Writeup4_yifan-tutorial-original.ipynb
File

Can you help me understand what this neural network architecture is doing? It looks bit opaque to me. Can you describe to me: 1) what the architecture is, 2) what the goal of the objective function is, and 3) if you think this architecture is designed well? I was given this code by a colleague, but I don't quite know what it does... I was asked to improve its performance without changing the architecture, but it's not entirely clear to me what the goal even is. I think it's currently being run on simulated data in this script.

Here's a clean read of what your colleague's notebook is doing and why it looks "GAN-ish". I'll stick to your three asks and then add a short "how to improve without changing the architecture" section.

1) What the architecture is

It's actually three very simple modules wired together in a training loop:

1. **BinaryClassifier**

```python
class BinaryClassifier(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.linear = nn.Linear(input_dim, 1)  # no hidden layers

    def forward(self, x):
        return self.linear(x)  # logits for D
```

- Purpose: predict the binary variable **D** from covariates.
- Used in two ways:
    - **Baseline**: trained on **X only** → estimates β_1 (effect of X on D).
    - **With latent Z**: trained on **[X, Ẑ]** during generator training → estimates β_1 and γ_1-like effect through Ẑ.

2. **NBModel** (Negative Binomial regression)

```python
class NBModel(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.fc = nn.Linear(input_dim, output_dim, bias=False)  # log μ = W^T inputs

    def forward(self, X_aug):
        log_mu = self.fc(X_aug)
        return log_mu
```

- Purpose: predict **Y** (count vector, p genes) with an NB likelihood.

- Inputs:
  - **Baseline**: **[X, D]** only.
  - **With latent Z**: **[X, D, Ẑ]** during generator training.
- Loss: NB negative log-likelihood with a per-gene dispersion **α** (stored/learned elsewhere in the script).

3. **Generator** (to infer the latent confounder Z)

```python
class Generator(nn.Module):
    def __init__(self, input_dim, z_dim):
        super().__init__()
        self.fc = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, z_dim),
            nn.Sigmoid()  # outputs in (0,1)
        )

    def forward(self, x, d, y):
        return self.fc(torch.cat([x, d, y], dim=1))
```

- Purpose: infer **Ẑ** from the **joint (X, D, Y)**.
- It turns Ẑ into a pseudo-binary variable via a **straight-through estimator**:

  ```python
  probs = generator(X, D, Y)
  z_bin = (probs > 0.5).float()
  Z_hat = z_bin + probs - probs.detach()  # gradient = gradient(probs), forward = binary
  ```

- Then it **re-trains** tiny predictors on the fly:
  - BinaryClassifier on **[X, Ẑ] → D**
  - NBModel on **[X, D, Ẑ] → Y**
- The idea is: if Ẑ is good, both D and Y become more predictable than the **baseline** models that *do not* have Z.

Data-generating story in the notebook (simulated):

- X ∈ ℝ^{n×d}, Z ∈ {0,1}^{n×k} with P(Z=1)=p_z.
- D ~ Bernoulli(sigmoid($X\beta_1$ + $Z\gamma_1$)).
- Y ~ NB with log-mean: log μ = [X, D] β_2 + Z γ_2 and dispersion α.
  So Z is a shared latent that affects both D and Y.

2) What the objective is trying to do

During **generator training**, the script computes three pieces:

1. **Y likelihood term**
   NB negative log-likelihood using NBModel trained on **[X, D, Ẑ]**:

- `nll_loss = nb_loss(log_mu(X,D,Ẑ), alpha, Y)`
  (lower is better)

2. **D likelihood term**
   BCEWithLogits using BinaryClassifier trained on **[X, Ẑ]**:
   - `bce_loss = BCEWithLogits(logits(X,Ẑ), D)`
     (lower is better)
3. **"DBPR" terms = deviance-based partial R^2 improvements**
   The code computes, for both D and Y, the **improvement vs a baseline** model that does *not* use Z:
   - For **D**: compare deviance of baseline logits **(X→D)** vs current logits **([X,Ẑ]→D)** → **w′ ∈ [0,1]** approximately.
   - For **Y**: compare deviance of baseline **([X,D]→Y)** vs current **([X,D,Ẑ]→Y)** → **v′ ∈ [0,1]** approximately.
   - Intuition: higher w′ and v′ ⇒ Ẑ explains variation that X (and D) alone could not.

The **total loss** used to update **the generator** is:

```python
total_loss = nll_loss + bce_scale * bce_loss + (mean(w′) + mean(v′))
```

> ⚠️ Important: because training *minimizes* `total_loss`, **adding** `(w′ + v′)` actually **penalizes** improvements, which contradicts the stated intent of "rewarding" Ẑ that improves prediction over baseline.
> To align with the intent, you'd usually **subtract** these terms, e.g.:
>
> ```ini
> total_loss = nll_loss + bce_scale * bce_loss - λ*(mean(w′)+mean(v′))
> ```
>
> or equivalently add them with a **negative weight**. As written, the generator is pushed *away* from useful Ẑ.

3) Is the design sensible?

What's good

- A clear, coherent generative story: Z affects both D and Y. Using both targets to learn Ẑ is principled.
- Proper count model for Y (NB with log link), matching how Y was simulated.
- The baseline vs. enhanced (with Ẑ) comparison is a smart way to quantify whether Ẑ adds value.

What needs attention

- **Sign of the DBPR reward** (critical): as implemented, it discourages improvement. If you're not allowed to change the objective *form*, you can still set the DBPR coefficient negative; otherwise the generator will prefer "bad" Ẑ.

- **Loss scaling / units**: NB NLL (sum over n×p) can dwarf BCE (n) and DBPR (unitless). Without normalization or weighting, the generator may ignore some terms. At minimum, divide each term by its natural count (e.g., number of samples/genes) and introduce tunable weights.
- **Training noise / instability**: the loop **re-trains tiny predictors from scratch every epoch** using Ẑ. That injects variance into the generator's signal. A standard trick is multiple predictor steps per generator step (or warm-start / partially freeze predictors) to make the learning signal steadier—this doesn't change the architecture.
- **No intercepts**: NBModel has `bias=False`. In the sim, there's no intercept, so it matches; but on real data you'd want an intercept. (That's architecture, so leave it—just note it.)
- **Very high LR in one place**: NB predictor uses `lr=18e-2`. That's aggressive and can hurt convergence; easy to tune down without changing structure.
- **Straight-through binarization**: works, but a Gumbel-Sigmoid (Concrete) relaxation with temperature annealing often trains more smoothly—again, a training trick, not an architectural change.

How to improve performance (without changing the architecture)

These are "training & tuning" changes only:

1. **Fix the reward direction (within the current loss)**
   Keep the same three terms but put a **negative** weight on `(w′+v′)`. If you can't flip the sign globally, set a hyperparameter `dbpr_scale = -c` and use:

   ```ini
   total_loss = nll_loss + bce_scale * bce_loss + dbpr_scale * (mean(w′)+mean(v′))
   ```

   Start with `c ∈ {0.1, 0.5, 1.0}`.
2. **Normalize and weight losses**
   - Use per-sample/per-gene averages:
     `nll_loss /= (n*p)`, `bce_loss /= n`.
   - Then tune `bce_scale` and `dbpr_scale` so gradients are on comparable magnitudes.
3. **Stabilize the inner predictors**
   - **Warm-start** them (do not reinitialize each epoch); or
   - Do **K predictor steps per 1 generator step** (e.g., K=5), so predictors track Ẑ better.
   - Add **weight decay** (small, e.g., 1e-4) and reduce the NB predictor LR (e.g., 1e-3 to 5e-3).
4. **Batching**
   - Move from full-batch to **mini-batch** training for the generator and predictors (consistent batching across the three). This often improves generalization and smooths optimization.
5. **Temperature-annealed relaxation for Ẑ** (no arch change)

- Replace hard threshold with **Gumbel-Sigmoid** during training (keep the straight-through idea). Start temperature at ~2→0.5 over epochs. Same module; only how you sample/relax $\hat{Z}$ changes.

6. **Diagnostics you already have—use them to tune**
   - Track **AUROC** for D (baseline vs with $\hat{Z}$) and **NB deviance** for Y over epochs.
   - Plot the progression of mean coefficients (you already log "Y gamma", "D gamma", etc.) and ensure they converge toward the "true" values on the sim.
7. **Sanity checks**
   - Try **k=0** (no latent) and confirm DBPR ~ 0.
   - Try **stronger Z signal** (increase γ_1, γ_2) and verify DBPR rises *when the sign is fixed*.
   - Sensitivity to class balance of D and to dispersion α.

TL;DR

- **Architecture**: tiny linear predictors for D and Y (BCE & NB), plus a small MLP generator that infers $\hat{Z}$ from (X, D, Y).
- **Objective**: make $\hat{Z}$ useful by improving D and Y predictions over baselines that lack Z, measured via deviance-based partial R^2; also include the usual BCE/NB likelihoods.
- **Key issue**: the code currently **adds** the partial-R^2 terms to the minimized loss, which **punishes** improvement. Flip the sign (or give them a negative weight), normalize/weight the losses, stabilize the inner predictors, and tune LRs. Those changes should materially boost performance **without altering the architecture**.

I'm curious about the `loss.backward(retain_graph=True)` lines. Is this typical? I'm not sure what it is for, or what happens if this `retain_graph` is not used

Good eye — that's an unusual thing to see sprinkled in code, so let's unpack it carefully.

---

1. What `retain_graph=True` does

When you call `loss.backward()`, PyTorch:

- **Traverses the computation graph** that produced `loss`.
- **Computes gradients** for all leaf parameters with `requires_grad=True`.
- **By default** (`retain_graph=False`), **frees** the intermediate tensors (activations, saved variables) in the graph to reclaim memory.

If you pass `retain_graph=True`, PyTorch **keeps** the computational graph in memory after backprop. That way you can call `.backward()` again on the *same* graph.

---

2. Why it's used here

Looking at your notebook:

- The **generator**, **NB predictor**, and **binary classifier** are all trained *sequentially* inside the same outer loop.
- Some parts of the code compute losses for one module, call **`loss.backward(retain_graph=True)`**, then later compute losses for another module and call **`backward()`** again.

The idea is:

- They want gradients to flow through **the same forward pass outputs** into different parameter groups.
- Example:
  - First backprop through **`bce_loss`** to update the classifier.
  - Then backprop through **`nll_loss`** to update the NB predictor.
  - Finally backprop through some combination (including DBPR terms) to update the generator.

If you didn't use **`retain_graph=True`** in the first backward call, PyTorch would free the computation graph after the BCE loss, and the second **`.backward()`** (for NB or generator) would fail with an error like:

```pgsql
RuntimeError: Trying to backward through the graph a second time,
but the buffers have already been freed. Specify retain_graph=True if you need to backward
through the graph a second time
```

---

3. Is this "typical"?

Not really in clean code. Two standard alternatives:

- **Accumulate losses first**:

  ```python
  total_loss = bce_loss + nll_loss + dbpr_loss
  total_loss.backward()
  ```

  That way, one **`.backward()`** covers all terms and you don't need to retain the graph.

- **Multiple backward passes** only if the training loop is deliberately *splitting* optimization steps (e.g., update classifier separately from generator). In that case, **`retain_graph=True`** is sometimes used — but it's usually a sign that the training logic is a bit tangled.

---

4. What happens if you remove it here?

- If you replace all those lines with a single combined **`total_loss.backward()`**, things will likely still work (and be more memory-efficient).
- If you just delete **`retain_graph=True`** but keep multiple **`.backward()`** calls on the same graph, PyTorch will error after the first call because the graph is gone.

- If the intention is indeed to **train predictors separately**, then **`retain_graph=True`** is keeping the graph alive until the generator also gets its gradient. But this increases memory use.

✅ **Rule of thumb**:
Use **`retain_graph=True`** only if you **must** backpropagate multiple times through the *same forward pass*. If you can restructure the code to accumulate all loss terms into one scalar and call **`.backward()`** once per batch, that's preferred.

Do you want me to trace through the exact section of your notebook where **`retain_graph=True`** is called and tell you whether it's truly necessary there, or whether the losses could just be accumulated into one call?

Is this potentially related to the fact that this is a GAN architecture (i.e., there's an adversary, so that's why retain_graph=True?)

Short answer: no—being "GAN-like" doesn't inherently require **`retain_graph=True`**.

In typical adversarial training you **don't** backprop twice through the *same* forward pass. Instead you do two separate forwards:

1. **Discriminator step**

- Build fakes with the generator, but **detach** them so gradients don't flow into G:

```python
fake = G(z).detach()
d_loss = D_loss(D(real), D(fake))
d_optim.zero_grad()
d_loss.backward()
d_optim.step()
```

2. **Generator step**

- Run a **new** forward where fakes are *not* detached (so G gets gradients):

```python
fake = G(z)
g_loss = G_loss(D(fake))
g_optim.zero_grad()
g_loss.backward()
g_optim.step()
```

Because each step uses its **own forward pass**, there's no need to keep a previous graph alive; **`retain_graph=True`** just burns memory.

When *would* **`retain_graph=True`** show up around GANs?

- If you intentionally do **multiple `.backward()` calls on the exact same forward graph**, e.g. computing and backpropagating several losses separately rather than summing them first. (You can almost always just sum the losses and call **`.backward()` once**.)
- Rare cases with **higher-order gradients** (unrolled GANs, some regularizers). Even then you usually use **`create_graph=True`** in **`torch.autograd.grad`**, not **`retain_graph=True`** on repeated **`.backward()`**—and you still try to structure things to avoid retaining large graphs.

In your colleague's code, **`retain_graph=True`** is there because they backprop the BCE term, the NB term, and the "DBPR" term **separately** through the **same forward** that produced $\hat{\mathbf{z}}$. That's a design choice, not a necessity of adversarial setups. Two cleaner (and more memory-friendly) options that keep the architecture unchanged:

- **Accumulate then backward once**

  ```python
  total_loss = bce_loss + nb_loss + dbpr_loss   # with whatever weights
  optim.zero_grad()
  total_loss.backward()      # no retain_graph needed
  optim.step()
  ```

- **Separate steps = separate forwards**
  - Do a predictor step (update the classifier/NB model) using one forward, step their optimizers.
  - Re-forward to compute the generator loss (with predictors frozen or warm-started), then step the generator.
  - Again, no need for **`retain_graph=True`**.

Rule of thumb: if you see **`retain_graph=True`**, ask "am I calling **`.backward()`** more than once on the *same* tensors?" If yes, either (a) sum losses and backward once, or (b) do another forward for the next step. Both are usually simpler and more stable than retaining the graph.

> Out of curiosity, is it possible (during training) to "freeze" certain parts of the architecture so that only some weights in certain modules are updated?

Yes — that's a very common pattern in PyTorch training loops (and in GANs in particular). You can "freeze" modules or parameters so they don't get updated, while still using their outputs in the forward pass.

🔧 How to freeze parameters

1. **Turn off gradients for a module/parameter**

   ```python
   for param in model.submodule.parameters():
       param.requires_grad = False
   ```

- Those parameters won't accumulate gradients during `.backward()`.
- Optimizers will ignore them since they're excluded from `param_groups` with `requires_grad=False`.

2. **Selectively include parameters in the optimizer**

```
python

optimizer = torch.optim.Adam(
    filter(lambda p: p.requires_grad, model.parameters()), lr=1e-3
)
```

- Ensures only the "trainable" ones are updated.

📌 Example: Freezing the discriminator while training the generator

```
python

# Freeze discriminator
for p in D.parameters():
    p.requires_grad = False

g_optim.zero_grad()
g_loss = criterion(D(G(z)), real_labels)  # still uses D in forward
g_loss.backward()
g_optim.step()

# Unfreeze discriminator for next step
for p in D.parameters():
    p.requires_grad = True
```

Even though `D` is frozen, you can still compute `D(G(z))`. Gradients flow **through** D into G (so G updates), but D's weights don't change.

Why it matters

- **GAN training**: alternate between freezing G and D.
- **Transfer learning**: freeze early layers (e.g. pretrained CNN backbone), only train a new head.
- **Multi-objective setups**: freeze one predictor while training another (like in your code — you could freeze the predictors and just update the generator, or vice versa).

✅ **Summary**
Yes, you can freeze parts of the architecture:

- Set `requires_grad=False` on parameters you don't want updated.
- Make sure the optimizer only sees the "active" parameters.
- You can still use frozen modules in the forward pass, and gradients can flow *through* them to earlier trainable parts.

Do you want me to show you how this would look if you wanted to freeze the **predictors** in your colleague's notebook while only updating the **generator** (i.e., predictors act as fixed critics)?

Can you give me a bare-bones but workable architecture for the following single-cell RNA-seq task? This might be useful for me understanding how to recode the architecture I just showed you:

Suppose I have 2 single-cell RNA-seq datasets on the same set of genes. I want to design a paired VAE, where there are two VAEs (one for each dataset), and the goal is to integrate both datasets together through its low-dimensional space. That is, there is an adversarial classifier that is trying to predict from the posterior means which cell comes from which dataset. The adversary is trying to minimize its prediction loss. The encoders/decoders on the other hand, want to: 1) minimize the ELBO and 2) maximize the prediction loss.

Could you give me the python architecture for this?

Totally—here's a minimal but *working* "paired VAE + adversarial domain classifier" in PyTorch. Two VAEs (one per dataset) learn a shared latent space; a domain classifier tries to predict which dataset a sample came from using the **posterior means**; a gradient-reversal layer makes the encoders maximize the classifier's loss while you minimize the ELBO.

```python
# bare-bones paired VAE with adversarial domain classifier (PyTorch)
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset
from itertools import cycle

# ---------- Gradient Reversal ----------
class _GradReverseFn(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, lambd):
        ctx.lambd = float(lambd)
        return x.view_as(x)
    @staticmethod
    def backward(ctx, grad_output):
        return -ctx.lambd * grad_output, None

class GradReverse(nn.Module):
    def __init__(self, lambd=1.0):
        super().__init__()
        self.lambd = lambd
    def forward(self, x, lambd=None):
        return _GradReverseFn.apply(x, self.lambd if lambd is None else lambd)

# ---------- VAE components ----------
class Encoder(nn.Module):
    def __init__(self, input_dim, latent_dim, hidden=256):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, hidden), nn.ReLU(),
            nn.Linear(hidden, hidden), nn.ReLU()
        )
        self.mu = nn.Linear(hidden, latent_dim)
        self.logvar = nn.Linear(hidden, latent_dim)
    def forward(self, x):
        h = self.net(x)
        mu = self.mu(h)
        logvar = self.logvar(h)
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
```

```
        z = mu + eps * std
        return z, mu, logvar

class Decoder(nn.Module):
    def __init__(self, input_dim, latent_dim, hidden=256):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(latent_dim, hidden), nn.ReLU(),
            nn.Linear(hidden, hidden), nn.ReLU(),
            nn.Linear(hidden, input_dim)
        )
    def forward(self, z):
        return self.net(z)

class VAE(nn.Module):
    def __init__(self, input_dim, latent_dim, hidden=256):
        super().__init__()
        self.enc = Encoder(input_dim, latent_dim, hidden)
        self.dec = Decoder(input_dim, latent_dim, hidden)
    def forward(self, x):
        z, mu, logvar = self.enc(x)
        x_hat = self.dec(z)
        return x_hat, z, mu, logvar

# ---------- Domain classifier ----------
class DomainDiscriminator(nn.Module):
    def __init__(self, latent_dim, hidden=128, num_domains=2):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(latent_dim, hidden), nn.ReLU(),
            nn.Linear(hidden, num_domains)
        )
    def forward(self, z):
        return self.net(z)

# ---------- Loss helpers ----------
def mse_recon(x_hat, x):
    # For log1p-CPM or other normalized inputs; for counts swap to NB/Poisson.
    return F.mse_loss(x_hat, x, reduction="mean")

def kl_normal(mu, logvar):
    # mean over batch
    return torch.mean(-0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp(), dim=1))

# (Optional) NB likelihood if you use raw counts:
# def nb_nll(x, mu, theta, eps=1e-8):
#     # x >= 0 counts, mu > 0 mean, theta > 0 inverse-dispersion (per-gene or scalar)
#     t1 = torch.lgamma(x + theta) - torch.lgamma(theta) - torch.lgamma(x + 1)
#     t2 = theta * torch.log(theta + eps) - theta * torch.log(mu + theta + eps)
#     t3 = x * (torch.log(mu + eps) - torch.log(mu + theta + eps))
#     nll = -(t1 + t2 + t3)
#     return nll.mean()

# ---------- Training skeleton ----------
def train_paired_vae(
    Xa, Xb,               # torch.FloatTensor (n_a x g), (n_b x g), same genes/order
    input_dim, latent_dim,
    batch_size=256, epochs=50, device="cuda" if torch.cuda.is_available() else "cpu",
    beta_kl=1.0,          # ELBO KL weight
    adv_weight=1.0,       # multiplies domain loss
    grl_lambda=1.0,       # gradient reversal strength
    lr_vae=1e-3, lr_adv=1e-3,
):
    vae_a = VAE(input_dim, latent_dim).to(device)
    vae_b = VAE(input_dim, latent_dim).to(device)
    disc  = DomainDiscriminator(latent_dim).to(device)
    grl   = GradReverse()

    opt_vae = torch.optim.Adam(
        list(vae_a.parameters()) + list(vae_b.parameters()),
        lr=lr_vae
    )
    opt_adv = torch.optim.Adam(disc.parameters(), lr=lr_adv)

    ds_a = DataLoader(TensorDataset(Xa), batch_size=batch_size, shuffle=True, drop_last=True)
```

```
    ds_b = DataLoader(TensorDataset(Xb), batch_size=batch_size, shuffle=True, drop_last=True)

    for epoch in range(1, epochs + 1):
        vae_a.train(); vae_b.train(); disc.train()
        iters = min(len(ds_a), len(ds_b))
        total_elbo = total_adv = 0.0

        for (xa_batch,), (xb_batch,) in zip(ds_a, ds_b):
            xa = xa_batch.to(device)
            xb = xb_batch.to(device)

            # ----- VAE forward for each domain -----
            xa_hat, za, mua, logvara = vae_a(xa)
            xb_hat, zb, mub, logvarb = vae_b(xb)

            # ELBO (sum over the two datasets)
            recon = mse_recon(xa_hat, xa) + mse_recon(xb_hat, xb)
            kld   = kl_normal(mua, logvara) + kl_normal(mub, logvarb)
            elbo_loss = recon + beta_kl * kld

            # ----- Adversarial domain loss on posterior MEANS -----
            # labels: 0 for A, 1 for B
            z_mu = torch.cat([mua, mub], dim=0)
            y_dom = torch.cat([
                torch.zeros(mua.size(0), dtype=torch.long),
                torch.ones(mub.size(0), dtype=torch.long)
            ], dim=0).to(device)

            # gradient reversal so encoders maximize CE while discriminator minimizes it
            z_rev = grl(z_mu, lambd=grl_lambda)
            logits = disc(z_rev)
            adv_loss = F.cross_entropy(logits, y_dom)

            # ----- Joint update -----
            # Using two optimizers lets you use different LRs; GRL handles the sign flip to
encoders.
            opt_vae.zero_grad()
            opt_adv.zero_grad()
            total_loss = elbo_loss + adv_weight * adv_loss
            total_loss.backward()
            opt_vae.step()
            opt_adv.step()

            total_elbo += elbo_loss.item()
            total_adv  += adv_loss.item()

        print(f"Epoch {epoch:03d} | ELBO: {total_elbo/iters:.4f} | ADV: {total_adv/iters:.4f}")

    return vae_a, vae_b, disc

# ---------- Example usage ----------
if __name__ == "__main__":
    # Dummy shapes: two datasets with the same genes (g)
    g = 2000
    n_a, n_b = 5000, 6000
    torch.manual_seed(0)

    # Assume inputs are log1p-normalized; replace with your real tensors
    Xa = torch.randn(n_a, g)
    Xb = torch.randn(n_b, g)

    vae_a, vae_b, disc = train_paired_vae(
        Xa, Xb, input_dim=g, latent_dim=32,
        batch_size=256, epochs=10,
        beta_kl=1.0, adv_weight=1.0, grl_lambda=1.0,
        lr_vae=1e-3, lr_adv=1e-3
    )

    # Getting latent means for downstream use:
    vae_a.eval(); vae_b.eval()
    with torch.no_grad():
        _, _, mu_a, _ = vae_a(Xa)
        _, _, mu_b, _ = vae_b(Xb)
```

```
    Z_a = mu_a.cpu().numpy()
    Z_b = mu_b.cpu().numpy()
```

Notes you can tweak without changing the core idea:

- **Reconstruction**: for raw counts, switch **`mse_recon`** to a Negative Binomial NLL (like scVI). Keep decoder output positive (e.g., softplus) and add a dispersion parameter.
- **Balancing**: if dataset sizes differ a lot, sample batches with **`cycle()`** so both see equal steps, or weight the domain loss to avoid class imbalance.
- **β-VAE**: tune **`beta_kl`** for tighter/looser posteriors.
- **Adversary strength**: **`adv_weight`** and **`grl_lambda`** control how hard you push for alignment; ramp them up over epochs if training is unstable.

This is super interesting.

Is there a way to write this without the gradient reversal (since my intended architecture might not be able to use a similar strategy) but instead to freeze certain weights? Also, can you explain to me the loss of the VAE and the loss of the discriminator for such an architecture?

Yep—you can do this cleanly **without** a gradient-reversal layer by **freezing** the discriminator when you want the encoders to "fool" it, and freezing the encoders when you want the discriminator to get better. Below I'll (1) show a minimal training loop that alternates those two phases, and (2) spell out the losses precisely for both the VAE(s) and the discriminator.

A. Bare-bones training without gradient reversal (with freezing)

Two VAEs (A and B) share the same latent prior; a discriminator tries to tell their **posterior means** apart. We alternate:

- **D-step** (train discriminator): freeze encoders/decoders; update D to **minimize** classification loss on μ.
- **E-step** (train encoders/decoders): freeze D; update encoders/decoders to (i) **minimize ELBO** and (ii) **maximize** the discriminator's loss (i.e., make domains indistinguishable).

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset

# ----------------- VAE blocks -----------------
class Encoder(nn.Module):
    def __init__(self, input_dim, latent_dim, hidden=256):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, hidden), nn.ReLU(),
            nn.Linear(hidden, hidden), nn.ReLU()
```

```
        )
        self.mu = nn.Linear(hidden, latent_dim)
        self.logvar = nn.Linear(hidden, latent_dim)
    def forward(self, x):
        h = self.net(x)
        mu = self.mu(h)
        logvar = self.logvar(h)
        std = torch.exp(0.5 * logvar)
        z = mu + std * torch.randn_like(std)  # reparameterization
        return z, mu, logvar

class Decoder(nn.Module):
    def __init__(self, input_dim, latent_dim, hidden=256):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(latent_dim, hidden), nn.ReLU(),
            nn.Linear(hidden, hidden), nn.ReLU(),
            nn.Linear(hidden, input_dim)
        )
    def forward(self, z):
        return self.net(z)  # for log1p normalized inputs; use softplus + NB for counts

class VAE(nn.Module):
    def __init__(self, input_dim, latent_dim, hidden=256):
        super().__init__()
        self.enc = Encoder(input_dim, latent_dim, hidden)
        self.dec = Decoder(input_dim, latent_dim, hidden)
    def forward(self, x):
        z, mu, logvar = self.enc(x)
        x_hat = self.dec(z)
        return x_hat, z, mu, logvar

class DomainDiscriminator(nn.Module):
    def __init__(self, latent_dim, hidden=128, num_domains=2):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(latent_dim, hidden), nn.ReLU(),
            nn.Linear(hidden, num_domains)
        )
    def forward(self, z_mu):
        return self.net(z_mu)

# -------------- Loss helpers --------------
def recon_mse(x_hat, x):
    return F.mse_loss(x_hat, x, reduction="mean")

def kl_normal(mu, logvar):
    # mean over batch
    return torch.mean(-0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp(), dim=1))

# -------------- Training loop --------------
def train_paired_no_grl(
    Xa, Xb, input_dim, latent_dim,
    batch_size=256, epochs=50, device="cuda" if torch.cuda.is_available() else "cpu",
    beta_kl=1.0, adv_weight=1.0,
    lr_vae=1e-3, lr_disc=1e-3, d_steps=1, e_steps=1
):
    vae_a = VAE(input_dim, latent_dim).to(device)
    vae_b = VAE(input_dim, latent_dim).to(device)
    disc  = DomainDiscriminator(latent_dim).to(device)

    opt_vae  = torch.optim.Adam(list(vae_a.parameters()) + list(vae_b.parameters()), lr=lr_vae)
    opt_disc = torch.optim.Adam(disc.parameters(), lr=lr_disc)

    loader_a = DataLoader(TensorDataset(Xa), batch_size=batch_size, shuffle=True,
drop_last=True)
    loader_b = DataLoader(TensorDataset(Xb), batch_size=batch_size, shuffle=True,
drop_last=True)
    iters = min(len(loader_a), len(loader_b))

    for epoch in range(1, epochs + 1):
        vae_a.train(); vae_b.train(); disc.train()
        tot_elbo = tot_d = tot_adv_enc = 0.0

        for (xa_batch,), (xb_batch,) in zip(loader_a, loader_b):
```

```
            xa = xa_batch.to(device); xb = xb_batch.to(device)

            # -----------------------
            # D-step(s): train discriminator, freeze encoders/decoders
            # -----------------------
            for p in vae_a.parameters(): p.requires_grad = False
            for p in vae_b.parameters(): p.requires_grad = False
            for p in disc.parameters():  p.requires_grad = True

            for _ in range(d_steps):
                with torch.no_grad():
                    _, _, mu_a, _ = vae_a(xa)
                    _, _, mu_b, _ = vae_b(xb)
                logits = disc(torch.cat([mu_a, mu_b], dim=0))
                y = torch.cat([
                    torch.zeros(mu_a.size(0), dtype=torch.long),
                    torch.ones (mu_b.size(0), dtype=torch.long)
                ]).to(device)
                d_loss = F.cross_entropy(logits, y)

                opt_disc.zero_grad()
                d_loss.backward()
                opt_disc.step()

            # -----------------------
            # E-step(s): train encoders/decoders (VAEs), freeze discriminator
            # -----------------------
            for p in vae_a.parameters(): p.requires_grad = True
            for p in vae_b.parameters(): p.requires_grad = True
            for p in disc.parameters():  p.requires_grad = False  # frozen critic

            for _ in range(e_steps):
                xa_hat, _, mu_a, logvar_a = vae_a(xa)
                xb_hat, _, mu_b, logvar_b = vae_b(xb)

                # ELBO parts
                recon = recon_mse(xa_hat, xa) + recon_mse(xb_hat, xb)
                kld   = kl_normal(mu_a, logvar_a) + kl_normal(mu_b, logvar_b)
                elbo  = recon + beta_kl * kld

                # Adversarial part for encoders: maximize D's loss while D is frozen.
                logits_frozen = disc(torch.cat([mu_a, mu_b], dim=0))  # gradients flow to
encoders only
                y = torch.cat([
                    torch.zeros(mu_a.size(0), dtype=torch.long),
                    torch.ones (mu_b.size(0), dtype=torch.long)
                ]).to(device)

                # Option A: maximize CE by minimizing its negative
                adv_enc = -F.cross_entropy(logits_frozen, y)

                # (Equivalent Option B: push to uniform distribution)
                # probs = F.log_softmax(logits_frozen, dim=1)
                # uniform = torch.full_like(probs, 1.0 / probs.size(1)).detach()
                # adv_enc = F.kl_div(probs, uniform, reduction="batchmean") * (-1.0)

                total_loss = elbo + adv_weight * adv_enc  # adv_enc is negative CE

                opt_vae.zero_grad()
                total_loss.backward()
                opt_vae.step()

            tot_elbo     += elbo.item()
            tot_d        += d_loss.item()
            tot_adv_enc  += (-adv_enc).item()  # report as positive CE magnitude

        print(f"Epoch {epoch:03d} | ELBO: {tot_elbo/iters:.4f} | D CE: {tot_d/iters:.4f} | Enc-
Adv CE: {tot_adv_enc/iters:.4f}")

    return vae_a, vae_b, disc
```

Key mechanics:

- **D-step**: we compute μ with `torch.no_grad()` (or by setting encoders `requires_grad=False`) so gradients don't flow into encoders; only D updates.
- **E-step**: we **freeze D** (`requires_grad=False`). We still forward μ through D; this routes gradients **through D's fixed computations into the encoders**, and we **maximize** the discriminator's loss by minimizing its negative. That's exactly what GRL does, just made explicit via freezing + sign flip.

B. Losses spelled out

1) VAE loss (per domain; sum over domains A & B)

Let the encoder define $q_\phi(z \mid x) = \mathcal{N}(\mu_\phi(x), \mathrm{diag}(\sigma^2_\phi(x)))$ and the decoder define $p_\theta(x \mid z)$.

ELBO to minimize (negative ELBO):

$$\mathcal{L}_{\mathrm{VAE}}(x;\theta,\phi) = \underbrace{\mathbb{E}_{q_\phi(z|x)}\big[-\log p_\theta(x \mid z)\big]}_{\text{reconstruction loss}} + \beta \cdot \underbrace{D_{\mathrm{KL}}(q_\phi(z \mid x) \,\|\, \mathcal{N}(0, I))}_{\text{regularization}}.$$

- If you feed **log1p-normalized** expressions, a simple **MSE** on decoder output works as the reconstruction term.
- If you model **raw counts**, use **NB** or **Poisson** negative log-likelihood for $-\log p_\theta(x \mid z)$; make the decoder output the mean (via `softplus`) and (optionally) learn per-gene dispersion.

For two datasets:

$$\mathcal{L}_{\text{ELBO-total}} = \mathcal{L}^{(A)}_{\mathrm{VAE}} + \mathcal{L}^{(B)}_{\mathrm{VAE}}.$$

2) Discriminator loss (domain classification)

Let $f_\psi(\mu)$ output logits over domains. With labels $y \in \{0, 1\}$:

Discriminator step (train D, encoders frozen):

$$\mathcal{L}_D = \mathrm{CE}\big(f_\psi(\mu^{(A)}), 0\big) + \mathrm{CE}\big(f_\psi(\mu^{(B)}), 1\big),$$

minimize w.r.t. ψ.

Encoder step (train encoders, D frozen): make domains indistinguishable using one of:

- **Maximize discriminator CE** (label-preserving):

  $$\mathcal{L}_{\text{adv-enc}} = -\mathcal{L}_D.$$

  (What the code above does via `-F.cross_entropy(...)`.)

- **Push to uniform (entropy maximization):**

  $$\mathcal{L}_{\text{adv-enc}} = -\mathbb{E}\Big[\sum_k \tfrac{1}{K} \log p_\psi(k \mid \mu)\Big] = -H_{\text{uniform}} + \mathrm{KL}\big(\tfrac{1}{K} \,\|\, p_\psi(\cdot \mid \mu)\big)\,,$$

  i.e., minimize the negative cross-entropy to a uniform target (equivalently, maximize the entropy of D's predictions).

Total encoder/decoder loss (during E-step):

$$\mathcal{L}_{\text{enc/dec}} = \mathcal{L}_{\text{ELBO-total}} + \lambda_{\text{adv}} \cdot \mathcal{L}_{\text{adv-enc}},$$

minimize w.r.t. (θ, ϕ). Note $\mathcal{L}_{\text{adv-enc}}$ is **negative** CE in the label-preserving version.

Practical knobs (no architecture change)

- β in the ELBO to balance compression vs. reconstruction (β-VAE).
- λ_{adv} to control alignment strength; consider a **warm-up** schedule (start small, ramp up).
- **Update ratio**: more D-steps early (e.g., `d_steps=5`) can stabilize the signal for the encoders.
- **Imbalance**: if $n_A \neq n_B$, balance batches or weight the CE accordingly.

This setup gives you the exact same adversarial pressure as GRL, but implemented with **freezing + sign-flip**, which fits scenarios where gradient reversal isn't available or desired.

> Can you explain to me what exactly gradient reversal is (and if, from a training perspective) it's more or less desirable compared to freezing?

Great question 👍 — these two strategies (gradient reversal vs. freezing + alternating updates) are functionally related but not the same, and understanding the distinction makes the training dynamics much clearer.

---

1. What Gradient Reversal actually does

Formally, the **Gradient Reversal Layer (GRL)** is an identity operation in the forward pass, but in the backward pass it multiplies the incoming gradient by $-\lambda$.

- **Forward:**

  $$\text{GRL}_\lambda(x) = x$$

  (the discriminator sees exactly the latent representation as-is).

- **Backward:**
  If $\frac{\partial L}{\partial x}$ is the gradient flowing back from the discriminator, the GRL passes back:

  $$\frac{\partial L}{\partial x}_{\text{after GRL}} = -\lambda \cdot \frac{\partial L}{\partial x}.$$

So to the discriminator, everything looks normal: it tries to minimize cross-entropy.
But to the encoder, gradients are flipped in sign → it's as if the encoder is **maximizing** the discriminator's loss.

That's why people say: *GRL is just a "trick" that lets you implement adversarial training with a single loss function and a single optimizer step.*

2. What Freezing + Alternating Updates does

Instead of flipping gradients in-place, you train discriminator and encoders in separate phases:

- **D-step**: freeze encoder, update D to minimize CE.
- **E-step**: freeze D, update encoder to *explicitly maximize* CE (or equivalently minimize $-$CE).

This mimics what GRL does, but:

- Requires two separate optimizers / training phases.
- Requires careful scheduling (how many D-steps vs. E-steps).
- Often looks more like GAN training (where D and G are trained alternately).

3. Comparison from a training perspective

Aspect	Gradient Reversal	Freezing + Alternating
Implementation	Simple: one forward, one backward; loss includes both ELBO and domain CE.	More bookkeeping: two optimizers, two phases.
Stability	Often more stable (like domain-adversarial training, DANN) because updates are simultaneous.	Can be less stable if D overpowers encoders or vice versa (like GAN instability).
Control	Adversarial weight λ is explicit (scale in GRL).	Control comes from update ratio (#D vs. #E steps) and sign flip.
Interpretability	Easy: one loss function, standard autograd.	Matches GAN intuition: separate agents fighting.
When useful	Most domain adaptation / invariant representation setups.	When you want explicit GAN-style control (e.g. multiple discriminator updates per encoder update).

4. Which is more desirable?

- **If you can use GRL**:
  - It's **simpler**, uses less code, often trains smoother (because encoder and discriminator update in the same backward pass).
  - Very common in **domain adaptation VAEs** and **DANN (Domain-Adversarial Neural Networks)**.
- **If you can't use GRL** (e.g., your framework doesn't support it or you want explicit GAN-like updates):
  - Freezing + alternating works fine.
  - You have more knobs to tune (update ratios, when to freeze, etc.), but it can be trickier to balance.

✅ **Summary:**
Gradient reversal is basically an *automatic adversarial training hack*: it flips gradients during backprop, so you don't need to alternate steps. It's often **more stable and simpler** for domain-invariant representation learning.
Freezing + alternating is the **more general GAN-like approach**, giving you finer control but also more hyperparameters to juggle.

Would you like me to sketch a little diagram (in ASCII or math) showing how the gradient flow differs between GRL vs freezing? It might make the distinction crystal clear.

> This is what I have so far. I will send the .ipynb script of what I have so far in the next message.
>
> Based on my simulated data, I would like to train the `FullArchitecture` in an alternating fashion. (Technically, this doesn't make sense, but I want to try this as a proof-of-concept to later change the objective to what I'm truly interested later on.)
>
> Here is what I'm hoping you can help me with:
> 1) Based on the simulated data, train in minibatches using DataLoader (size 32)
> 2) Using the freezing+alternating strategy: After initialization, every "odd" iteration trains `BinaryClassifier` and `NBModel` separately (minimzing the `bce_loss` and `nb_loss` separately) until convergence. Then, every "even" iteration trains the full `FullArchitecture` model (which will update `ZGenerator` weights) based on minimizing the sum of `bce_loss` and `nb_loss` losses.
>
> The end goal is have the method make an educated guess on what "Z" is (the unmeasured confounders), even though the model only sees X (the covariates), D (the treatment), and Y (the outcome, which can be thought of as gene expression).
>
> If you think there are errors in my code already, please also let me know

Awesome — here's a clean, *working* skeleton that does exactly what you asked:

- mini-batches via `DataLoader(batch_size=32)`;
- **odd** outer iters: freeze the generator and train **BinaryClassifier** and **NBModel separately** (each with its own optimizer/loss);
- **even** outer iters: freeze the predictors, unfreeze the **ZGenerator**, and train the **FullArchitecture** to minimize **(bce_loss + nb_loss)** so gradients flow into `ZGenerator` through the frozen predictors.

I also included a solid NB loss (mean/dispersion parameterization), and comments at all the "gotcha" points (types, shapes, freezing, detach, etc.). You can paste this into a fresh cell and wire it to your simulated data tensors `X, D, Y`.

```python
# ============================================================
# Alternating training (freeze+unfreeze) for Z-estimation PoC
# ============================================================
import torch
```

```python
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader

device = "cuda" if torch.cuda.is_available() else "cpu"
torch.manual_seed(0)

# -------------------------
# Dataset / DataLoader
# -------------------------
class XDYDataset(Dataset):
    """Holds X (covariates, float), D (binary 0/1), Y (counts or normalized)."""
    def __init__(self, X, D, Y):
        # Expect torch tensors already; cast if needed
        self.X = X.float()
        self.D = D.float().view(-1, 1)    # BCEWithLogits target expects float
        self.Y = Y.float()                # counts can be float tensor
        assert len(self.X) == len(self.D) == len(self.Y)
    def __len__(self): return len(self.X)
    def __getitem__(self, i): return self.X[i], self.D[i], self.Y[i]

# Example hookup (replace with your simulated tensors)
# X: (n, d), D: (n,), Y: (n, p)
# X, D, Y = your_simulation()
# ds = XDYDataset(X, D, Y)
# loader = DataLoader(ds, batch_size=32, shuffle=True, drop_last=True, pin_memory=True)

# -------------------------
# Models
# -------------------------
class BinaryClassifier(nn.Module):
    """Logit(D | features). Keep simple & linear unless you intend otherwise."""
    def __init__(self, in_dim):
        super().__init__()
        self.linear = nn.Linear(in_dim, 1)
    def forward(self, feats):  # feats: (B, in_dim)
        return self.linear(feats)  # logits (B,1)

class NBModel(nn.Module):
    """
    Negative Binomial regression for Y given features.
    Outputs log-mean per gene. Keeps a learnable inverse-dispersion theta >= 0.
    """
    def __init__(self, in_dim, n_genes, bias=True, init_theta=1.0):
        super().__init__()
        self.fc = nn.Linear(in_dim, n_genes, bias=bias)  # outputs log_mu
        # inverse-dispersion (a.k.a. theta) per gene; softplus to keep positive
        self.theta_log = nn.Parameter(torch.full((n_genes,), float(init_theta)).log())
    def forward(self, feats):
        log_mu = self.fc(feats)                     # (B, p)
        theta = F.softplus(self.theta_log) + 1e-8  # (p,)
        return log_mu, theta

def nb_nll_from_logmu(y, log_mu, theta, eps=1e-8):
    """
    NB NLL with mean mu=exp(log_mu) and inverse-dispersion theta (>0), broadcast over batch.
    y: (B,p), log_mu: (B,p), theta: (p,) or (1,)
    """
    mu = torch.exp(log_mu) + eps
    theta = theta.view(1, -1)  # (1,p)
    # log-likelihood components
    t1 = torch.lgamma(y + theta) - torch.lgamma(theta) - torch.lgamma(y + 1.0)
    t2 = theta * (torch.log(theta + eps) - torch.log(theta + mu))
    t3 = y * (torch.log(mu) - torch.log(theta + mu))
    nll = -(t1 + t2 + t3)      # (B,p)
    return nll.mean()          # scalar

class ZGenerator(nn.Module):
    """
    Infers Z from (X, D, Y). You can change inputs later (e.g., X only) without changing the
loop.
    Returns a *binarized* Z via straight-through estimator so grads flow to probs.
    """
    def __init__(self, x_dim, d_dim, y_dim, z_dim=1, hidden=128):
        super().__init__()
```

```
        self.net = nn.Sequential(
            nn.Linear(x_dim + d_dim + y_dim, hidden), nn.ReLU(),
            nn.Linear(hidden, hidden), nn.ReLU(),
            nn.Linear(hidden, z_dim), nn.Sigmoid()
        )
    def forward(self, X, D, Y):
        probs = self.net(torch.cat([X, D, Y], dim=1))
        # straight-through binarization (optional; you can keep it continuous, too)
        z_hard = (probs > 0.5).float()
        Z_hat = z_hard + (probs - probs.detach())  # forward = hard; backward = soft
        return Z_hat, probs

class FullArchitecture(nn.Module):
    """
    Wires ZGenerator -> predictors.
    - BinaryClassifier sees [X, Z_hat]
    - NBModel sees [X, D, Z_hat]
    """
    def __init__(self, x_dim, d_dim, y_dim, z_dim, n_genes):
        super().__init__()
        self.gen = ZGenerator(x_dim, d_dim, y_dim, z_dim=z_dim)
        self.cls = BinaryClassifier(in_dim=x_dim + z_dim)
        self.nb  = NBModel(in_dim=x_dim + d_dim + z_dim, n_genes=n_genes)

    def forward(self, X, D, Y):
        Z_hat, Z_prob = self.gen(X, D, Y)
        logits_D = self.cls(torch.cat([X, Z_hat], dim=1))              # (B,1)
        log_mu_Y, theta = self.nb(torch.cat([X, D, Z_hat], dim=1))     # (B,p), (p,)
        return Z_hat, Z_prob, logits_D, log_mu_Y, theta
```

Alternating training loop (odd = predictors, even = generator)

- **Odd iteration**
  Freeze **`ZGenerator`**. Train **BinaryClassifier** (BCE) to predict **`D`** from **`[X, Ẑ]`** and **NBModel** (NB NLL) to predict **`Y`** from **`[X, D, Ẑ]`**. Each has its **own** optimizer. I train them *separately* across the same mini-batches.
- **Even iteration**
  Freeze **both predictors**. Unfreeze **`ZGenerator`**. Compute **Ẑ**, feed it through the frozen predictors, compute **`bce_loss + nb_loss`**, and **update only the generator** so it learns Z that helps both tasks.

```python
# -------------------------
# Training config
# -------------------------
def alternating_train(
    X, D, Y,
    batch_size=32,
    outer_iters=10,             # number of odd/even outer iterations
    pred_epochs_per_odd=1,      # passes over data for each predictor (odd iters)
    gen_epochs_per_even=1,      # passes over data for generator (even iters)
    lr_cls=1e-3, lr_nb=1e-3, lr_gen=1e-3,
    z_dim=1
):
    n, x_dim = X.shape
    _, p     = Y.shape
    d_dim    = 1
    y_dim    = p

    ds = XDYDataset(X, D, Y)
    loader = DataLoader(ds, batch_size=batch_size, shuffle=True, drop_last=True,
pin_memory=True)

    model = FullArchitecture(x_dim, d_dim, y_dim, z_dim=z_dim, n_genes=p).to(device)
    bce = nn.BCEWithLogitsLoss()

    # separate optimizers
```

```python
    opt_cls = torch.optim.Adam(model.cls.parameters(), lr=lr_cls)
    opt_nb  = torch.optim.Adam(model.nb.parameters(),  lr=lr_nb)
    opt_gen = torch.optim.Adam(model.gen.parameters(), lr=lr_gen)

    for outer in range(1, outer_iters + 1):
        is_odd = (outer % 2 == 1)

        if is_odd:
            print(f"\n[Odd {outer}] Train predictors (cls & nb), generator frozen")
            # Freeze generator; train predictors *separately*
            for p in model.gen.parameters(): p.requires_grad = False
            for p in model.cls.parameters(): p.requires_grad = True
            for p in model.nb.parameters():  p.requires_grad = True

            # ---- BinaryClassifier: minimize BCE ----
            for epoch in range(1, pred_epochs_per_odd + 1):
                model.train()
                running = 0.0
                for Xb, Db, Yb in loader:
                    Xb, Db, Yb = Xb.to(device), Db.to(device), Yb.to(device)
                    with torch.no_grad():   # don't build graph through generator in odd phase
                        Z_hat, _ = model.gen(Xb, Db, Yb)
                    logits = model.cls(torch.cat([Xb, Z_hat], dim=1))
                    loss_bce = bce(logits, Db)

                    opt_cls.zero_grad(set_to_none=True)
                    loss_bce.backward()
                    opt_cls.step()
                    running += loss_bce.item()
                print(f"  BCE epoch {epoch}: {running/len(loader):.4f}")

            # ---- NBModel: minimize NB NLL ----
            for epoch in range(1, pred_epochs_per_odd + 1):
                model.train()
                running = 0.0
                for Xb, Db, Yb in loader:
                    Xb, Db, Yb = Xb.to(device), Db.to(device), Yb.to(device)
                    with torch.no_grad():
                        Z_hat, _ = model.gen(Xb, Db, Yb)
                    log_mu, theta = model.nb(torch.cat([Xb, Db, Z_hat], dim=1))
                    loss_nb = nb_nll_from_logmu(Yb, log_mu, theta)

                    opt_nb.zero_grad(set_to_none=True)
                    loss_nb.backward()
                    opt_nb.step()
                    running += loss_nb.item()
                print(f"  NB  epoch {epoch}: {running/len(loader):.4f}")

        else:
            print(f"\n[Even {outer}] Train generator, predictors frozen (min BCE + NB)")
            # Freeze predictors; unfreeze generator
            for p in model.gen.parameters(): p.requires_grad = True
            for p in model.cls.parameters(): p.requires_grad = False
            for p in model.nb.parameters():  p.requires_grad = False

            for epoch in range(1, gen_epochs_per_even + 1):
                model.train()
                running_total = 0.0
                for Xb, Db, Yb in loader:
                    Xb, Db, Yb = Xb.to(device), Db.to(device), Yb.to(device)

                    # Forward end-to-end; predictors are frozen so grads flow into generator
only.
                    Z_hat, Z_prob, logits, log_mu, theta = model(Xb, Db, Yb)
                    loss_bce = bce(logits, Db)
                    loss_nb  = nb_nll_from_logmu(Yb, log_mu, theta)
                    loss_gen = loss_bce + loss_nb

                    opt_gen.zero_grad(set_to_none=True)
                    loss_gen.backward()
                    opt_gen.step()

                    running_total += loss_gen.item()
                print(f"  GEN epoch {epoch}: {running_total/len(loader):.4f}")
```

```
    return model

# -------------------------
# Example call (uncomment and plug your tensors)
# -------------------------
# model = alternating_train(X.to(device), D.to(device), Y.to(device),
#                           batch_size=32, outer_iters=10,
#                           pred_epochs_per_odd=1, gen_epochs_per_even=1,
#                           lr_cls=1e-3, lr_nb=1e-3, lr_gen=1e-3, z_dim=1)
```

Why this matches your spec

- **Mini-batches:** `DataLoader(..., batch_size=32, shuffle=True, drop_last=True)`.
- **Odd iters:** generator is frozen; predictors are trained **separately**:
  - BCE on `logits = cls([X, Ẑ])` (minimize).
  - NB NLL on `log_mu = nb([X, D, Ẑ])` (minimize).
- **Even iters:** predictors are frozen; we minimize **sum** `(BCE + NB)` and update **only** the generator (`ZGenerator`) through the frozen nets.
- **No `retain_graph=True`:** each phase runs fresh forwards/backwards; no need to retain graphs.

Common pitfalls (quick checklist)

- **Targets dtype:** `BCEWithLogitsLoss` expects `Db.float()` with shape `(B,1)`; logits shape must match.
- **Freezing vs. detaching:** when training predictors, I used `with torch.no_grad()` around the generator to *avoid* building graphs you don't need. When training the generator, I froze predictors (`requires_grad=False`) but **did not** `detach` their inputs, so gradients flow **through** them into the generator.
- **NB parameterization:** this uses mean–theta (inverse-dispersion) parameterization. If your own NB loss expects `log_mu` exactly (like scVI's), keep it as shown; ensure your `NBModel` outputs `log_mu`.
- **Scale of losses:** NB NLL can dominate BCE. You might want weights, e.g. `loss_gen = lam_bce * BCE + lam_nb * NB`. Start with both 1.0, then tune.
- **"Convergence" for odd phase:** the skeleton uses `pred_epochs_per_odd`. If you truly want "until convergence," add a moving-average early-stopping on each loss with patience.

If you drop your `.ipynb` next, I can map these pieces 1:1 onto your classes (keeping your exact shapes and any special loss code you already have) and flag anything suspicious (e.g., shape/dtype, alpha/theta handling, unnecessary `retain_graph=True`, etc.).

Writeup4_redesign-attempt1.ipynb
File

This is what I have so far. Can you update your suggestions based on what I have so far (and correct anything that looks wrong -- let me know what you correct)

Got it—thanks for sharing your notebook. I read through `/mnt/data/Writeup4_redesign-attempt1.ipynb` and here's a concrete, fixed-up training skeleton that plugs into what you already have, plus a quick list of corrections I made.

What I corrected (important!)

1. **Bug in `FullArchitecture.forward`**
   You were returning `y_pred = self.classifier(...)` (i.e., using the *binary classifier* to predict Y).
   ✅ **Fix:** call the NB head:

   ```python
   log_mu = self.geneexp(x=x, d=d, z=z)
   return z, d_pred, log_mu
   ```

2. **Missing import for `math`** (used by `logit = lambda u: math.log(u/(1-u))`).
   ✅ Add `import math`.
3. **Shapes for BCE**
   `BCEWithLogitsLoss` needs `D` shaped `(B,1)` and `float`.
   ✅ Ensure the dataset returns `D.float().view(-1,1)`.
4. **Freezing logic**
   When training the predictors (odd iterations), you don't want gradients into the generator.
   ✅ Wrap generator forward in `with torch.no_grad()` during odd iterations, and set `requires_grad=False` on generator params.
   When training the generator (even iterations), predictors must be frozen (so gradients flow *through* them into the generator, but their weights don't update).
5. **No `retain_graph=True` needed**
   The loop below uses fresh forwards per step; you do not need to retain graphs.
6. **Naming consistency**
   Your `NBModel.forward(x,d,z)` returns `log_mu`. Your `nb_loss(log_mu, alpha, y)` expects `alpha` externally (from your simulation). The training code below wires that correctly.

> I left your linear heads with `bias=False` to respect your current architecture. (On real data, I'd usually turn biases on, but that's architecture, so I didn't change it.)

Drop-in code: dataset, fixed `FullArchitecture`, and alternating training

Paste these **in place** of your training section. This assumes you already defined:

- `BinaryClassifier` (linear on `[x,z]`, bias=False)
- `NBModel` (linear to `log_mu` on `[x,d,z]`, bias=False)
- `ZGenerator` (MLP on `[x,d,y]` returning probabilities in (0,1))
- `bce_loss(d_pred, d)` and `nb_loss(log_mu, alpha, y)` (as in your notebook)

- Tensors from your simulation: **X (n,d)**, **D (n,) or (n,1)**, **Y (n,p)**, and **alpha (1,p)**.

```python
# =========================
# 0) Imports and device
# =========================
import math
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader, TensorDataset

device = "cuda" if torch.cuda.is_available() else "cpu"

# =========================
# 1) Fix FullArchitecture
# =========================
class FullArchitecture(nn.Module):
    def __init__(self, x_dim, y_dim, z_dim):
        super().__init__()
        # your generator takes [x,d,y] → z
        self.generator = ZGenerator(input_dim=x_dim + y_dim + 1, z_dim=z_dim)
        # classifier on [x,z]
        self.classifier = BinaryClassifier(input_dim=x_dim + z_dim)
        # NB head on [x,d,z]
        self.geneexp   = NBModel(input_dim=x_dim + z_dim + 1, output_dim=y_dim)

    def forward(self, x, y, d):
        # z_hat
        z = self.generator(x=x, d=d, y=y)
        # logits for D
        d_pred = self.classifier(x=x, z=z)
        # log_mu for Y  (BUGFIX: use NB head, not classifier)
        log_mu = self.geneexp(x=x, d=d, z=z)
        return z, d_pred, log_mu

# =========================
# 2) Dataset / DataLoader
# =========================
class XDYDataset(Dataset):
    def __init__(self, X, D, Y):
        self.X = X.float()
        self.D = D.float().view(-1, 1)    # BCE requires (B,1) float
        self.Y = Y.float()
        assert len(self.X) == len(self.D) == len(self.Y)
    def __len__(self): return len(self.X)
    def __getitem__(self, i): return self.X[i], self.D[i], self.Y[i]

# Example:
# ds = XDYDataset(X, D, Y)
# loader = DataLoader(ds, batch_size=32, shuffle=True, drop_last=True, pin_memory=True)

# =========================
# 3) Alternating training
#    - Odd iters: train predictors ONLY (generator frozen)
#    - Even iters: train generator ONLY (predictors frozen)
# =========================
def alternating_train(
    X, D, Y, alpha,
    batch_size=32,
    outer_iters=10,
    pred_epochs_per_odd=1,   # you can increase if you really want “convergence”
    gen_epochs_per_even=1,
    lr_cls=1e-3, lr_nb=1e-3, lr_gen=1e-3,
    z_dim=1
):
    n, x_dim = X.shape
    _, p     = Y.shape
    y_dim    = p

    ds = XDYDataset(X, D, Y)
    loader = DataLoader(ds, batch_size=batch_size, shuffle=True, drop_last=True,
```

```
pin_memory=True)

    model = FullArchitecture(x_dim=x_dim, y_dim=y_dim, z_dim=z_dim).to(device)

    # separate optimizers
    opt_cls = torch.optim.Adam(model.classifier.parameters(), lr=lr_cls)
    opt_nb  = torch.optim.Adam(model.geneexp.parameters(),   lr=lr_nb)
    opt_gen = torch.optim.Adam(model.generator.parameters(), lr=lr_gen)

    alpha = alpha.to(device)  # (1,p) from your simulation

    for outer in range(1, outer_iters + 1):
        is_odd = (outer % 2 == 1)

        if is_odd:
            print(f"\n[Odd {outer}] Train predictors (BCE & NB), generator frozen")

            # freeze generator
            for p_ in model.generator.parameters(): p_.requires_grad = False
            for p_ in model.classifier.parameters(): p_.requires_grad = True
            for p_ in model.geneexp.parameters():   p_.requires_grad = True

            # ---- BinaryClassifier pass ----
            for epoch in range(1, pred_epochs_per_odd + 1):
                model.train()
                running = 0.0
                for Xb, Db, Yb in loader:
                    Xb, Db, Yb = Xb.to(device), Db.to(device), Yb.to(device)
                    with torch.no_grad():    # do not backprop through generator here
                        Zb = model.generator(x=Xb, d=Db, y=Yb)
                    logits = model.classifier(x=Xb, z=Zb)
                    loss_bce = bce_loss(logits, Db)

                    opt_cls.zero_grad(set_to_none=True)
                    loss_bce.backward()
                    opt_cls.step()
                    running += loss_bce.item()
                print(f"  BCE: {running/len(loader):.4f}")

            # ---- NBModel pass ----
            for epoch in range(1, pred_epochs_per_odd + 1):
                model.train()
                running = 0.0
                for Xb, Db, Yb in loader:
                    Xb, Db, Yb = Xb.to(device), Db.to(device), Yb.to(device)
                    with torch.no_grad():
                        Zb = model.generator(x=Xb, d=Db, y=Yb)
                    log_mu = model.geneexp(x=Xb, d=Db, z=Zb)
                    loss_nb = nb_loss(log_mu, alpha, Yb)

                    opt_nb.zero_grad(set_to_none=True)
                    loss_nb.backward()
                    opt_nb.step()
                    running += loss_nb.item()
                print(f"  NB : {running/len(loader):.4f}")

        else:
            print(f"\n[Even {outer}] Train generator (min BCE + NB), predictors frozen")

            # freeze predictors
            for p_ in model.generator.parameters(): p_.requires_grad = True
            for p_ in model.classifier.parameters(): p_.requires_grad = False
            for p_ in model.geneexp.parameters():   p_.requires_grad = False

            for epoch in range(1, gen_epochs_per_even + 1):
                model.train()
                running = 0.0
                for Xb, Db, Yb in loader:
                    Xb, Db, Yb = Xb.to(device), Db.to(device), Yb.to(device)
                    Zb, logits, log_mu = model(Xb, Yb, Db)   # uses fixed heads
                    loss_bce = bce_loss(logits, Db)
                    loss_nb  = nb_loss(log_mu, alpha, Yb)
                    loss     = loss_bce + loss_nb

                    opt_gen.zero_grad(set_to_none=True)
```

```
                    loss.backward()
                    opt_gen.step()
                    running += loss.item()
                print(f"  GEN: {running/len(loader):.4f}")

    return model

# ========================
# 4) Example call
# ========================
# model = alternating_train(
#     X.to(device), D.to(device), Y.to(device), alpha.to(device),
#     batch_size=32, outer_iters=10,
#     pred_epochs_per_odd=1, gen_epochs_per_even=1,
#     lr_cls=1e-3, lr_nb=1e-3, lr_gen=1e-3, z_dim=1
# )
```

A few quick sanity checks for your sim

- Make sure your simulated `D` is **float** 0/1 and **reshaped** to `(n,1)` before wrapping in the dataset.

  ```python
  D = D.float().view(-1,1)
  ```

- Your NB simulation block looks fine (Gamma–Poisson with mean `mu` if `rate = r/mu`). Keep `alpha` on the device.
- If you want to *binarize* `ZGenerator`'s output during generator training, you can do a straight-through step (optional):

  ```python
  probs = self.net(torch.cat([x,d,y], dim=1))
  z_hard = (probs > 0.5).float()
  z = z_hard + (probs - probs.detach())  # forward hard, backward soft
  ```

  For now, your continuous `z` is fine for a proof-of-concept.

Tuning tips (no architecture change)

- NB often dominates BCE; if needed, weight the losses:
  `loss = λ_bce * loss_bce + λ_nb * loss_nb` (start both at 1.0).
- If predictors seem underfit in odd iterations, bump `pred_epochs_per_odd` (e.g., 3–5).
  If generator chases noise, reduce `gen_epochs_per_even` or learning rate.
- Track simple metrics to validate behavior:
  - `torch.sigmoid(logits).mean()` vs. `D.mean()` for calibration.
  - NB NLL trend over outer iterations.
  - Correlation between true `Z` (you have it in sim) and `Z_hat` from the generator.

If you paste this in and hit run, it should train exactly in the odd/even pattern you described and fix the `y_pred` bug. If anything else in your notebook behaves oddly (e.g., shapes, device moves), drop that snippet and I'll adjust it 1:1.

This is great! After fitting the model, how do get the following ingredients:
1) I want to know what the estimated Z (given the X, D, Y I trained on) is,
2) I want to know what the predicted D's and Y's are

Totally doable. After training, you can run a batched "inference pass" to extract:

- $\hat{\mathbf{Z}}$ (your estimated latent confounder)
- $\hat{\mathbf{D}}$ logits and probabilities (sigmoid)
- $\hat{\mathbf{Y}}$ as the NB mean $\hat{\mu}$ (i.e., exp(log μ))

Below is a drop-in helper that works with the **`FullArchitecture`** you and I set up (generator → **`[X,D,Y]→Ẑ`**, classifier on **`[X,Ẑ]`**, NB head on **`[X,D,Ẑ]`**).

Batch inference utility

```python
import torch
from torch.utils.data import DataLoader

@torch.no_grad()
def predict_all(model, X, D, Y, batch_size=256, device=None):
    """
    Returns:
      Z_hat:    (n, z_dim)          estimated latent
      D_logit:  (n, 1)              logits for D
      D_prob:   (n, 1)              P(D=1 | X, Z_hat)
      log_mu:   (n, p)              NB log-mean for Y
      mu_hat:   (n, p)              NB mean (exp(log_mu))
      theta:    (p,) or None        inverse-dispersion if your NB head returns it
    """
    if device is None:
        device = next(model.parameters()).device

    model.eval()

    # use the same Dataset you trained with
    ds = XDYDataset(X, D, Y)
    loader = DataLoader(ds, batch_size=batch_size, shuffle=False, drop_last=False)

    Z_list, Dlogit_list, Dprob_list, logmu_list, mu_list = [], [], [], [], []
    theta_final = None

    for Xb, Db, Yb in loader:
        Xb, Db, Yb = Xb.to(device), Db.to(device), Yb.to(device)

        # 1) infer Z_hat from (X,D,Y)
        Zb = model.generator(x=Xb, d=Db, y=Yb)     # if your generator returns (Z_hat, probs), 
do: Zb, _ = ...
        # 2) predict D from (X, Z_hat)
        Dlogit_b = model.classifier(x=Xb, z=Zb)    # (B,1)
        Dprob_b  = torch.sigmoid(Dlogit_b)         # (B,1)
        # 3) predict Y from (X, D, Z_hat)
        out = model.geneexp(x=Xb, d=Db, z=Zb)

        # Support both variants of NB head:
        if isinstance(out, tuple) and len(out) == 2:
```

```
            logmu_b, theta_b = out                  # (B,p), (p,)
            theta_final = theta_b.detach().cpu()    # same per-batch; keep last
        else:
            logmu_b = out                           # (B,p)

        mu_b = torch.exp(logmu_b)

        Z_list.append(Zb.detach().cpu())
        Dlogit_list.append(Dlogit_b.detach().cpu())
        Dprob_list.append(Dprob_b.detach().cpu())
        logmu_list.append(logmu_b.detach().cpu())
        mu_list.append(mu_b.detach().cpu())

    Z_hat   = torch.cat(Z_list,    dim=0)
    D_logit = torch.cat(Dlogit_list, dim=0)
    D_prob  = torch.cat(Dprob_list,  dim=0)
    log_mu  = torch.cat(logmu_list,  dim=0)
    mu_hat  = torch.cat(mu_list,     dim=0)

    return {
        "Z_hat":   Z_hat,           # (n, z_dim)
        "D_logit": D_logit,         # (n, 1)
        "D_prob":  D_prob,          # (n, 1)
        "log_mu":  log_mu,          # (n, p)
        "mu_hat":  mu_hat,          # (n, p)
        "theta":   theta_final      # (p,) or None
    }
```

Example usage

```
python

# after training:
out = predict_all(model, X, D, Y, batch_size=256)

Z_hat  = out["Z_hat"].numpy()     # estimated latent(s)
D_prob = out["D_prob"].numpy()    # P(D=1)
Y_hat  = out["mu_hat"].numpy()    # NB means (predicted expression)

# if you want hard D labels:
D_pred = (out["D_prob"] > 0.5).float()
```

Notes / small gotchas

- If your **`ZGenerator`** returns both a binarized **`Z_hat`** and “soft” probabilities, change:

  ```
  python

  Zb = model.generator(x=Xb, d=Db, y=Yb)
  ```

  to:

  ```
  python

  Zb, Zprob_b = model.generator(x=Xb, d=Db, y=Yb)
  ```

  and stash **`Zprob_b`** too if you want it.
- If your NB head returns **`(log_mu, theta)`**, the helper already records **`theta`** once (it’s shared across batches). If it returns only **`log_mu`**, **`theta`** will be **`None`**.
- Shapes:

- `Z_hat`: `(n, z_dim)` (often `z_dim=1` in your PoC).
- `D_prob`: `(n, 1)`; logits match that shape.
- `mu_hat`: `(n, p)` where `p` is #genes; this is the **predicted mean** for each gene per cell.

- If you split train/val during fitting, pass the corresponding tensors to `predict_all` for each split.

If you want, I can add a quick evaluation cell (BCE/NB NLL on the predictions, AUROC for D, and a sanity plot comparing `Z_hat` to your true simulated `Z`).

Since I want to output the Z for the specific sample's X, D, and Y, is there a reason I need to create a new DataLoader? (Is this mainly just to interface correctly with `model`, and why you turned `shuffle` to False?)

Short answer: no, you don't *have* to use a `DataLoader` for inference. I used it as a convenience for batching and—when `shuffle=False`—to guarantee the outputs stay in the same order as your inputs. If you're extracting Z (and D/Y predictions) for specific samples, you can pass those tensors directly through the model.

Here are three compact patterns—pick what fits:

---

1) Single sample (exact X, D, Y → Ẑ, D̂, Ŷ)

```python
model.eval()
i = 123  # your sample index

x = X[i:i+1].to(model.generator.fc[0].weight.device)  # keep batch dim
d = D[i:i+1].view(-1, 1).to(x)
y = Y[i:i+1].to(x)

with torch.no_grad():
    # If your generator returns (Z_hat, probs), unpack both
    out = model.generator(x=x, d=d, y=y)
    Z_hat = out[0] if isinstance(out, tuple) else out          # (1, z_dim)

    d_logit = model.classifier(x=x, z=Z_hat)                   # (1, 1)
    d_prob  = torch.sigmoid(d_logit)

    nb_out  = model.geneexp(x=x, d=d, z=Z_hat)                 # (1,p) or ( (1,p), (p,) )
    if isinstance(nb_out, tuple):
        log_mu, theta = nb_out
    else:
        log_mu, theta = nb_out, None
    y_mean = torch.exp(log_mu)                                 # (1, p)
```

---

2) A specific subset by indices (preserves order of `idx`)

```python
idx = torch.tensor([0, 5, 42, 123, 999], dtype=torch.long)
device = next(model.parameters()).device
```

```
x = X[idx].to(device)
d = D[idx].view(-1,1).to(device)
y = Y[idx].to(device)

model.eval()
with torch.no_grad():
    out    = model.generator(x=x, d=d, y=y)
    Z_hat  = out[0] if isinstance(out, tuple) else out          # (k, z_dim)
    d_prob = torch.sigmoid(model.classifier(x=x, z=Z_hat))      # (k, 1)

    nb_out = model.geneexp(x=x, d=d, z=Z_hat)
    log_mu = nb_out[0] if isinstance(nb_out, tuple) else nb_out # (k, p)
    y_mean = torch.exp(log_mu)                                   # (k, p)
```

3) Whole dataset without a `DataLoader` (manual batching, no shuffle)

This avoids the `DataLoader` but still keeps memory safe on big data.

```python
device = next(model.parameters()).device
B = 512
Z_list, Dp_list, Ym_list = [], [], []

model.eval()
with torch.no_grad():
    for s in range(0, len(X), B):
        e = min(s + B, len(X))
        x = X[s:e].to(device)
        d = D[s:e].view(-1,1).to(device)
        y = Y[s:e].to(device)

        out    = model.generator(x=x, d=d, y=y)
        Z_hat  = out[0] if isinstance(out, tuple) else out
        d_prob = torch.sigmoid(model.classifier(x=x, z=Z_hat))
        nb_out = model.geneexp(x=x, d=d, z=Z_hat)
        log_mu = nb_out[0] if isinstance(nb_out, tuple) else nb_out
        y_mean = torch.exp(log_mu)

        Z_list.append(Z_hat.cpu()); Dp_list.append(d_prob.cpu()); Ym_list.append(y_mean.cpu())

Z_hat_all = torch.cat(Z_list, dim=0)   # (n, z_dim), aligned with original row order
D_prob_all= torch.cat(Dp_list, dim=0)  # (n, 1)
Y_mean_all= torch.cat(Ym_list, dim=0)  # (n, p)
```

Why I used `DataLoader(shuffle=False)` earlier

- **Order preservation:** You often want the outputs (Ẑ, D̂, Ŷ) to align with your original sample order or a DataFrame index. `shuffle=False` guarantees that.
- **Memory:** For large `n × p`, mini-batching prevents OOM.
- **Convenience:** Handles pinning to host memory and collation automatically.

If you're just querying a few specific samples or your data fits comfortably in memory, direct tensor calls (as above) are perfectly fine.

One more question. Why is `@torch.no_grad()` called outside the function? If I define the `predict_all` function, should I just call `@torch.no_grad()` before I use the function? (Meaning, should `@torch.no_grad()` be in my .py file that defines all the model architecture functions/classes, etc.)

Great question! `@torch.no_grad()` is a **decorator** (and also available as a **context manager**) that tells PyTorch not to build an autograd graph for the ops it wraps. Here's how to think about where to put it.

When/where to use it

A) Decorator on an inference helper (recommended for pure inference utilities)

Put it **on the function definition** of a helper that is *always* used for inference (e.g., `predict_all`). That way every call runs without grad tracking.

```python
@torch.no_grad()
def predict_all(model, X, D, Y, batch_size=256, device=None):
    model.eval()
    # ... forward passes only ...
    return outputs
```

- You do **not** "call" the decorator right before using the function.
- Define it once (e.g., in your `inference_utils.py`). When imported, the function is already wrapped.
- This is cleaner than sprinkling `with torch.no_grad():` in many places.

B) Context manager at the call site (when you *sometimes* want grads)

Use this if the same function might be used in training and inference.

```python
def forward_pass(model, X, D, Y):
    # may build a graph (used in training)
    return model(X, D, Y)

# inference time:
model.eval()
with torch.no_grad():
    out = forward_pass(model, X, D, Y)
```

Why I used `with torch.no_grad()` in training earlier

In your alternating loop, during **odd** steps you trained the predictors and **didn't want** gradients through the generator. Wrapping the generator's forward with `torch.no_grad()` avoids building an unnecessary graph and saves memory:

```python
with torch.no_grad():
    Z_hat = model.generator(x=Xb, d=Db, y=Yb)
# train classifier/NB on Z_hat; no grads into generator
```

During **even** steps you *do* want gradients to flow **through** the frozen predictors **into the generator**, so you **must not** use `no_grad()` there. Instead, freeze predictors with `requires_grad=False` and let the graph be built so the generator gets gradients.

Don't put `no_grad` "globally"

- Don't wrap your whole module/file or class definitions in a `with torch.no_grad():` block — you'd accidentally disable grads for training too.
- Don't decorate model methods that you also use in training; decorate only pure inference helpers.

`eval()` vs `no_grad()` (they're different)

- `model.eval()` switches layers like Dropout/BatchNorm to inference behavior.
- `torch.no_grad()` disables gradient tracking. You typically want **both** at inference time.

Bonus: `torch.inference_mode()`

For pure inference, you can use:

```python
@torch.inference_mode()
def predict_all(...):
    ...
```

It's like `no_grad()` but can be even faster and more memory-efficient. Don't use it inside training steps.

TL;DR

- Put `@torch.no_grad()` **on your inference helper** (e.g., `predict_all`) if it's always inference.
- Otherwise, use `with torch.no_grad():` **at the call site**.
- In your alternating training: use `no_grad()` around the generator only in the **predictor-training** phase; don't use it in the **generator-training** phase.

Great!

I would to now make a jittered plot of Z_hat (my predicted Z's) and the true Z's from the simulation. I would also like to write the correlation in the title of the plot totally—here's a tight, no-dependency (matplotlib + numpy) snippet to make a **jittered scatter** of your predicted $\hat{Z}$ vs. the **true** Z (binary 0/1), and put the **Pearson correlation** in the title.

```python
import numpy as np
import matplotlib.pyplot as plt

def jitter_plot_zhat_vs_true(Z_true, Z_hat, jitter=0.08, seed=0, z_dim_index=0):
    """
    Jittered scatter of predicted Z_hat vs true Z (0/1), with Pearson r in title.

    Z_true : array-like (n,) or (n,1), binary 0/1 from simulation
    Z_hat  : array-like (n,), (n,1), or (n,z_dim); will use column z_dim_index if needed
    jitter : float, amount of horizontal jitter around 0/1
    seed   : int, controls jitter reproducibility
    z_dim_index : which latent dim to plot if Z_hat has multiple columns
    """
    # --- to numpy & flatten ---
    Zt = Z_true
    Zh = Z_hat
    if hasattr(Zt, "detach"): Zt = Zt.detach().cpu().numpy()
    if hasattr(Zh, "detach"): Zh = Zh.detach().cpu().numpy()
    Zt = np.asarray(Zt).reshape(-1)
    Zh = np.asarray(Zh)
    if Zh.ndim == 2 and Zh.shape[1] > 1:
        Zh = Zh[:, z_dim_index]
    Zh = Zh.reshape(-1)
    assert Zt.shape[0] == Zh.shape[0], "Z_true and Z_hat must have same length"

    # --- correlation (Pearson = point-biserial for binary Z_true) ---
    # Prefer scipy if you have it; else do np.corrcoef:
    try:
        from scipy.stats import pearsonr
        r, p = pearsonr(Zt, Zh)
    except Exception:
        r = np.corrcoef(Zt, Zh)[0, 1]
        p = np.nan

    # --- jittered x positions around 0/1 ---
    rng = np.random.default_rng(seed)
    x = Zt + rng.uniform(-jitter, jitter, size=Zt.size)

    # --- plot ---
    plt.figure(figsize=(6, 4))
    plt.scatter(x, Zh, s=12, alpha=0.6)  # small points, a touch of transparency
    plt.axhline(0.5, linestyle="--", linewidth=1)  # optional: reference line
    plt.xticks([0, 1], ["Z=0", "Z=1"])
    plt.xlim(-0.4, 1.4)
    plt.xlabel("True Z (with jitter)")
    plt.ylabel("Estimated Ẑ")
    if np.isnan(p):
        title = f"Ẑ vs True Z  —  Pearson r = {r:.3f}"
    else:
        title = f"Ẑ vs True Z  —  Pearson r = {r:.3f}, p = {p:.2e}"
    plt.title(title)
    plt.tight_layout()
    plt.show()

# --- example usage ---
# If you used predict_all(...) earlier:
# out   = predict_all(model, X, D, Y)
# Z_hat = out["Z_hat"]              # (n, z_dim) tensor or np array
# Z_true is whatever you simulated, shape (n,)
# jitter_plot_zhat_vs_true(Z_true, Z_hat, jitter=0.08, seed=0, z_dim_index=0)
```

Notes:

- If your $\hat{Z}$ has multiple dimensions, **`z_dim_index`** selects which one to plot (default 0).

- For stable visuals across runs, **seed** fixes the jitter.
- The correlation is just Pearson between continuous $\hat{Z}$ and binary Z (i.e., the point-biserial correlation).

Can you modify jitter_plot_zhat_vs_true so that it has the correlation line shown (correlation between Zt and Zh)? It can be a dotted orange line.

yep! here's a drop-in upgrade to add a dotted orange **regression (correlation) line**. since $Z_t \in \{0, 1\}$, the least-squares line is $y = \alpha + \beta x$ with $\beta = r \cdot \sigma_y / \sigma_x$ and $\alpha = \bar{y} - \beta \bar{x}$. it'll pass through the group means at $x = 0$ and $x = 1$.

```python
import numpy as np
import matplotlib.pyplot as plt

def jitter_plot_zhat_vs_true(
    Z_true, Z_hat, jitter=0.08, seed=0, z_dim_index=0, show_reg_line=True
):
    """
    Jittered scatter of predicted Z_hat vs true Z (0/1), with Pearson r in title
    and an optional dotted orange regression line.

    Z_true : (n,) or (n,1) binary array/tensor
    Z_hat  : (n,), (n,1), or (n,z_dim) array/tensor
    """
    # --- to numpy & flatten ---
    Zt = Z_true
    Zh = Z_hat
    if hasattr(Zt, "detach"): Zt = Zt.detach().cpu().numpy()
    if hasattr(Zh, "detach"): Zh = Zh.detach().cpu().numpy()
    Zt = np.asarray(Zt).reshape(-1)
    Zh = np.asarray(Zh)
    if Zh.ndim == 2 and Zh.shape[1] > 1:
        Zh = Zh[:, z_dim_index]
    Zh = Zh.reshape(-1)
    assert Zt.shape[0] == Zh.shape[0], "Z_true and Z_hat must have same length"

    # --- correlation ---
    try:
        from scipy.stats import pearsonr
        r, p = pearsonr(Zt, Zh)
    except Exception:
        r = np.corrcoef(Zt, Zh)[0, 1]
        p = np.nan

    # --- regression line parameters (OLS) ---
    # slope = r * sy/sx ; intercept = mean_y - slope * mean_x
    mx, my = Zt.mean(), Zh.mean()
    sx = Zt.std(ddof=0)
    sy = Zh.std(ddof=0)
    if show_reg_line and sx > 0 and np.isfinite(r):
        slope = r * (sy / (sx + 1e-12))
        intercept = my - slope * mx
    else:
        slope = 0.0
        intercept = my  # horizontal line at mean if sx==0

    # --- jittered x for scatter ---
    rng = np.random.default_rng(seed)
    x_jit = Zt + rng.uniform(-jitter, jitter, size=Zt.size)

    # --- plot ---
```

```
    plt.figure(figsize=(6, 4))
    plt.scatter(x_jit, Zh, s=12, alpha=0.6)

    if show_reg_line:
        xline = np.linspace(-0.3, 1.3, 100)
        yline = intercept + slope * xline
        plt.plot(xline, yline, linestyle=":", color="orange", linewidth=2)

    plt.xticks([0, 1], ["Z=0", "Z=1"])
    plt.xlim(-0.4, 1.4)
    plt.xlabel("True Z (with jitter)")
    plt.ylabel("Estimated Ẑ")
    title = f"Ẑ vs True Z  —  Pearson r = {r:.3f}" + (f", p = {p:.2e}" if np.isfinite(p) else
"")
    plt.title(title)
    plt.tight_layout()
    plt.show()
```

usage:

```python
# out = predict_all(model, X, D, Y)
# Z_hat = out["Z_hat"]  # (n, z_dim) or (n,)
# Z_true = your_sim_Z   # (n,)
jitter_plot_zhat_vs_true(Z_true, Z_hat, jitter=0.08, seed=0, z_dim_index=0)
```

if you ever want to confirm the line passes through the class means:

- mean at x=0 is `Zh[Zt==0].mean()` (should equal intercept),
- mean at x=1 is `intercept + slope`.

In my code, there are 100 correlations, one per gene, where it's the correlation between each column of log_mu_hat and the corresponding column in log_mu, where log_mu_hat is defined as below:

```
# after training:
out = predict_all(model, X, D, Y, batch_size=256)

Z_hat  = out["Z_hat"].numpy()     # estimated latent(s)
D_prob = out["D_prob"].numpy()    # P(D=1)
Y_hat  = out["mu_hat"].numpy()    # NB means (predicted expression)
log_mu_hat  = out["log_mu"].numpy()    # log of NB means
```

Can you help me: 1) compute the correlation of all 100 genes, and then 2) make a histogram of these correlations?

Yep—here's a compact, vectorized way to get the per-gene correlations and a histogram.

```python
import numpy as np
import matplotlib.pyplot as plt

def per_gene_correlations(log_mu_hat, log_mu):
    """
    Pearson r for each gene (column-wise) between predicted and true log means.
```

```
    Shapes: (n_cells, n_genes) for both arrays.
    Returns: r (n_genes,) with NaN where a column has zero variance.
    """
    # to numpy
    A = log_mu_hat.detach().cpu().numpy() if hasattr(log_mu_hat, "detach") else
np.asarray(log_mu_hat)
    B = log_mu.detach().cpu().numpy()     if hasattr(log_mu, "detach")     else
np.asarray(log_mu)
    assert A.shape == B.shape, f"Shape mismatch: {A.shape} vs {B.shape}"

    # center columns
    A0 = A - A.mean(axis=0, keepdims=True)
    B0 = B - B.mean(axis=0, keepdims=True)

    # numerator and denominator for Pearson r
    num   = (A0 * B0).sum(axis=0)                            # (p,)
    den_a = np.sqrt((A0**2).sum(axis=0))                     # (p,)
    den_b = np.sqrt((B0**2).sum(axis=0))                     # (p,)
    with np.errstate(divide='ignore', invalid='ignore'):
        r = num / (den_a * den_b)                            # (p,)
    return r  # NaN where a column is constant in A or B

# 1) compute correlations (one per gene)
r = per_gene_correlations(log_mu_hat, log_mu)   # log_mu = true log means from your simulation

# quick summary
print(f"Genes: {r.size}, mean r = {np.nanmean(r):.3f}, median r = {np.nanmedian(r):.3f}, "
      f"min r = {np.nanmin(r):.3f}, max r = {np.nanmax(r):.3f}")

# 2) histogram
plt.figure(figsize=(6,4))
plt.hist(r[~np.isnan(r)], bins=20)
plt.xlabel("Per-gene Pearson correlation (predicted vs. true log μ)")
plt.ylabel("Count of genes")
plt.title(f"Per-gene correlations — mean={np.nanmean(r):.3f}, median={np.nanmedian(r):.3f}")
plt.tight_layout()
plt.show()
```

Tips:

- If you have gene names (length p), you can wrap **`r`** into a **`pandas.Series(r, index=gene_names)`** and quickly list best/worst genes: **`r.sort_values().head()`** / **`.tail()`**.
- Columns with zero variance in either **`log_mu_hat`** or **`log_mu`** will yield **`NaN`**; the histogram drops those automatically with **`~np.isnan(r)`**.

Thanks!!

This is my training so far:

```
def alternating_train(
    X, D, Y, alpha,
    batch_size=32,
    outer_iters=10,
    pred_epochs_per_odd=1,   # you can increase if you really want “convergence”
    gen_epochs_per_even=1,
    lr_cls=1e-3, lr_nb=1e-3, lr_gen=1e-3,
    z_dim=1,
    device="cpu"
):
    n, x_dim = X.shape
    _, p     = Y.shape
    y_dim    = p

    ds = XDYDataset(X, D, Y)
    loader = DataLoader(ds, batch_size=batch_size, shuffle=True, drop_last=True,
```

```
    pin_memory=True)

    model = FullArchitecture(x_dim=x_dim, y_dim=y_dim, z_dim=z_dim).to(device)

    # separate optimizers
    opt_cls = torch.optim.Adam(model.classifier.parameters(), lr=lr_cls)
    opt_nb  = torch.optim.Adam(model.geneexp.parameters(),   lr=lr_nb)
    opt_gen = torch.optim.Adam(model.generator.parameters(), lr=lr_gen)

    alpha = alpha.to(device)  # (1,p) from your simulation
    # [[KZL: Ignore optimizing alpha for now]]

    for outer in range(1, outer_iters + 1):
        is_odd = (outer % 2 == 1)

        if is_odd:
            print(f"\n[Odd {outer}] Train predictors (BCE & NB), generator frozen")

            # freeze generator
            for p_ in model.generator.parameters(): p_.requires_grad = False
            for p_ in model.classifier.parameters(): p_.requires_grad = True
            for p_ in model.geneexp.parameters():   p_.requires_grad = True

            # ---- BinaryClassifier pass ----
            for epoch in range(1, pred_epochs_per_odd + 1):
                model.train()
                running = 0.0
                for Xb, Db, Yb in loader:
                    Xb, Db, Yb = Xb.to(device), Db.to(device), Yb.to(device)
                    with torch.no_grad():    # do not backprop through generator here
                        Zb = model.generator(X=Xb, D=Db, Y=Yb)
                    logits = model.classifier(X=Xb, Z=Zb)
                    loss_bce = bce_loss(logits, Db)

                    opt_cls.zero_grad(set_to_none=True)
                    loss_bce.backward()
                    opt_cls.step()
                    running += loss_bce.item()
                print(f"  BCE: {running/len(loader):.4f}")

            # ---- NBModel pass ----
            for epoch in range(1, pred_epochs_per_odd + 1):
                model.train()
                running = 0.0
                for Xb, Db, Yb in loader:
                    Xb, Db, Yb = Xb.to(device), Db.to(device), Yb.to(device)
                    with torch.no_grad():
                        Zb = model.generator(X=Xb, D=Db, Y=Yb)
                    log_mu = model.geneexp(X=Xb, D=Db, Z=Zb)
                    loss_nb = nb_loss(log_mu, alpha, Yb)

                    opt_nb.zero_grad(set_to_none=True)
                    loss_nb.backward()
                    opt_nb.step()
                    running += loss_nb.item()
                print(f"  NB : {running/len(loader):.4f}")

        else:
            print(f"\n[Even {outer}] Train generator (min BCE + NB), predictors frozen")

            # freeze predictors
            for p_ in model.generator.parameters(): p_.requires_grad = True
            for p_ in model.classifier.parameters(): p_.requires_grad = False
            for p_ in model.geneexp.parameters():   p_.requires_grad = False

            for epoch in range(1, gen_epochs_per_even + 1):
                model.train()
                running = 0.0
                for Xb, Db, Yb in loader:
                    Xb, Db, Yb = Xb.to(device), Db.to(device), Yb.to(device)
                    Zb, logits, log_mu = model(X=Xb, D=Db, Y=Yb)   # uses fixed heads
                    loss_bce = bce_loss(logits, Db)
                    loss_nb  = nb_loss(log_mu, alpha, Yb)
                    loss     = loss_bce + loss_nb
```

```
                opt_gen.zero_grad(set_to_none=True)
                loss.backward()
                opt_gen.step()
                running += loss.item()
            print(f"  GEN: {running/len(loader):.4f}")

    return model

# =========================
# 4) Example call
# =========================
# model = alternating_train(
#     X.to(device), D.to(device), Y.to(device), alpha.to(device),
#     batch_size=32, outer_iters=10,
#     pred_epochs_per_odd=1, gen_epochs_per_even=1,
#     lr_cls=1e-3, lr_nb=1e-3, lr_gen=1e-3, z_dim=1
# )
```

Can you help me plot the loss curve on the even iterations over epochs? I would like to also keep track of the loss of `loss_bce` and `loss_nb` separately (in addition to their total, `loss`)

```
loss_bce = bce_loss(logits, Db)
loss_nb  = nb_loss(log_mu, alpha, Yb)
loss     = loss_bce + loss_nb
```

Nice—here's a tidy way to log and plot the **generator-phase** losses (even iterations), tracking **BCE**, **NB**, and **total** separately.

1) Add loss logging to your training loop

This version augments your `alternating_train` to collect averages per **generator epoch** (i.e., each pass over the loader during even iterations). It returns both `model` and a `history` dict.

```python
import matplotlib.pyplot as plt
import numpy as np

def alternating_train(
    X, D, Y, alpha,
    batch_size=32,
    outer_iters=10,
    pred_epochs_per_odd=1,
    gen_epochs_per_even=1,
    lr_cls=1e-3, lr_nb=1e-3, lr_gen=1e-3,
    z_dim=1,
    device="cpu"
):
    n, x_dim = X.shape
    _, p     = Y.shape
    y_dim    = p

    ds = XDYDataset(X, D, Y)
    loader = DataLoader(ds, batch_size=batch_size, shuffle=True, drop_last=True,
pin_memory=True)

    model = FullArchitecture(x_dim=x_dim, y_dim=y_dim, z_dim=z_dim).to(device)

    opt_cls = torch.optim.Adam(model.classifier.parameters(), lr=lr_cls)
    opt_nb  = torch.optim.Adam(model.geneexp.parameters(),   lr=lr_nb)
    opt_gen = torch.optim.Adam(model.generator.parameters(), lr=lr_gen)
```

```python
    alpha = alpha.to(device)

    # ---- NEW: history containers for even iterations ----
    history = {
        "even_step": [],        # monotonically increasing counter for each generator epoch
        "outer_iter": [],       # which outer iteration this belongs to (even only)
        "gen_epoch": [],        # which epoch within the even block (1..gen_epochs_per_even)
        "loss_total": [],
        "loss_bce": [],
        "loss_nb":  [],
    }
    even_step = 0

    for outer in range(1, outer_iters + 1):
        is_odd = (outer % 2 == 1)

        if is_odd:
            print(f"\n[Odd {outer}] Train predictors (BCE & NB), generator frozen")

            for p_ in model.generator.parameters(): p_.requires_grad = False
            for p_ in model.classifier.parameters(): p_.requires_grad = True
            for p_ in model.geneexp.parameters():   p_.requires_grad = True

            # ---- BinaryClassifier pass ----
            for epoch in range(1, pred_epochs_per_odd + 1):
                model.train()
                running = 0.0
                for Xb, Db, Yb in loader:
                    Xb, Db, Yb = Xb.to(device), Db.to(device), Yb.to(device)
                    with torch.no_grad():
                        Zb = model.generator(X=Xb, D=Db, Y=Yb)
                    logits = model.classifier(X=Xb, Z=Zb)
                    loss_bce = bce_loss(logits, Db)

                    opt_cls.zero_grad(set_to_none=True)
                    loss_bce.backward()
                    opt_cls.step()
                    running += loss_bce.item()
                print(f"  BCE: {running/len(loader):.4f}")

            # ---- NBModel pass ----
            for epoch in range(1, pred_epochs_per_odd + 1):
                model.train()
                running = 0.0
                for Xb, Db, Yb in loader:
                    Xb, Db, Yb = Xb.to(device), Db.to(device), Yb.to(device)
                    with torch.no_grad():
                        Zb = model.generator(X=Xb, D=Db, Y=Yb)
                    log_mu = model.geneexp(X=Xb, D=Db, Z=Zb)
                    loss_nb_ = nb_loss(log_mu, alpha, Yb)

                    opt_nb.zero_grad(set_to_none=True)
                    loss_nb_.backward()
                    opt_nb.step()
                    running += loss_nb_.item()
                print(f"  NB : {running/len(loader):.4f}")

        else:
            print(f"\n[Even {outer}] Train generator (min BCE + NB), predictors frozen")

            for p_ in model.generator.parameters(): p_.requires_grad = True
            for p_ in model.classifier.parameters(): p_.requires_grad = False
            for p_ in model.geneexp.parameters():   p_.requires_grad = False

            for epoch in range(1, gen_epochs_per_even + 1):
                model.train()
                sum_total = 0.0
                sum_bce   = 0.0
                sum_nb    = 0.0

                for Xb, Db, Yb in loader:
                    Xb, Db, Yb = Xb.to(device), Db.to(device), Yb.to(device)
                    Zb, logits, log_mu = model(X=Xb, D=Db, Y=Yb)
                    loss_bce = bce_loss(logits, Db)
                    loss_nb_ = nb_loss(log_mu, alpha, Yb)
```

```
                loss     = loss_bce + loss_nb_

                opt_gen.zero_grad(set_to_none=True)
                loss.backward()
                opt_gen.step()

                sum_total += loss.item()
                sum_bce   += loss_bce.item()
                sum_nb    += loss_nb_.item()

            # epoch averages
            avg_total = sum_total / len(loader)
            avg_bce   = sum_bce   / len(loader)
            avg_nb    = sum_nb    / len(loader)
            print(f"  GEN epoch {epoch}: total={avg_total:.4f}  BCE={avg_bce:.4f}  NB=
{avg_nb:.4f}")

            # ---- NEW: log to history
            even_step += 1
            history["even_step"].append(even_step)
            history["outer_iter"].append(outer)
            history["gen_epoch"].append(epoch)
            history["loss_total"].append(avg_total)
            history["loss_bce"].append(avg_bce)
            history["loss_nb"].append(avg_nb)

    return model, history
```

⚠️ Small reminder: your calls use capitalized keywords (**`X=`**, **`D=`**, **`Y=`**). Make sure your modules' **`forward`** signatures accept those names; otherwise switch to positional args.

---

2) Plotting function

Three curves (total, BCE, NB) against generator epochs (i.e., each even-iteration epoch). Includes optional EMA smoothing.

```python
def plot_gen_loss_curves(history, smooth_alpha=None):
    """
    Plot generator-phase losses logged during even iterations.

    smooth_alpha: float in (0,1); if provided, plot an exponential moving average too.
    """
    x   = np.array(history["even_step"])
    tot = np.array(history["loss_total"])
    bce = np.array(history["loss_bce"])
    nb  = np.array(history["loss_nb"])

    plt.figure(figsize=(7,4.5))
    plt.plot(x, tot, label="Total (BCE+NB)")
    plt.plot(x, bce, label="BCE (D)", linestyle="--")
    plt.plot(x, nb,  label="NB  (Y)", linestyle=":")

    if smooth_alpha is not None:
        def ema(v, a):
            y = np.empty_like(v, dtype=float)
            m = v[0]
            for i, val in enumerate(v):
                m = a*val + (1-a)*m
                y[i] = m
            return y
        plt.plot(x, ema(tot, smooth_alpha), label=f"EMA total (α={smooth_alpha})", linewidth=2)

    plt.xlabel("Generator epochs across even iterations")
    plt.ylabel("Loss (avg per epoch)")
    plt.title("Generator-phase loss curves (even iterations)")
```

```
    plt.legend()
    plt.grid(alpha=0.2)
    plt.tight_layout()
    plt.show()
```

Example usage

```python
model, hist = alternating_train(
    X.to(device), D.to(device), Y.to(device), alpha.to(device),
    batch_size=32, outer_iters=10,
    pred_epochs_per_odd=1, gen_epochs_per_even=2,  # e.g., 2 gen epochs per even block
    lr_cls=1e-3, lr_nb=1e-3, lr_gen=1e-3, z_dim=1, device=device
)

plot_gen_loss_curves(hist, smooth_alpha=0.3)
```

This will give you a clean curve of how the generator-phase objective decomposes and evolves over time. If you later want **odd-phase** curves too (BCE-only and NB-only), we can add similar logging there.

In my current simulation code:

```
# 5. Candidate simulate GamPoi Y ~ D + X + Z with log-link
beta_2 = torch.randn(d + 1, p) + 0 # try no biases for now
gamma_2 = torch.randn(k, p) + 0
```

How do I replace it so that beta_2 is the all-0 matrix?

Use zeros instead of randoms. Two easy, safe options:

```python
# assuming you have X (n × d) and want beta_2 shape (d+1) × p
device = X.device
dtype  = X.dtype

beta_2  = torch.zeros(d + 1, p, device=device, dtype=dtype)  # all zeros
gamma_2 = torch.randn(k, p, device=device, dtype=dtype)      # keep as before (if you want)
```

or, if you already have `X` and want a shorthand tied to its dtype/device:

```python
beta_2 = X.new_zeros(d + 1, p)   # all zeros on same device/dtype as X
```

(Optional) since these are “true” simulation parameters and not trainable, you can also mark them non-grad:

```python
beta_2.requires_grad_(False)
```

```
gamma_2.requires_grad_(False)
```

This is my current simulation. (A pretty extreme setting where X and D have nothing to do with Y):

```
# ============================================================ #
# 1. dims
# observation: larger d gives a much larger mu = exp(X @ beta)
n, p, d, k = 1000, 100, 4, 1

# target prevalence for D
logit = lambda u: math.log(u/(1-u))

# ============================================================ #
# 2. simulate X as d independent binary covariates
px = torch.full((d,), 0.5)                          # per-column P(X=1); tweak if you like
X = torch.bernoulli(px.expand(n, d)).float()        # shape [n, d]

# ============================================================ #
# 3. simulate Z independent of X
p_z = 0.5
Z = torch.bernoulli(torch.full((n, k), float(p_z)))

# ============================================================ #
# 4. simulate D via logistic link with calibrated strengths
# 4.1 draw beta_1, then rescale so sd(X @ beta_1) hits a target (keeps away from saturation)
# beta_1 = torch.randn(d, 1)
beta_1  = torch.zeros(d, 1)  # all zeros
with torch.no_grad():
    sd_target = 0.8
    xb = X @ beta_1
    beta_1 *= (sd_target / (xb.std() + 1e-8))

# 4.2 choose Z strength by variance-ratio on the logit scale: Var(Z*gamma_1) = r_star *
Var(X*beta_1)
r_star = 0.30 # try 0.25–0.50 for clear but non-extreme effect
with torch.no_grad():
    sd_xb = (X @ beta_1).std()
    sd_z  = Z.std()                                    # for Bernoulli(~0.52), ~0.50
    gamma_1 = (torch.sqrt(torch.tensor(r_star)) * sd_xb / (sd_z + 1e-8)).view(1,1)  #
positive effect
    # flip sign if you want inverse effect: gamma_1 = -gamma_1

logits_D = X @ beta_1 + Z @ gamma_1                          # linear predictor, no
intercept
D = torch.bernoulli(torch.sigmoid(logits_D)).view(n,1)
# quick checks
# print(f"sd(Xb)={sd_xb.item():.3f} | sd(Zg)={(Z@gamma_1).std().item():.3f} | Var
ratio≈{((Z@gamma_1).var()/ (X@beta_1).var()).item():.3f}")
# print(f"Simulated prevalence mean(D)={D.mean().item():.3f}")

# ============================================================ #
# 5. Candidate simulate GamPoi Y ~ D + X + Z with log-link
# beta_2 = torch.randn(d + 1, p) + 0 # try no biases for now
beta_2  = torch.zeros(d + 1, p)  # all zeros
gamma_2 = torch.randn(k, p) + 0

X_aug = torch.cat([X, D], dim=1)                       # [n, d+1]
log_mu = X_aug @ beta_2 + Z @ gamma_2                   # no intercept
mu = torch.exp(log_mu)

# NB via Poisson-Gamma (Gamma-Poisson) with per-gene alpha
alpha = torch.abs(torch.randn(1, p)) + 0.5              # >0
r = 1.0 / alpha
gamma_rate = torch.distributions.Gamma(concentration=r, rate=r / (mu + 1e-8)).sample()
Y = torch.poisson(gamma_rate)
```

How do make a scatterplot of **`logits_D`** against the predicted **`D_logit`**?

Absolutely—here's a clean snippet to scatter **true** vs **predicted** logits for D, with a 45° reference line and a dotted orange regression line, plus the Pearson r in the title.

```python
import numpy as np
import matplotlib.pyplot as plt

# 1) Get predicted logits from your trained model
out = predict_all(model, X, D, Y, batch_size=256)
D_logit_pred = out["D_logit"]  # (n,1) tensor or np.array

# 2) Convert both to flat numpy arrays
def to_flat_np(a):
    if hasattr(a, "detach"):
        a = a.detach().cpu().numpy()
    return np.asarray(a).reshape(-1)

x_true = to_flat_np(logits_D)     # true logit(Xβ1 + Zγ1)
y_pred = to_flat_np(D_logit_pred) # predicted logits from classifier

# 3) Correlation + regression line (OLS)
from math import isfinite
try:
    from scipy.stats import pearsonr
    r, p = pearsonr(x_true, y_pred)
except Exception:
    r = np.corrcoef(x_true, y_pred)[0, 1]
    p = np.nan

mx, my = x_true.mean(), y_pred.mean()
sx, sy = x_true.std(ddof=0), y_pred.std(ddof=0)
if sx > 0 and np.isfinite(r):
    slope = r * (sy / (sx + 1e-12))
    intercept = my - slope * mx
else:
    slope, intercept = 0.0, my

# 4) Plot
plt.figure(figsize=(6, 5))
plt.scatter(x_true, y_pred, s=12, alpha=0.6)

# 45° identity line
lo = min(x_true.min(), y_pred.min())
hi = max(x_true.max(), y_pred.max())
plt.plot([lo, hi], [lo, hi], linestyle="--", linewidth=1, color="k", label="y = x")

# Dotted orange regression (correlation) line
xline = np.linspace(lo, hi, 100)
yline = intercept + slope * xline
plt.plot(xline, yline, linestyle=":", linewidth=2, color="orange", label="OLS fit")

plt.xlabel("True logit for D (Xβ₁ + Zγ₁)")
plt.ylabel("Predicted logit for D (classifier)")
title = f"Predicted vs True D logits — r = {r:.3f}" + (f", p = {p:.2e}" if isfinite(p) else "")
plt.title(title)
plt.legend()
plt.tight_layout()
plt.show()
```

If you ever want a **probability** view instead, just apply a sigmoid to both and plot $\hat{p}$ vs p:

```python
sigmoid = lambda t: 1 / (1 + np.exp(-t))
p_true = sigmoid(x_true)
p_pred = sigmoid(y_pred)
# …then scatter p_true vs p_pred, with y=x reference.
```

Can you wrap this as a function?

Here you go — a tidy, self-contained function that makes the scatterplot of **true** vs **predicted** D logits, overlays the 45° line and a dotted orange OLS regression line, and returns the correlation and fitted line stats.

```python
def plot_d_logit_true_vs_pred(
    logits_D_true,
    model=None, X=None, D=None, Y=None,
    D_logit_pred=None,
    batch_size=256,
    device=None,
    title_prefix="Predicted vs True D logits",
):
    """
    Scatter plot of true vs predicted logits for D, with y=x and a dotted OLS fit.
    Computes Pearson r (and p if scipy is available), and returns plot + stats.

    You can either:
      - Pass D_logit_pred directly, OR
      - Pass model, X, D, Y and it will call predict_all(...) to get D_logit.

    Args
    ----
    logits_D_true : array/tensor of shape (n,) or (n,1)
    model, X, D, Y: used only if D_logit_pred is None
    D_logit_pred  : array/tensor of shape (n,) or (n,1) with predicted logits
    batch_size    : batch size for predict_all if used
    device        : device for predict_all if used
    title_prefix  : str for the plot title

    Returns
    -------
    dict with keys:
        r, p, slope, intercept, fig, ax
    """
    import numpy as np
    import matplotlib.pyplot as plt

    # helper
    def to_flat_np(a):
        if hasattr(a, "detach"):
            a = a.detach().cpu().numpy()
        return np.asarray(a).reshape(-1)

    # get predictions if not provided
    if D_logit_pred is None:
        assert model is not None and X is not None and D is not None and Y is not None, \
            "Either provide D_logit_pred, or provide model, X, D, and Y."
        out = predict_all(model, X, D, Y, batch_size=batch_size, device=device)
        D_logit_pred = out["D_logit"]

    x_true = to_flat_np(logits_D_true)
    y_pred = to_flat_np(D_logit_pred)
```

```
    # correlation and (optional) p-value
    try:
        from scipy.stats import pearsonr
        r, p = pearsonr(x_true, y_pred)
    except Exception:
        r = np.corrcoef(x_true, y_pred)[0, 1]
        p = float("nan")

    # OLS line: slope = r * sy/sx ; intercept = mean_y - slope*mean_x
    mx, my = x_true.mean(), y_pred.mean()
    sx, sy = x_true.std(ddof=0), y_pred.std(ddof=0)
    if sx > 0 and np.isfinite(r):
        slope = r * (sy / (sx + 1e-12))
        intercept = my - slope * mx
    else:
        slope, intercept = 0.0, my  # fallback if sx==0

    # plot
    fig, ax = plt.subplots(figsize=(6, 5))
    ax.scatter(x_true, y_pred, s=12, alpha=0.6)

    lo = float(min(x_true.min(), y_pred.min()))
    hi = float(max(x_true.max(), y_pred.max()))
    ax.plot([lo, hi], [lo, hi], ls="--", lw=1, color="k", label="y = x")

    xline = np.linspace(lo, hi, 100)
    yline = intercept + slope * xline
    ax.plot(xline, yline, ls=":", lw=2, color="orange", label="OLS fit")

    ax.set_xlabel("True logit for D (Xβ₁ + Zγ₁)")
    ax.set_ylabel("Predicted logit for D (classifier)")
    ttl = f"{title_prefix} — r = {r:.3f}" + (f", p = {p:.2e}" if np.isfinite(p) else "")
    ax.set_title(ttl)
    ax.legend()
    fig.tight_layout()
    return {"r": r, "p": p, "slope": slope, "intercept": intercept, "fig": fig, "ax": ax}
```

Example

```python
# after training
out_stats = plot_d_logit_true_vs_pred(
    logits_D_true=logits_D,
    model=model, X=X, D=D, Y=Y,
    batch_size=256
)
print(out_stats["r"], out_stats["slope"], out_stats["intercept"])
```

If you already have **`D_logit_pred`** from **`predict_all`**, pass it directly:

```python
pred = predict_all(model, X, D, Y, batch_size=256)
plot_d_logit_true_vs_pred(logits_D, D_logit_pred=pred["D_logit"])
```