARS
        )
    )

    return dict(
        acc=float(acc),
        nll=float(nll),
        ece=float(ece),
        cv=cv,
        nmi=nmi,
        overflow=overflow_rate,
        routing_entropy=routing_entropy,
        H_E_norm=H_E_norm,
        H_Y_norm=H_Y_norm,
        EAS=EAS,
        DARS=DARS,
    )


@torch.no_grad()
def compute_confidence_stats(model: nn.Module,
                             loader: DataLoader,
                             device: torch.device,
                             tag: str) -> Dict[str, Any]:
    log(f"[Eval-{tag}] Computing max-softmax confidence stats")
    model.eval()
    softmax = nn.Softmax(dim=1)
    conf_list: List[torch.Tensor] = []

    for xb, _ in loader:
        xb = xb.to(device, non_blocking=True)
        logits = model(xb, return_aux=False)
        probs = softmax(logits)
        max_conf, _ = probs.max(dim=1)
        conf_list.append(max_conf.detach().cpu())

    if not conf_list:
        log(f"[Eval-{tag}] Empty loader; confidence stats NaN")
        return {"mean_conf": float("nan"), "std_conf": float("nan"), "n": 0}

    all_conf = torch.cat(conf_list, dim=0).numpy()
    mean_conf = float(all_conf.mean())
```

```
    std_conf = float(all_conf.std())
    n = int(all_conf.size)

    log(
        f"[Eval-{tag}] mean_conf={mean_conf:.4f} std_conf={std_conf:.4f} n={n}"
    )

    return {
        "mean_conf": mean_conf,
        "std_conf": std_conf,
        "n": n,
    }


# ============================================================================
# Main experiment loop (no CLI; run directly)
# ============================================================================

def main():
    # device
    if torch.cuda.is_available():
        device = torch.device("cuda:0")
    else:
        device = torch.device("cpu")

    log(f"[Device] Using device: {device}")
    if device.type == "cuda":
        log(f"[Device] cuda.is_available()={torch.cuda.is_available()}")
        log(f"[Device] GPU count={torch.cuda.device_count()}")
        log(
            f"[Device] current device index={torch.cuda.current_device()} | "
            f"name={torch.cuda.get_device_name(0)}"
        )

    all_results: List[Dict[str, Any]] = []

    for ds in RUN_DATASETS:
        log("=" * 80)
        log(f"[Run] Dataset={ds} (ID) with SVHN as OOD")
        log("=" * 80)

        set_seed(GLOBAL_SEED)

        train_loader, test_loader, num_classes = build_id_loaders(
            ds, batch_size=BATCH_SIZE, num_workers=NUM_WORKERS, download=True
        )
        ood_loader = build_ood_loader_svhn(
            batch_size=BATCH_SIZE, num_workers=NUM_WORKERS, download=True
        )

        for spec in RUN_SPECS:
            model_name = spec["model"]
            router_mode = spec.get("router_mode", "")

            set_seed(GLOBAL_SEED) # ensure comparability

            if model_name == "bayes":
                cfg = BAYES_PRESETS[ds][router_mode]
                lr = cfg["lr"]
                weight_decay = cfg["weight_decay"]
                warmup_epochs = cfg["warmup_epochs"]

                log(
                    f"[Run] Model=Bayes router_mode={router_mode} on {ds} | "
                    f"beta_kl={cfg['beta_kl']} tau={cfg['tau']} "
```

```
            f"top_k={cfg['top_k']} capacity={cfg['capacity']} "
            f"elbo_samples={cfg['elbo_samples']} "
            f"warmup_epochs={warmup_epochs} lr={lr} wd={weight_decay}"
        )

        model = build_bayesian_moe(num_classes, device, cfg)
        train_model(
            model=model,
            train_loader=train_loader,
            device=device,
            epochs=EPOCHS,
            lr=lr,
            weight_decay=weight_decay,
            model_name="bayes",
            bayes_warmup_epochs=warmup_epochs,
        )

    elif model_name == "simple":
        log(f"[Run] Model=Simple_Moe on {ds}")
        model = build_simple_moe(num_classes, device)
        train_model(
            model=model,
            train_loader=train_loader,
            device=device,
            epochs=EPOCHS,
            lr=DEFAULT_LR,
            weight_decay=DEFAULT_WEIGHT_DECAY,
            model_name="simple",
            bayes_warmup_epochs=0,
        )

    elif model_name == "ec":
        log(f"[Run] Model=Expert_Choice on {ds}")
        model = build_expert_choice(num_classes, device)
        train_model(
            model=model,
            train_loader=train_loader,
            device=device,
            epochs=EPOCHS,
            lr=DEFAULT_LR,
            weight_decay=DEFAULT_WEIGHT_DECAY,
            model_name="ec",
            bayes_warmup_epochs=0,
        )

    elif model_name == "auxfree":
        log(f"[Run] Model=Aux_Free_Moe on {ds}")
        model = build_aux_free(num_classes, device)
        train_model(
            model=model,
            train_loader=train_loader,
            device=device,
            epochs=EPOCHS,
            lr=DEFAULT_LR,
            weight_decay=DEFAULT_WEIGHT_DECAY,
            model_name="auxfree",
            bayes_warmup_epochs=0,
        )

    elif model_name == "base":
        log(f"[Run] Model=BASE_Moe on {ds}")
        model = build_base_moe(num_classes, device)
        train_model(
            model=model,
            train_loader=train_loader,
```

```
        device=device,
        epochs=EPOCHS,
        lr=DEFAULT_LR,
        weight_decay=DEFAULT_WEIGHT_DECAY,
        model_name="base",
        bayes_warmup_epochs=0,
    )

else:
    raise ValueError(f"Unknown model spec: {model_name}")

# ===== Evaluation (ID + OOD) =====
ev = evaluate_moe(
    model=model,
    test_loader=test_loader,
    device=device,
    num_experts=NUM_EXPERTS,
    num_classes=num_classes,
)
id_stats = compute_confidence_stats(model, test_loader, device, tag="ID")
ood_stats = compute_confidence_stats(model, ood_loader, device,
     tag="OOD")

if model_name == "bayes":
    row_lr = cfg["lr"]
    row_wd = cfg["weight_decay"]
    row_warm = cfg["warmup_epochs"]
    row_router_mode = router_mode
else:
    row_lr = DEFAULT_LR
    row_wd = DEFAULT_WEIGHT_DECAY
    row_warm = 0
    row_router_mode = ""

row = dict(
    dataset=ds,
    model=model_name,
    bayes_router_mode=row_router_mode,
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    lr=row_lr,
    weight_decay=row_wd,
    bayes_warmup_epochs=row_warm,
    # ID metrics
    test_acc=ev["acc"],
    test_nll=ev["nll"],
    ece=ev["ece"],
    cv=ev["cv"],
    nmi=ev["nmi"],
    overflow=ev["overflow"],
    routing_entropy=ev["routing_entropy"],
    H_E_norm=ev["H_E_norm"],
    H_Y_norm=ev["H_Y_norm"],
    EAS=ev["EAS"],
    DARS=ev["DARS"],
    id_mean_conf=id_stats["mean_conf"],
    id_std_conf=id_stats["std_conf"],
    id_n=id_stats["n"],
    # OOD metrics
    ood_dataset="svhn",
    ood_mean_conf=ood_stats["mean_conf"],
    ood_std_conf=ood_stats["std_conf"],
    ood_n=ood_stats["n"],
)
all_results.append(row)
```

```
            log(
                "[Summary] [{ds}][{model}] acc={acc:.4f} nll={nll:.3f} "
                "ece={ece:.4f} cv={cv:.3f} nmi={nmi:.3f} overflow={ov:.4f} "
                "H_E_norm={He:.3f} H_Y_norm={Hy:.3f} EAS={eas:.3f} "
                "DARS={dars:.3f} id_mean_conf={idc:.4f} "
                "ood_mean_conf={odc:.4f}".format(
                    ds=ds,
                    model=(model_name if model_name != "bayes"
                           else f"bayes-{router_mode}"),
                    acc=row["test_acc"],
                    nll=row["test_nll"],
                    ece=row["ece"],
                    cv=row["cv"],
                    nmi=row["nmi"],
                    ov=row["overflow"],
                    He=row["H_E_norm"],
                    Hy=row["H_Y_norm"],
                    eas=row["EAS"],
                    dars=row["DARS"],
                    idc=row["id_mean_conf"],
                    odc=row["ood_mean_conf"],
                )
            )

    # store and print
    df = pd.DataFrame(all_results)
    df.to_csv(RESULTS_CSV, index=False)
    log(f"[Result] Saved results to {RESULTS_CSV}")

    cols = [
        "dataset", "model", "bayes_router_mode", "bayes_warmup_epochs",
        "test_acc", "test_nll", "ece",
        "cv", "nmi", "overflow",
        "H_E_norm", "H_Y_norm", "EAS", "DARS",
        "id_mean_conf", "ood_mean_conf",
    ]
    log("\n" + df[cols].to_string(index=False))
    log("==== Moe_Compare_Models finished ====")

    # close log file
    global _LOG_FH
    if _LOG_FH is not None:
        _LOG_FH.close()
        _LOG_FH = None


if __name__ == "__main__":
    main()
```