EM as a Neural Operator Layer

Concept Overview

This project implements a **differentiable EM algorithm** for a Gaussian Mixture Model (GMM) as a **PyTorch layer**. It treats EM as a **neural operator**: data is transformed into a latent responsibility field, iteratively refined, and decoded back into a reconstructed data distribution using `torch.logsumexp`.

The architecture draws an analogy with **Fourier Neural Operators (FNOs)**:

Stage	**Fourier Neural Operator (FNO)**	**EM-as-Layer (Neural Operator)**
Transform (Encoder)	Applies FFT to map the input to the spectral domain	E-step computes posterior responsibilities, projecting data into latent mixture space.
Operator Core	Learns spectral multipliers (kernels in Fourier domain)	Iterative EM updates with learnable residual corrections via MLPs.
Inverse Transform (Decoder)	Uses IFFT to reconstruct the signal in the spatial domain	Uses `torch.logsumexp` to reconstruct data-space likelihood from μ and Σ.

Table 1: Conceptual analogy between Fourier Neural Operators and the EM-as-Layer framework.

This formulation views EM as a *statistical neural operator*, mapping data distributions into structured probabilistic representations.

Mathematical Formulation

1. Log-Likelihood of GMM

For data $X = \{x_n\}_{n=1}^N$ and mixture parameters $\Theta = \{\alpha_k, \mu_k, \Sigma_k\}_{k=1}^K$:

$$\log p(X|\Theta) = \sum_{n=1}^{N} \log \left(\sum_{k=1}^{K} \alpha_k \, \mathcal{N}(x_n|\mu_k, \Sigma_k) \right)$$

2. Transform (E-step)

Responsibilities r_{nk} are computed as:

$$r_{nk} = \frac{\alpha_k \, \mathcal{N}(x_n|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \alpha_j \, \mathcal{N}(x_n|\mu_j, \Sigma_j)}$$

Implementation:

```
lg = _log_gauss(X, mu, Sigma)          # [B, N, K]
w_log = torch.log_softmax(alpha, -1)   # [B, K]
log_r = lg + w_log[:, None, :]         # [B, N, K]
r = torch.softmax(log_r, dim=-1)       # [B, N, K]
```

This acts as the **transform**: mapping data into a latent mixture-coefficient space.

3. Operator Core (Iterative EM with Learnable Residuals)

Each EM iteration computes:

$$N_k = \sum_n r_{nk}, \quad \mu_k = \frac{1}{N_k}\sum_n r_{nk} x_n, \quad \Sigma_k = \frac{1}{N_k}\sum_n r_{nk}(x_n - \mu_k)(x_n - \mu_k)^\top$$

Then applies learnable residual corrections:

$$\mu \leftarrow \mu + \mathrm{MLP}_\mu(\mu), \qquad \Sigma \leftarrow \Sigma + \mathrm{MLP}_\Sigma(\Sigma)$$

4. Decoder (Mixture Reconstruction via `torch.logsumexp`)

After T refinement steps, the decoder reconstructs the data distribution using the refined parameters:

$$\log p(x_n) = \log \sum_k \exp(\log \pi_k + \log \mathcal{N}(x_n|\mu_k, \Sigma_k))$$

Implementation:

```
w_log = torch.log_softmax(alpha, dim=-1)
log_probs = _log_gauss(X, mu, Sigma)
ll = torch.logsumexp(log_probs + w_log[:, None, :], dim=-1).sum(dim=1)
```

The decoder is not the M-step — the M-step is internal, while the decoder reconstructs the observable density.

Data Flow Summary

Symbol	Meaning	Shape
X	Input data batch	$[B, N, D]$
α	Mixture logits	$[B, K]$
μ	Means	$[B, K, D]$
Σ	Covariances	$[B, K, D, D]$
r	Responsibilities	$[B, N, K]$

Table 2: Tensor notation and dimensional layout.

Neural Operator View

$$XE - steprIterativeEM + LearnableMLPs(\alpha, \mu, \Sigma) logsumexp(Decoder) \log p(X)$$

Encoder: E-step (projects into responsibility space)
Operator: Iterative EM refinement (latent evolution)
Decoder: `logsumexp` (reconstructs data likelihood)

Training Objective

Maximize log-likelihood:

$$\mathcal{L} = E[\log p(X|\Theta)]$$

Implementation:

```
loss = -ll.mean()
loss.backward()
```

Role	Function
Transform	E-step: projects data $\rightarrow$ latent
Operator	Unrolled EM + learnable MLPs
Decoder	`logsumexp`: reconstructs distribution from μ, Σ
Training	Maximize log-likelihood end-to-end

Table 3: Summary of EM-as-Layer functional roles.

Summary

EM-as-a-Layer = Neural Operator over probability distributions.

This framework generalizes EM into a continuous, trainable operator bridging statistical inference and deep representation learning.