DNN Inference for Beta Using Mixture_Gaussian

From Single Gaussian to Mixture of Gaussians: Improving Posterior Approximation

In Phase 1, we used a neural network to learn a **single Gaussian distribution**:

$$\mathcal{N}(\mu, \sigma^2)$$

to approximate the posterior:

$$p(\theta \mid y)$$

Now we use Mixture Density Network (MDN)

The neural network now outputs:

$$\begin{aligned}
&\text{- A set of mixture weights } \pi_1, \ldots, \pi_K, \quad \text{where } \sum_{k=1}^{K} \pi_k = 1 \\
&\text{- A set of component means } \mu_1, \ldots, \mu_K \\
&\text{- A set of component log-variances } \log \sigma_1^2, \ldots, \log \sigma_K^2
\end{aligned}$$

The predicted posterior becomes:

$$p_{\text{NN}}(\theta \mid y) = \sum_{k=1}^{K} \pi_k(y) \cdot \mathcal{N}(\theta \mid \mu_k(y), \sigma_k^2(y))$$

Loss Function: Negative Log-Likelihood of Mixture Model

Given a set of posterior samples $\{\theta_i\}_{i=1}^{N}$ for a specific observed (y), we define the loss as:

$$\mathcal{L}_{\text{Mixture}}(y) = -\frac{1}{N} \sum_{i=1}^{N} \log \left[\sum_{k=1}^{K} \pi_k(y) \cdot \mathcal{N}(\theta_i \mid \mu_k(y), \sigma_k^2(y)) \right]$$

In [1]:
```
# ——— Imports ———
import random
from collections import defaultdict
import math
import numpy as np
from scipy.stats import beta, binom, norm, gamma, poisson
import torch
import torch.nn as nn
import torch.optim as optim
```

```
# ───── Reproducibility ─────
SEED = 123
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# ───── Mixture Density Network ─────
class MixtureDensityNN(nn.Module):
    def __init__(self, num_components: int = 8):
        super().__init__()
        self.num_components = num_components
        # Feature extractor backbone
        self.backbone = nn.Sequential(
            nn.Linear(1, 24),
            nn.ReLU(),
            nn.Linear(24, 25),
            nn.ReLU(),
            nn.Linear(25, 16),
            nn.ReLU(),
        )
        # Heads for mixture parameters
        self.logits_head   = nn.Linear(16, num_components)   # raw mixture l
        self.means_head    = nn.Linear(16, num_components)   # Gaussian mear
        self.log_vars_head = nn.Linear(16, num_components)   # log(σ_k²) for

    def forward(self, x: torch.Tensor):
        """
        x: Tensor of shape (batch_size, 1)
        returns:
          weights:  Tensor (batch_size, K) — mixture weights π_k
          means:    Tensor (batch_size, K) — component means μ_k
          log_vars: Tensor (batch_size, K) — log-variances log(σ_k²)
        """
        features = self.backbone(x)
        logits   = self.logits_head(features)
        weights  = torch.softmax(logits, dim=-1)
        means    = self.means_head(features)
        log_vars = self.log_vars_head(features)
        return weights, means, log_vars

import math
import torch
from torch import Tensor

# Cell 2a: Corrected mixture_negative_log_likelihood
def mixture_negative_log_likelihood(
    theta_samples: Tensor,
    weights:       Tensor,
    means:         Tensor,
    log_vars:      Tensor
) -> Tensor:
    """
    Compute mean negative log-likelihood under a Gaussian mixture.
```

```
    - theta_samples: (N,1) tensor of true θ values
    - weights, means, log_vars: (1,K) tensors
    """
    # Precompute constant log(2π)
    log_2pi = math.log(2 * math.pi)

    # σ²_k
    vars_ = torch.exp(log_vars)
    # Expand θ to (N, K)
    theta_exp = theta_samples.expand(-1, vars_.shape[1])

    # Gaussian log-probabilities per component:
    #   -(1/2)[log(2πσ²_k) + ((θ - μ_k)² / σ²_k)]
    log_probs = -0.5 * (
        log_vars + log_2pi +
        (theta_exp - means)**2 / vars_
    )

    # Weighted mixture log-likelihood: log Σ_k π_k · N(...)
    log_weighted = torch.log(weights) + log_probs
    ll_per_sample = torch.logsumexp(log_weighted, dim=1)  # (N,)

    # Return mean negative log-likelihood
    return -ll_per_sample.mean()

model = MixtureDensityNN(num_components=8)

# ───── Prior & Sampling Settings ─────
alpha_prior  = 2.0
beta_prior   = 5.0
num_trials   = 100
num_samples  = 10**6

# Sample θ ~ Beta(alpha, beta), then y ~ Binomial(num_trials, θ)
theta_np = beta.rvs(alpha_prior, beta_prior, size=num_samples)
y_np     = binom.rvs(n=num_trials, p=theta_np)

# ───── Organize samples by observed y ─────
theta_by_y = defaultdict(list)
for θ, y in zip(theta_np, y_np):
    theta_by_y[y].append(θ)

# ───── Compute posterior statistics ─────────
posterior_means = {}
posterior_vars  = {}
for y_val in range(num_trials + 1):
    thetas = np.array(theta_by_y.get(y_val, [0.0]))
    posterior_means[y_val] = thetas.mean()
    posterior_vars[y_val]  = thetas.var() if thetas.size > 1 else 0.0
```

We train our model, then visualise with plots for differing observed y:

In [2]:
```
optimizer   = optim.Adam(model.parameters(), lr=1e-3)
n_epochs    = 500
```

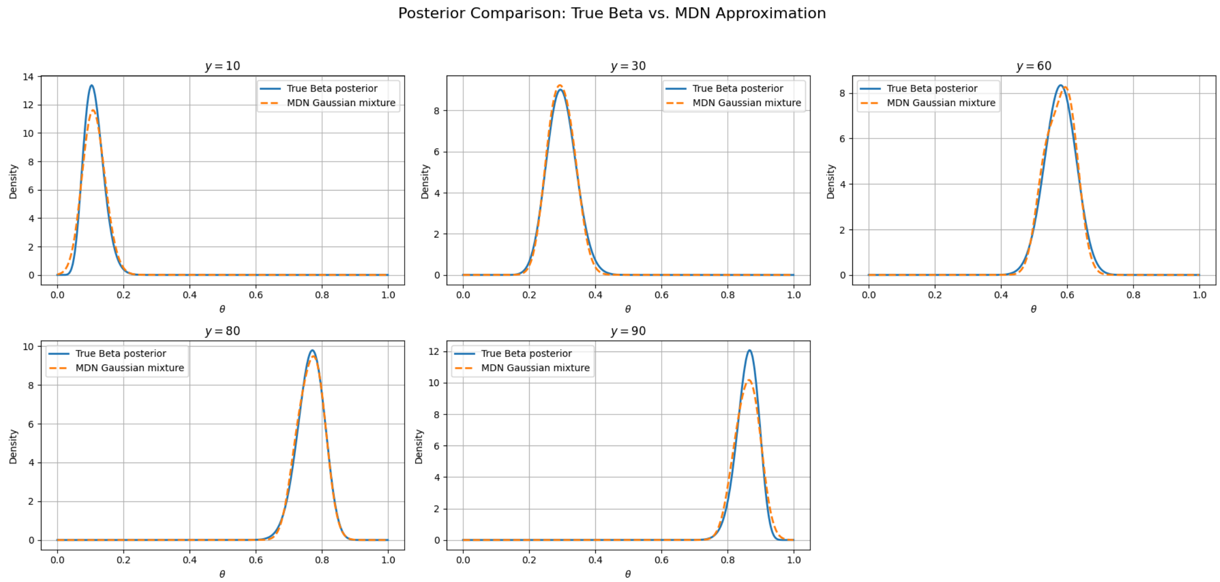
```
for epoch in range(n_epochs):
    model.train()
    total_loss = 0.0

    # Loop over each observed y and its θ-samples
    for y_val, thetas in theta_by_y.items():
        if len(thetas) < 2:
            continue  # skip if we don't have at least 2 samples

        # 1) normalize y → x input
        x_in = torch.tensor([[y_val / num_trials]], dtype=torch.float32)

        # 2) forward pass
        weights, means, log_vars = model(x_in)

        # 3) compute NLL on all θ samples for this y
        theta_tensor = torch.tensor(thetas, dtype=torch.float32).view(-1, 1)
        loss_i = mixture_negative_log_likelihood(theta_tensor,
                                                 weights, means, log_vars)

        total_loss += loss_i

    # 4) backward + step
    optimizer.zero_grad()
    total_loss.backward()
    optimizer.step()

    if epoch % 50 == 0:
        print(f"[Epoch {epoch:3d}] total_loss = {total_loss.item():.4f}")
```

```
[Epoch   0] total_loss = 102.4597
[Epoch  50] total_loss = 85.9368
[Epoch 100] total_loss = -18.2413
[Epoch 150] total_loss = -162.8556
[Epoch 200] total_loss = -168.1500
[Epoch 250] total_loss = -170.3476
[Epoch 300] total_loss = -171.9186
[Epoch 350] total_loss = -173.0599
[Epoch 400] total_loss = -173.7963
[Epoch 450] total_loss = -174.2781
```

In [5]:

```
import matplotlib.pyplot as plt

# 1. Configuration
y_obs_list = [10, 30, 60, 80, 90]                    # which y’s to plot
n_cols     = 3
n_rows     = math.ceil(len(y_obs_list) / n_cols)   # use math from Cell 1

# 2. θ-grid for density evaluation
theta_range = np.linspace(0.001, 0.999, 300)

# 3. Prepare figure
fig, axes = plt.subplots(n_rows, n_cols,
                         figsize=(6 * n_cols, 4.5 * n_rows))
axes_flat = axes.flatten()
```

```
# 4. Loop & plot
for idx, y_obs in enumerate(y_obs_list):
    # 4a. MDN output for normalized y
    x_input = torch.tensor([[y_obs / num_trials]],
                           dtype=torch.float32)
    with torch.no_grad():
        weights, means, log_vars = model(x_input)

    # 4b. True Beta posterior: Beta(α + y, β + n_trials − y)
    true_dist = beta(alpha_prior + y_obs,
                     beta_prior  + num_trials - y_obs)
    true_pdf  = true_dist.pdf(theta_range)

    # 4c. MDN Gaussian-mixture approximation
    approx_pdf = np.zeros_like(theta_range)
    for k in range(model.num_components):
        π_k = weights[0, k].item()
        μ_k = means[0,   k].item()
        σ_k = np.sqrt(np.exp(log_vars[0, k].item()))
        approx_pdf += π_k * norm.pdf(theta_range,
                                     loc=μ_k,
                                     scale=σ_k)

    # 4d. Plot on subplot
    ax = axes_flat[idx]
    ax.plot(theta_range, true_pdf,
            label="True Beta posterior", lw=2)
    ax.plot(theta_range, approx_pdf,
            '--', label="MDN Gaussian mixture", lw=2)
    ax.set(title=f"$y = {y_obs}$",
           xlabel="$\\theta$",
           ylabel="Density")
    ax.legend()
    ax.grid(True)

# 5. Remove unused axes
for ax in axes_flat[len(y_obs_list):]:
    fig.delaxes(ax)

# 6. Final touches
fig.suptitle("Posterior Comparison: True Beta vs. MDN Approximation",
             fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



DNN Inference for Gamma Using Mixture_Gaussian

Now we use the same idea for y sampled from a poisson distribution (not binomial)

In [6]:
```
# 1. Prior & sampling settings
alpha_gp    = 2.0
beta_gp     = 3.0
num_samples = 10**6
num_trials  = 100

# 2. Draw samples: θ ~ Gamma(α, scale=1/β), y ~ Poisson(num_trials·θ)
theta_np = gamma.rvs(a=alpha_gp, loc=0, scale=1/beta_gp, size=num_samples)
# Sum of n_trials Poisson(θ) is Poisson(n_trials·θ)
y_np     = poisson.rvs(mu=theta_np * num_trials)

# 3. Group θ-samples by observed y
theta_by_y = defaultdict(list)
for θ, y in zip(theta_np, y_np):
    theta_by_y[y].append(θ)

# 4. Compute empirical posterior means & variances
posterior_means = {}
posterior_vars  = {}
for y_val, thetas in theta_by_y.items():
    arr = np.array(thetas)
    posterior_means[y_val] = arr.mean()
    posterior_vars[y_val]  = arr.var() if arr.size > 1 else 0.0

# 5. Instantiate MDN & optimizer
model     = MixtureDensityNN(num_components=8)
optimizer = optim.Adam(model.parameters(), lr=1e-2)

# 6. Training loop
n_epochs = 500
for epoch in range(n_epochs):
    model.train()
```

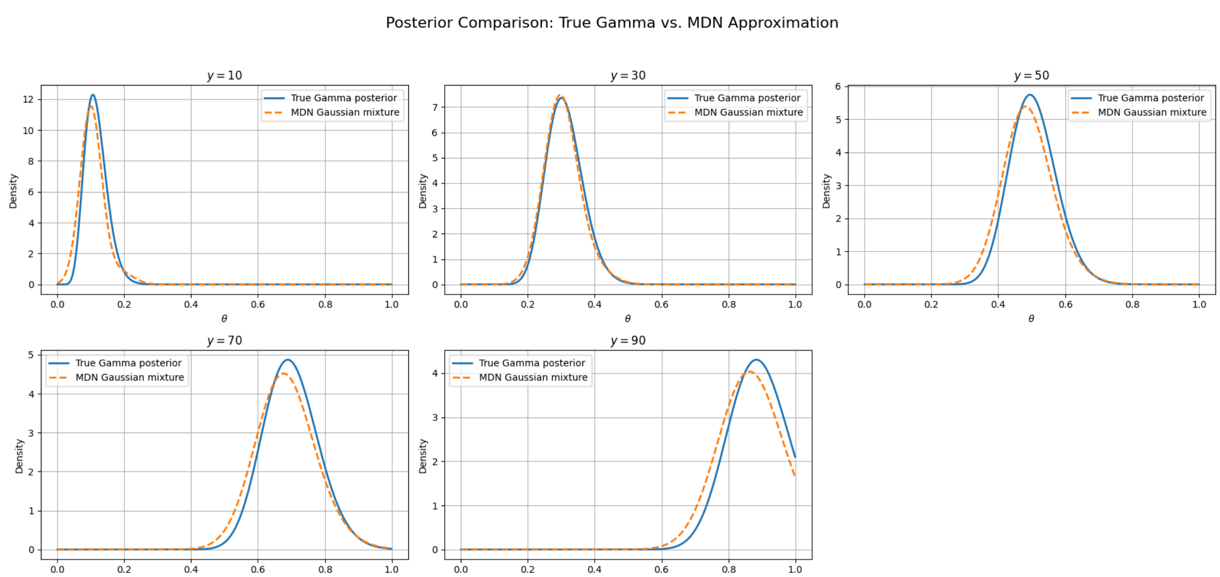
```
    total_loss = 0.0

    for y_val, thetas in theta_by_y.items():
        if len(thetas) < 2:
            continue

        # a) Normalize y → x input
        x_in = torch.tensor([[y_val / num_trials]], dtype=torch.float32)

        # b) Forward pass: get mixture params
        weights, means, log_vars = model(x_in)

        # c) Prepare θ-tensor and compute negative log-likelihood
        theta_tensor = torch.tensor(thetas, dtype=torch.float32).view(-1, 1)
        loss_i = mixture_negative_log_likelihood(theta_tensor,
                                                 weights,
                                                 means,
                                                 log_vars)
        total_loss += loss_i

    # d) Backprop & update
    optimizer.zero_grad()
    total_loss.backward()
    optimizer.step()

    if epoch % 100 == 0:
        print(f"[Epoch {epoch:4d}] loss = {total_loss.item():.4f}")
```

```
[Epoch    0] loss = 1509.5538
[Epoch  100] loss = -281.5116
[Epoch  200] loss = -256.7239
[Epoch  300] loss = -281.9673
[Epoch  400] loss = -287.2285
```

We now visualise with plots for differing observed y:

In [7]:
```
# 2. Configuration
y_obs_list = [10, 30, 50, 70, 90]
n_cols     = 3
n_rows     = math.ceil(len(y_obs_list) / n_cols)

# 3. θ-grid for density evaluation
theta_range = np.linspace(0.001, 0.999, 300)

# 4. Set up subplots
fig, axes = plt.subplots(n_rows, n_cols,
                         figsize=(6 * n_cols, 4.5 * n_rows))
axes_flat = axes.flatten()

# 5. Loop over observed y's and plot
for idx, y_obs in enumerate(y_obs_list):
    # a) MDN output for normalized y
    x_in = torch.tensor([[y_obs / num_trials]],
                        dtype=torch.float32)
    with torch.no_grad():
        weights, means, log_vars = model(x_in)
```

```
    # b) True Gamma posterior: Gamma(α + y, scale = 1 / (β + n_trials))
    true_dist = gamma(a=alpha_gp + y_obs,
                      scale=1.0 / (beta_gp + num_trials))
    true_pdf  = true_dist.pdf(theta_range)

    # c) MDN Gaussian-mixture approximation
    approx_pdf = np.zeros_like(theta_range)
    for k in range(model.num_components):
        π_k   = weights[0, k].item()
        μ_k   = means[0,   k].item()
        σ_k   = np.sqrt(np.exp(log_vars[0, k].item()))
        approx_pdf += π_k * norm.pdf(theta_range,
                                     loc=μ_k,
                                     scale=σ_k)

    # d) Plot on subplot
    ax = axes_flat[idx]
    ax.plot(theta_range, true_pdf,
            label="True Gamma posterior", lw=2)
    ax.plot(theta_range, approx_pdf,
            '--', label="MDN Gaussian mixture", lw=2)
    ax.set(title=f"$y = {y_obs}$",
           xlabel="$\\theta$",
           ylabel="Density")
    ax.legend()
    ax.grid(True)

# 6. Remove any unused axes
for extra_ax in axes_flat[len(y_obs_list):]:
    fig.delaxes(extra_ax)

# 7. Final layout tweaks
fig.suptitle("Posterior Comparison: True Gamma vs. MDN Approximation",
             fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



Summary

The Mixture of Gaussians (MoG) model accurately captures the true Beta and Gamma posteriors — even for extreme y observations—where a single-Gaussian fit would fail. However, this expressivity comes at a computational cost:

- **Epochs**: 500 does well, we get extremely good fits at about 1000 epochs
- **Learning rate**: 1×10^{-3}
- **Runtime**: Approximately 1 minute per 100 epochs (≈ 5 minutes total)

Overall, the single-Gaussian model trains faster but is limited to unimodal posteriors, whereas the MDN handles multi-modal shapes with longer runtimes.