A Calibrated Initialization for Gaussian Mixture Layers

Jay Paek

Updated: June 30, 2026

1 Gaussian Mixture Layers

Ordinary feedforward layers. Fix an input $x \in \mathbb{R}^d$ (e.g. $d = 784$ flattened pixels). A standard two-layer MLP uses two weight matrices, with no per-neuron output weight in the hidden layer:

$$z \;=\; \varphi(W_1 x) \;\in\; \mathbb{R}^m, \qquad \text{logits} \;=\; W_2 z \;\in\; \mathbb{R}^C,$$

where $W_1 \in \mathbb{R}^{m\times d}$, $W_2 \in \mathbb{R}^{C\times m}$, and φ is ReLU applied coordinatewise. Neuron j contributes one hidden coordinate $z_j = \varphi(\langle W_{1,j,\cdot}, x\rangle)$, and the hidden layer returns the full m-vector z. Each activation is kept for the next matrix multiply; nothing is averaged across neurons.

Mean-field neuron pairs. The GM paper [1] does not start from this single-matrix hidden layer. Instead it parametrizes a *two-layer block* (the hidden directions together with their readout into the next layer) via pairs (ω_j, β_j). Here $\beta_j \in \mathbb{R}^d$ is the analogue of row j of W_1, while ω_j is the analogue of how column j of W_2 scales that neuron's activation. One neuron's contribution to a single scalar output coordinate is

$$f_j(x) \;=\; \omega_j\, \varphi(\langle \beta_j, x\rangle) \;\in\; \mathbb{R}.$$

Summing over m such neurons and dividing by m gives

$$h(x) \;=\; \frac{1}{m}\sum_{j=1}^{m} f_j(x) \;=\; \frac{1}{m}\sum_{j=1}^{m} \omega_j\, \varphi(\langle \beta_j, x\rangle),$$

so $h\colon \mathbb{R}^d \to \mathbb{R}$. The $1/m$ factor is the mean-field normalization: it turns a finite sum over neurons into an average, which a standard hidden layer does not do. In the infinite-width limit one can then replace the sum by an integral over a weight distribution ρ:

$$h_\rho(x) \;=\; \int \omega\, \varphi(\langle \beta, x\rangle)\, \rho(d\omega, d\beta) \;\in\; \mathbb{R}.$$

In this view a layer is itself a distribution ρ over neuron parameters (ω, β), and a finite network is the special case where ρ is a pile of m point masses. An ordinary hidden layer $\varphi(W_1 x)$ never performs this averaging; doing so would destroy the m-dimensional hidden state that W_2 needs.

GM layers in practice. A **Gaussian mixture (GM) layer** keeps the mean-field view but replaces the m point masses with K smooth blobs: a mixture of K Gaussians $\rho = \frac{1}{K}\sum_k \mathcal{N}(\mu_k, \Sigma_k)$. Instead of storing thousands of discrete neurons, you store K "neuron clouds" and integrate over

each one in closed form. Conceptually it behaves like an infinitely wide two-layer block summarized by K Gaussians.

For multi-class (or multi-unit hidden) output the paper uses a reduced parametrization (diagonal β-covariance, affine readout $\omega = U\beta + v$), giving the closed-form forward map

$$h_\theta(x) \;=\; \frac{1}{K}\sum_{k=1}^{K} \mathbb{E}_{\beta\sim\mathcal{N}(\mu_k^\beta,\,\mathrm{diag}(\sigma_k^2))}[(U_k\beta + v_k)\,\varphi(\langle\beta, x\rangle)] \;\in\; \mathbb{R}^C,$$

where C is the layer width (e.g. $C = 100$ for a hidden GM layer, $C = 10$ for logits). So $h_\theta\colon \mathbb{R}^d \to \mathbb{R}^C$: each coordinate is one output unit, built from the same mean-field average-over-components recipe. Trainable parameters are $\theta = (\mu^\beta, \sigma, U, v)$ per component. The ReLU–Gaussian expectation has an exact analytic form, so no Monte-Carlo sampling is needed at train or eval time.

Parallel to a plain linear layer. The GM parametrization maps onto the standard two-matrix block as follows:

- μ^β behaves like the rows of W_1: the directions each neuron reads from x.
- U, v behave like W_2, mapping each neuron's activation to the C outputs. A plain hidden layer has no separate ω, because its readout lives in the next matrix W_2.
- σ is the new part: the spread of each neuron cloud. A classical linear layer is the degenerate $\sigma \to 0$ limit (point-mass neurons).
- The factor $1/K$ (like $1/m$ above) is the mean-field average over components, and it is absent from the forward pass $\varphi(W_1 x)$ of a standard hidden layer.

2 The open problem: initialization

The original paper trains GM layers from a simple random init (`BL`, for baseline): draw $\mu^\beta, U, v \sim \mathcal{N}(0, \gamma^2)$ and set $\sigma = \gamma$, with $\gamma = 1/2$ by default. Empirically the GM network reaches final accuracy comparable to a fully connected net, but it trains noticeably slower than a Kaiming-initialized FC baseline. The authors note this themselves: the observation "calls for future investigation on better initialization schemes for GM layers" [1]. `KUMAR` solves this.

3 The `KUMAR` method

`KUMAR` stands for **K**aiming **U**nlabeled **M**ixture **A**ctivation **R**escaling: Kaiming-style fan-in weights, followed by a label-free forward pass on real inputs that rescales the readout so initial activations land at the right scale. The fan-in geometry is standard; what `KUMAR` adds is control over the realized output scale, which fan-in variance alone does not pin down for a GM layer.

1. **Fan-in init.** With input dimension d, set $g = \sqrt{2/d}$. Draw $\mu \sim \mathcal{N}(0, g^2)$, $U \sim \mathcal{N}(0, g^2)$, $v \sim \mathcal{N}(0, g^2)$, and $\sigma = g$.
2. **Calibrate the readout.** Take 2048 training images (labels not used), run the exact closed-form forward to get the initial logits, and rescale only the readout parameters so the logit standard deviation hits a target:

$$\text{scale} = \frac{\texttt{target\_logit\_std}}{\text{std(logits)}}, \qquad U \mathrel{*}= \text{scale}, \quad v \mathrel{*}= \text{scale},$$

with `target_logit_std` $= 0.5$. For stacks this is done layer-by-layer: layer 2 is calibrated on layer 1's hidden activations.

Why it works. Plain fan-in scaling controls parameter variance, which is the right target for a sampled ReLU network. A GM layer is different. It computes a closed-form expectation inside each Gaussian component and then averages over K components, and that two-stage operation means the realized logit scale can land far from where the parameter variances suggest. `BL` pays for the mismatch with a large, badly scaled initial transient: the loss starts very high, and many early steps go to renormalizing the output scale before any feature learning happens. `KUMAR` instead measures the initial logit scale on real inputs and corrects it directly, so training starts in the right range. The construction is label-free, needs no pretraining or clustering, and costs essentially no gradient steps.

4 Experimental setup

The main experiment reuses the paper's stacked-GM setup so the comparison stays controlled; the columns below mark what was kept versus added.

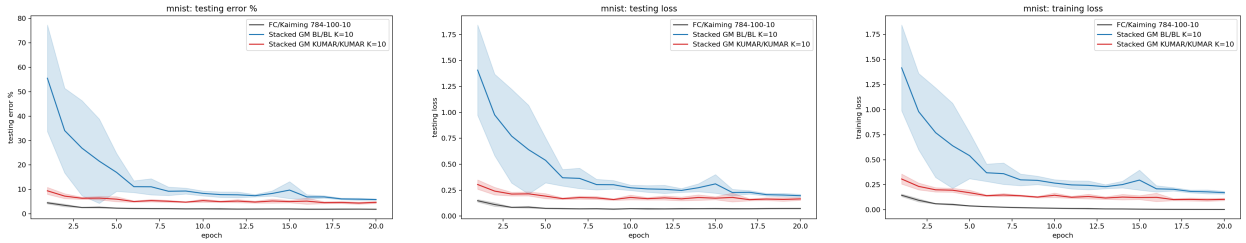
	Kept from the paper	**Added here**
Architecture	two stacked GM layers, $784 \to 100 \to 10$, $K = 10$ each	—
Forward	exact closed-form ReLU–Gaussian (no MC)	—
Optimizer	vanilla SGD, LR $= 1.0$ for σ, 0.1 otherwise, batch 64	—
Data	MNIST, vectorized, zero-mean / unit-std	—
Protocol	error bars over 5 seeds	paired seeds (0–4), identical data order
Methods	`BL` (baseline; paper random, $\gamma = 0.5$)	`KUMAR`; FC/Kaiming reference
Budget	—	20 epochs

The three arms are: a Kaiming FC net ($784 \to 100 \to 10$) as an external reference, `BL` on both GM layers, and `KUMAR` on both GM layers. (Prior init-screen evidence at K=20, 30 epochs, seeds 5–14 had `KUMAR` beating `BL` on best error, final-smoothed error, and AUC in 10/10 paired seeds on both MNIST and Fashion-MNIST; the stacked K=10 run below is consistent with that.)

5 Results

Training dynamics

The per-epoch curves in Fig. 1 show the basic pattern. `BL` starts with a large, high-variance transient (test error near 55% on average, and up to ∼77% across seeds) and crawls down to ∼5.8% only late in training. `KUMAR` starts near 9–10%, drops to ∼5% within a couple of epochs, and settles around 4.3% with a much tighter seed band. The FC reference sits below both throughout.

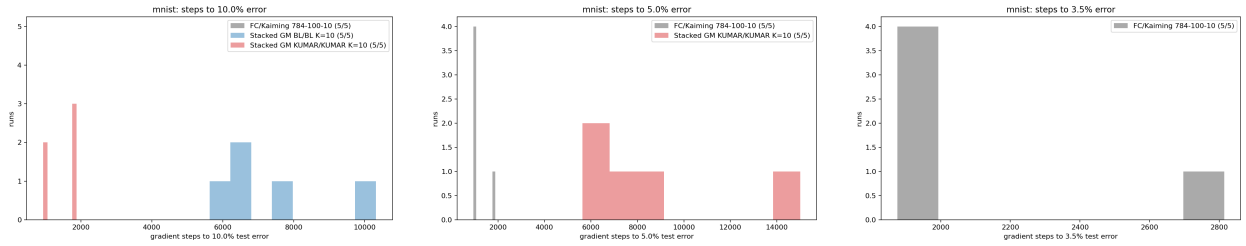


(a) Test error (b) Test loss (c) Train loss

Figure 1: Per-epoch curves (mean ± band over 5 seeds). `KUMAR` (red) removes the large `BL` (blue) startup transient and converges faster, with a tighter band; FC/Kaiming (grey) remains the floor.

Time-to-accuracy

The thresholds in Fig. 2 quantify the speedup. At 10% error, `KUMAR` arrives in ∼1–2k gradient steps while `BL` needs ∼6–10k. At 5%, all `KUMAR` seeds get there (∼6–15k steps), but `BL` does not reach it within the budget. At 3.5% only the FC reference qualifies; the gap to a tuned dense net is real and remains open.

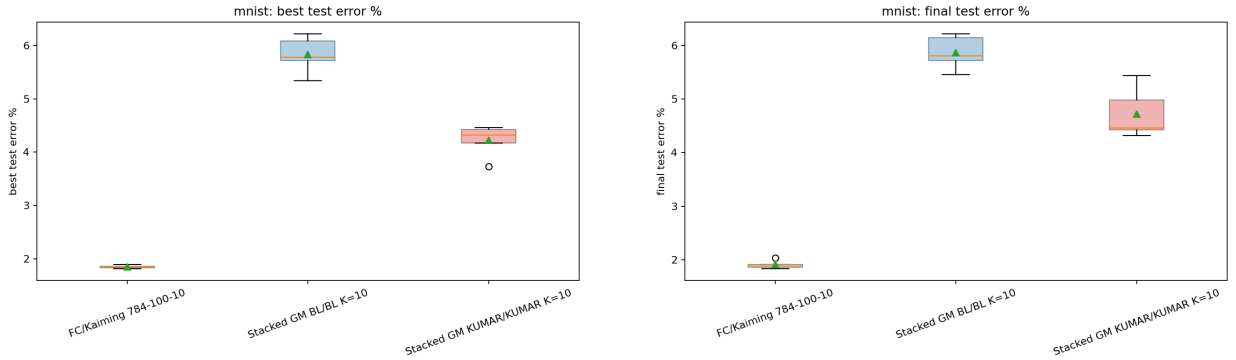


(a) Steps to 10% (b) Steps to 5% (c) Steps to 3.5%

Figure 2: Gradient steps to first hit each test-error threshold (counts of runs reaching it shown in legend).

Final and best error

The endpoint distributions are summarized in Fig. 3. Across seeds, `KUMAR` ends clearly below `BL` on both best-so-far and final test error, and its best-error spread is tighter. Neither GM arm closes the gap to the plain FC net.



(a) Best test error (`BL` $\approx 5.85\%$, `KUMAR` $\approx 4.2\%$)

(b) Final test error (`BL` $\approx 5.9\%$, `KUMAR` $\approx 4.7\%$)

Figure 3: Distribution over the 5 paired seeds. Green triangle = mean, orange line = median.

Component propagation (why it trains faster)

The step-count speedup leaves a geometric trace, visible in the separate single-layer diagnostic of Fig. 4. Following Fig. 4 of the paper, we track the K=5 component means of a single GM layer on MNIST over 20 epochs. Each panel plots 2σ covariance ellipses in a fixed PCA basis (the top two PCs of the final mixture), with opacity increasing over epochs.

Under `BL`, each ellipse is already large at epoch 0 and barely moves for the rest of training; the network spends its early budget rescaling the readout rather than relocating components. Under `KUMAR`, the components start in a tight cluster near the origin and propagate outward along distinct trajectories from the first epoch, with ellipses growing and rotating as they specialize. The epoch budget is the same, but `KUMAR` turns gradient steps into feature motion right away, whereas `BL` defers that motion until the output scale has been corrected. Final test error is similar in this single-layer snapshot (6.68% vs. 6.49%), yet the plot still exposes the training-speed gap: `KUMAR` is already exploring feature space while `BL` is effectively static.

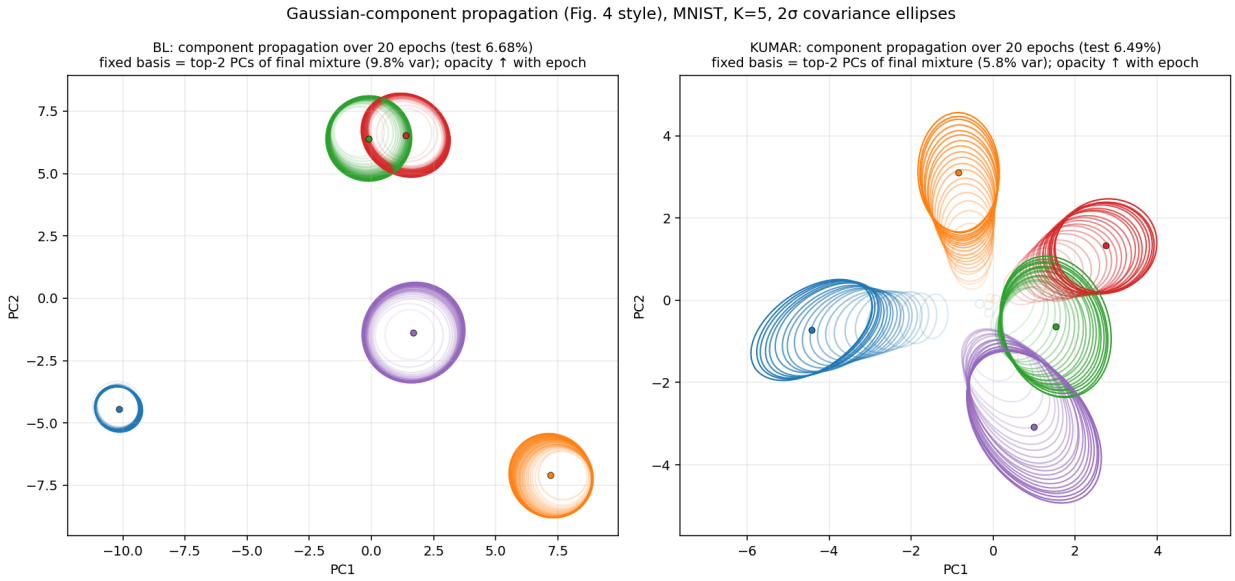


Figure 4: Gaussian-component propagation (MNIST, K=5, 20 epochs). `BL` (left): components remain nearly fixed after a badly scaled start. `KUMAR` (right): components propagate from a calibrated origin, turning early steps into feature learning rather than output rescaling.

6 Takeaway

`KUMAR` is a practical answer to the paper's call for better GM initialization [1]. On our stacked K=10 MNIST setup it beats `BL` on every measure we tracked: time to threshold, seed-to-seed variance, best and final error, and how soon the components begin propagating rather than waiting out a long output-rescaling transient. It does this with a single label-free forward pass and no gradient steps, by calibrating the initial logit scale on data after a Kaiming fan-in init.

GM layers remain useful when the parameter budget is tight, at the cost of not quite matching dense-net accuracy or speed. `KUMAR` does not close that gap entirely, but it removes the main training pathology tied to the paper's default init and makes the GM stack much easier to train in practice.

References

[1] S. Chewi, P. Rigollet, Y. Yan. *Gaussian mixture layers for neural networks*. arXiv:2508.04883, 2025. https://arxiv.org/pdf/2508.04883