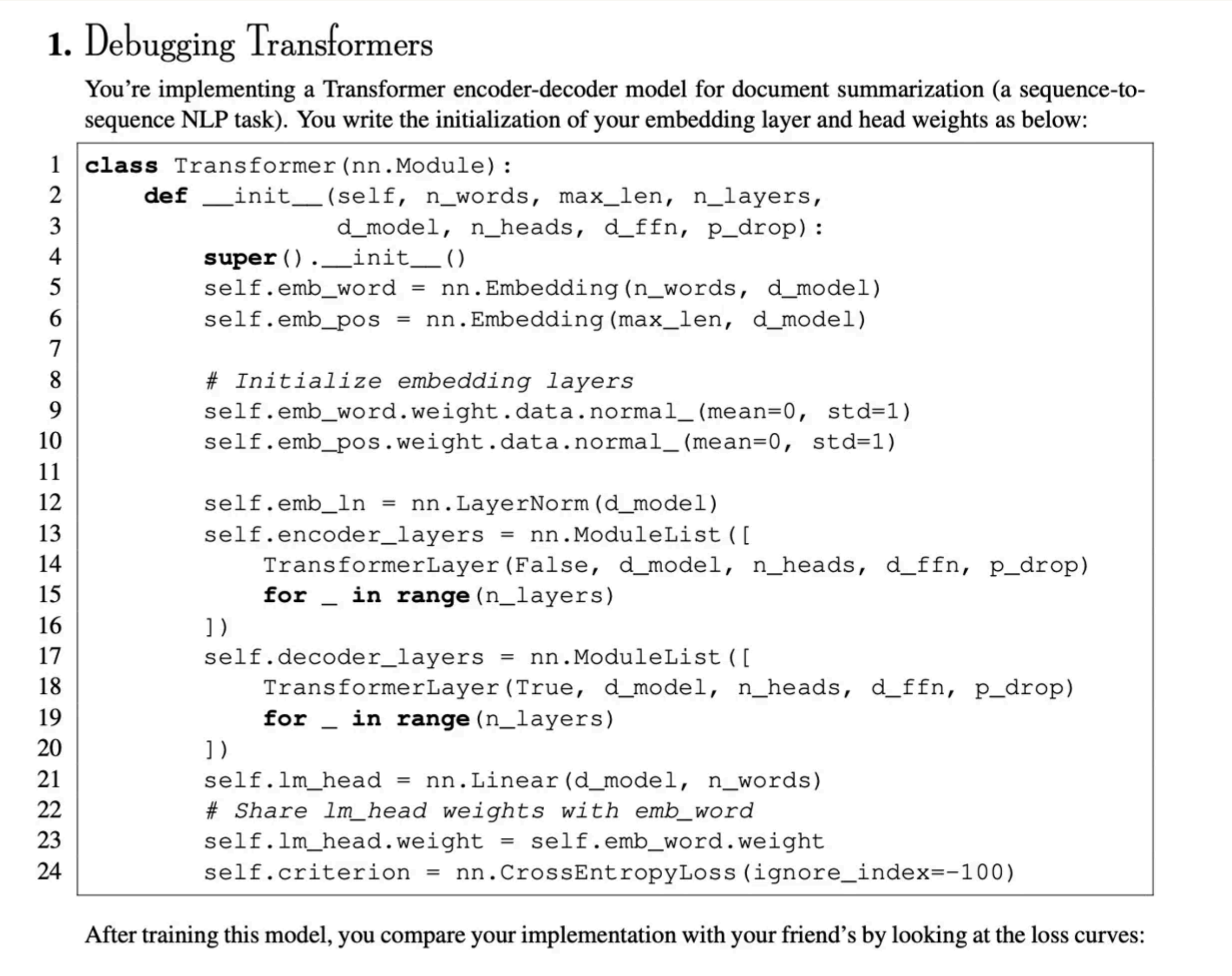
HW 12 Written - Sonnet 4.5

QUESTION 1

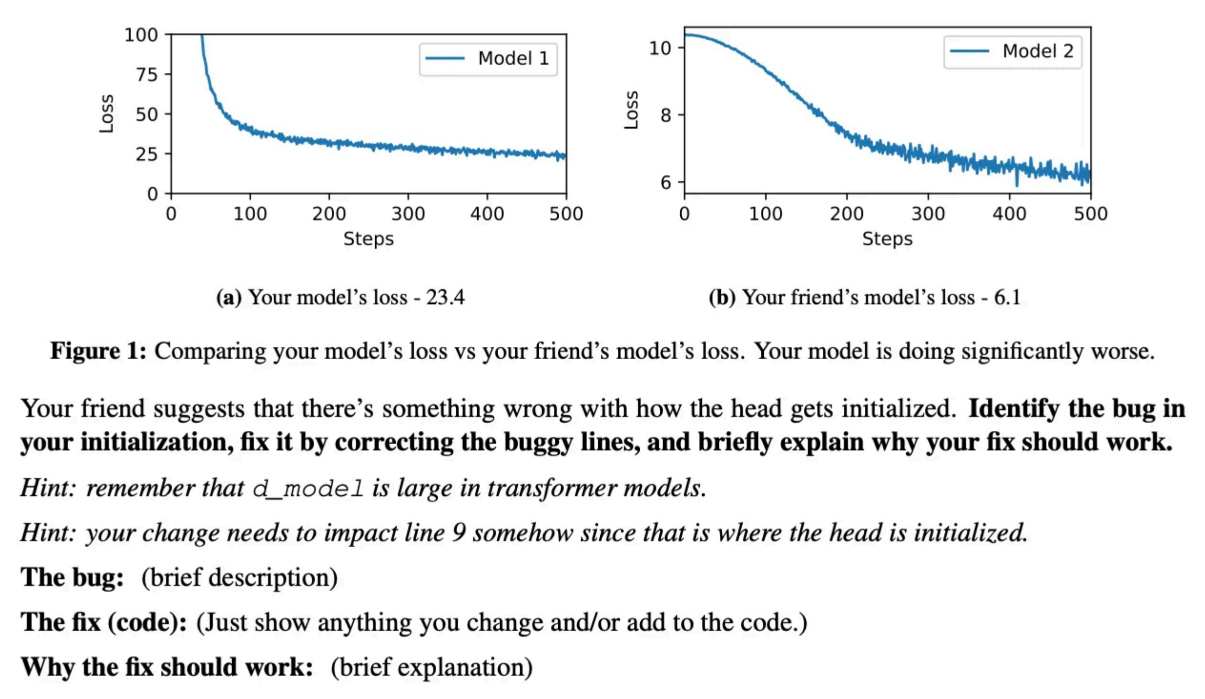


1. Debugging Transformers

You're implementing a Transformer encoder-decoder model for document summarization (a sequence-to-sequence NLP task). You write the initialization of your embedding layer and head weights as below:

```
1  class Transformer(nn.Module):
2      def __init__(self, n_words, max_len, n_layers,
3                   d_model, n_heads, d_ffn, p_drop):
4          super().__init__()
5          self.emb_word = nn.Embedding(n_words, d_model)
6          self.emb_pos = nn.Embedding(max_len, d_model)
7
8          # Initialize embedding layers
9          self.emb_word.weight.data.normal_(mean=0, std=1)
10         self.emb_pos.weight.data.normal_(mean=0, std=1)
11
12         self.emb_ln = nn.LayerNorm(d_model)
13         self.encoder_layers = nn.ModuleList([
14             TransformerLayer(False, d_model, n_heads, d_ffn, p_drop)
15             for _ in range(n_layers)
16         ])
17         self.decoder_layers = nn.ModuleList([
18             TransformerLayer(True, d_model, n_heads, d_ffn, p_drop)
19             for _ in range(n_layers)
20         ])
21         self.lm_head = nn.Linear(d_model, n_words)
22         # Share lm_head weights with emb_word
23         self.lm_head.weight = self.emb_word.weight
24         self.criterion = nn.CrossEntropyLoss(ignore_index=-100)
```

After training this model, you compare your implementation with your friend's by looking at the loss curves:



(a) Your model's loss - 23.4

(b) Your friend's model's loss - 6.1

Figure 1: Comparing your model's loss vs your friend's model's loss. Your model is doing significantly worse.

Your friend suggests that there's something wrong with how the head gets initialized. **Identify the bug in your initialization, fix it by correcting the buggy lines, and briefly explain why your fix should work.**

Hint: remember that `d_model` *is large in transformer models.*

Hint: your change needs to impact line 9 somehow since that is where the head is initialized.

The bug: (brief description)

The fix (code): (Just show anything you change and/or add to the code.)

Why the fix should work: (brief explanation)

I want you to solve some Deep Learning related questions. Can you please solve the question in the attached images?

11/30/2025, 11:46:00 PM

```
Thought process: Diagnosed embedding initialization bug causing training instability.

Let me analyze this debugging problem for a Transformer model.

Looking at the code:
- Lines 5-6: Creating embedding layers for words and positions
- Lines 9-10: Initializing embedding weights with normal distribution (mean=0, std=1)
- Line 21: Creating a language model head `self.lm_head = nn.Linear(d_model, n_words)`
- Line 23: **Weight sharing** - `self.lm_head.weight = self.emb_word.weight`

The problem is clear from the loss curves:
- Model 1 (user's model): Loss ~23.4 - much worse
- Model 2 (friend's model): Loss ~6.1 - much better

The hints say:
1. `d_model` is large in transformer models
2. The change needs to impact line 9 (where the head is initialized)

The bug is in **line 9**:
`self.emb_word.weight.data.normal_(mean=0, std=1)`

When `d_model` is large (e.g., 512 or 768), initializing embeddings with std=1 leads to embeddings with very large norms. Specifically, if each dimension is sampled from N(0,1), then the
```

```
expected squared norm of an embedding vector is `d_model` (since it's the sum of `d_model` squared standard normals, each with expected value 1).

This means:
- Expected norm = sqrt(d_model) ≈ sqrt(512) ≈ 22.6 or sqrt(768) ≈ 27.7

When the embeddings have such large norms, and the lm_head shares weights with emb_word (line 23), the logits before softmax will be extremely large. This causes:
1. Numerical instability in softmax
2. Very confident (but wrong) predictions early in training
3. Vanishing gradients due to saturated softmax

I'll adjust the initialization to scale the standard deviation by the inverse square root of the model dimension. This technique normalizes the embedding vector lengths, preventing numerical instability and improving training dynamics. By using `std=1/sqrt(d_model)`, we ensure that the embedding vectors have more consistent and manageable magnitudes across different model sizes, which helps stabilize the training process and prevents extreme logit values.

The key is maintaining a consistent embedding vector scale regardless of the model's dimensionality, allowing for more reliable and predictable neural network behavior.
```

What I like about Sonnet's explanation here is that it actually computes the expected norm of the embedding vector, which could give a student a more concrete understanding of why controlling the variance is important.

Looking at this Transformer debugging problem, I can identify the issue by analyzing the loss curves and the hints provided.

The Bug

The bug is in **line 9** (and line 10): the embedding weights are initialized with `std=1`, which is too large when `d_model` is large (typical values are 512, 768, etc.).

```
self.emb_word.weight.data.normal_(mean=0, std=1)  # Bug!
```

When `d_model` is large, each embedding vector has an expected squared norm of approximately `d_model` (since you're summing `d_model` independent N(0,1) random variables). This means the expected norm is `√d_model` ≈ 22-27 for typical transformer dimensions.

One thing I would have liked to see is a brief explanation of how we get the expected norm; this could be helpful for students who don't have a great handle on probability.

The Fix

Scale the standard deviation by `1/√d_model`:

```
# Line 9 - fixed
self.emb_word.weight.data.normal_(mean=0, std=d_model**-0.5)
# Line 10 - also fix this
self.emb_pos.weight.data.normal_(mean=0, std=d_model**-0.5)
```

Or equivalently:

```
import math
std = 1.0 / math.sqrt(d_model)
self.emb_word.weight.data.normal_(mean=0, std=std)
self.emb_pos.weight.data.normal_(mean=0, std=std)
```

Why This Fix Works

1. **Prevents exploding logits**: Since `lm_head` shares weights with `emb_word` (line 23), the poorly initialized embeddings directly affect the output layer.

With std=1 and large `d_model`, the logits become extremely large (magnitude ~20-30), causing numerical instability.

2. **Normalized embedding norms**: With `std = 1/√d_model`, each embedding vector has expected squared norm ≈ 1, keeping values in a reasonable range throughout the network.

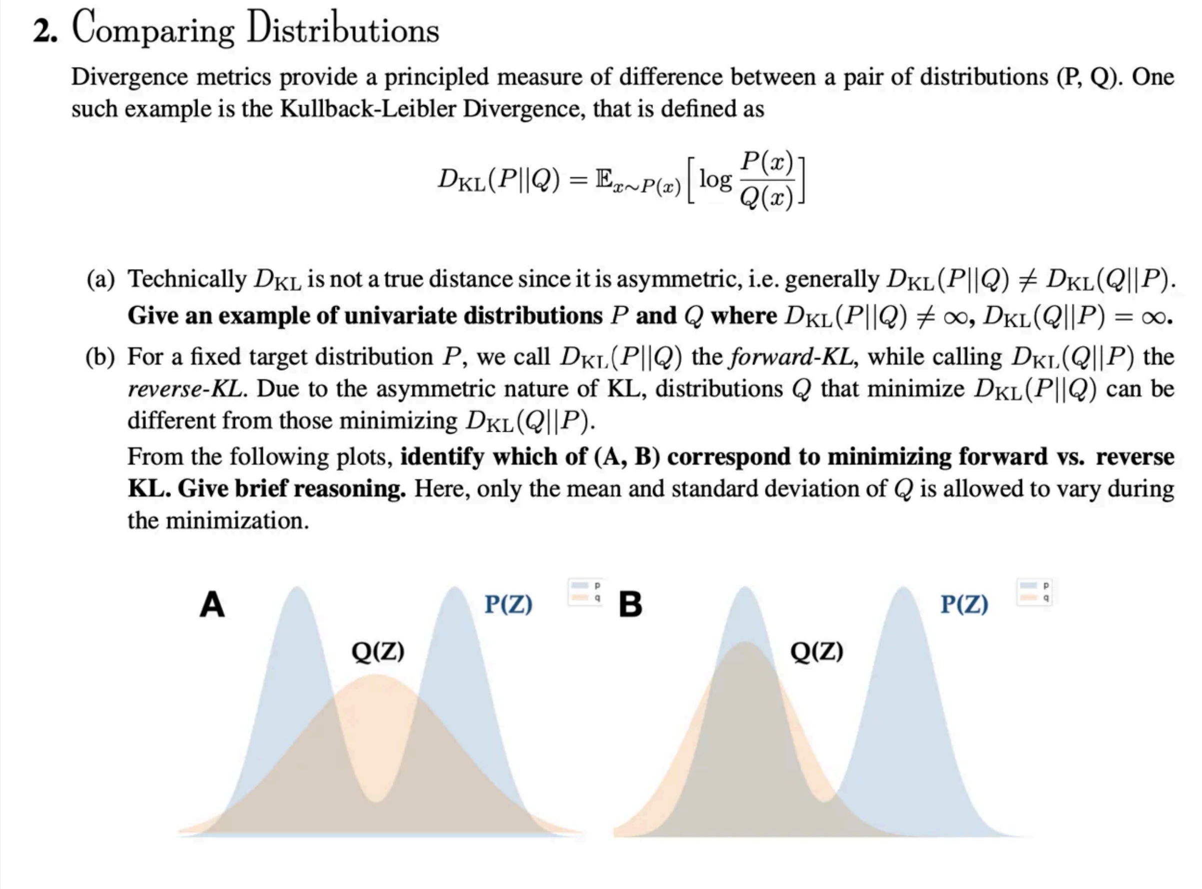
3. **Better gradient flow**: Smaller initial weights prevent saturation in softmax, allowing gradients to flow properly during early training.

This is standard initialization for embeddings in transformers (similar to Xavier/Glorot initialization principle).

11/30/2025, 11:46:28 PM

Another strength of Sonnet's response in this question is that it has great step-by-step reasoning, first finding where the issue lies in the code, doing some brief computation/analysis to explain why that line might be problematic, and then proposes a fix based on fundamental deep learning principles.

QUESTION 2



2. Comparing Distributions

Divergence metrics provide a principled measure of difference between a pair of distributions (P, Q). One such example is the Kullback-Leibler Divergence, that is defined as

$$D_{\mathrm{KL}}(P||Q) = \mathbb{E}_{x\sim P(x)}\Big[\log\frac{P(x)}{Q(x)}\Big]$$

(a) Technically D_{KL} is not a true distance since it is asymmetric, i.e. generally $D_{\mathrm{KL}}(P||Q) \neq D_{\mathrm{KL}}(Q||P)$. **Give an example of univariate distributions P and Q where $D_{\mathrm{KL}}(P||Q) \neq \infty$, $D_{\mathrm{KL}}(Q||P) = \infty$.**

(b) For a fixed target distribution P, we call $D_{\mathrm{KL}}(P||Q)$ the *forward-KL*, while calling $D_{\mathrm{KL}}(Q||P)$ the *reverse-KL*. Due to the asymmetric nature of KL, distributions Q that minimize $D_{\mathrm{KL}}(P||Q)$ can be different from those minimizing $D_{\mathrm{KL}}(Q||P)$.

From the following plots, **identify which of (A, B) correspond to minimizing forward vs. reverse KL. Give brief reasoning.** Here, only the mean and standard deviation of Q is allowed to vary during the minimization.

I have attached another question. Can you solve it?

11/30/2025, 11:49:01 PM

```
Thought process: Analyzed KL divergence properties to distinguish
mode-covering from mode-seeking behavior.

This is asking about KL divergence and which plot corresponds to
minimizing forward KL vs reverse KL.

Let me think about this carefully.

Forward KL: $D_{KL}(P||Q) = \mathbb{E}_{x \sim P(x)}[\log
\frac{P(x)}{Q(x)}]$

Reverse KL: $D_{KL}(Q||P) = \mathbb{E}_{x \sim Q(x)}[\log
```

Sonnet gets this question correct; I find it interesting that Sonnet does not reason from first principles. Instead it starst from statements such as. “Reverse KL is mode-seeking” that it knows to be true, and reasons backwards using those “theorems” to describe the desired distribution of mass. This might shed some doubt on to what extent these LLMs are doing actual reasoning vs memorizing some training data (it seems likely that this very basic property of KL would be in the data).

```
\frac{Q(x)}{P(x)}]$

Key properties:
- Forward KL is zero-forcing: Q must cover all of P (otherwise if
P(x) > 0 but Q(x) = 0, we get infinity)
- Reverse KL is mode-seeking: Q will concentrate on modes of P
(if Q(x) > 0 but P(x) ≈ 0, we get large penalty)

Looking at the plots:
- P(Z) appears to be bimodal (two peaks)
- We're trying to fit Q(Z) to approximate P(Z)

Plot A: Q(Z) is broad and covers both modes of P(Z), but doesn't
concentrate strongly on either mode.

Plot B: Q(Z) is narrower and concentrates on one mode of P(Z),
missing the other mode.

Forward KL minimization (minimizing $D_{KL}(P||Q)$):
- This is expectation under P
- If P has mass somewhere, Q must have mass there too, otherwise
we get $\log \frac{P(x)}{0} = \infty$
- This forces Q to be "zero-avoiding" or "mode-covering"
- Q will spread out to cover all modes of P
- This corresponds to Plot A

Reverse KL minimization (minimizing $D_{KL}(Q||P)$):
- This is expectation under Q



- If Q has mass somewhere but P doesn't, we get large penalty
from $\log \frac{Q(x)}{P(x)}$
- Q will avoid putting mass where P has little mass
- This is "mode-seeking" - Q will concentrate on the highest
mode(s) of P
```

```
- This corresponds to Plot B

I'll analyze the KL divergence characteristics for these
distribution approximations. When minimizing forward KL, the goal
is comprehensive coverage, forcing Q to spread across all P's
modes. This means Q must have non-zero probability wherever P has
non-zero probability, resulting in a broader distribution that
captures both peaks.

In contrast, reverse KL minimization prioritizes concentration on
P's most significant modes. Q becomes more selective, focusing
intensely on the most prominent peak while potentially neglecting
less significant modes. This leads to a narrower, more peaked
distribution that captures the primary mode of P.

The key distinction lies in the expectation: forward KL looks
from P's perspective, demanding complete coverage, while reverse
KL looks from Q's perspective, seeking to align with P's most
important regions.
```

Looking at this KL divergence question:

Part (a): Example where D_KL(P||Q) ≠ ∞ but D_KL(Q||P) = ∞

Example:

- P = N(0, 1) (standard normal)
- Q = N(0, 4) (normal with variance 4)

Although this example is correct, the explanation is slightly misguided. I don't think its mathematically precise to argue that Q has support everywhere that P does, because the same holds true vice versa (a normal dist. has support everywhere), even though the intuition is correct.

Why this works:

- D_KL(P||Q) is finite because Q has support everywhere P does (both have full real line support)

- D_KL(Q||P) = ∞ because Q has heavier tails than P. When computing the expectation under Q, we sample from regions where Q(x) > 0 but P(x) → 0 (in the tails), making log(Q(x)/P(x)) → ∞

More generally: whenever Q has support that extends beyond P's support, reverse KL diverges.

Part (b): Forward vs. Reverse KL

Answer: Plot A corresponds to forward KL, Plot B corresponds to reverse KL

Reasoning:

Forward KL: D_KL(P||Q) (Plot A)

- Expectation is taken under P
- If P(x) > 0 but Q(x) ≈ 0, we get log(P(x)/Q(x)) → ∞
- This forces Q to be **zero-avoiding** or **mode-covering**
- Q must "cover" all regions where P has mass
- Result: Q spreads out to cover both modes of P (even if it doesn't match either perfectly)

Reverse KL: D_KL(Q||P) (Plot B)

- Expectation is taken under Q
- If Q(x) > 0 but P(x) ≈ 0, we get log(Q(x)/P(x)) → ∞
- This forces Q to be **mode-seeking**
- Q avoids placing mass where P has little probability
- Result: Q concentrates on one dominant mode of P, ignoring the other

Sonnet gets this question correct; one thing that was very impresive to me is how its able to “reverse” its thought process to create a pedogically better explanation.

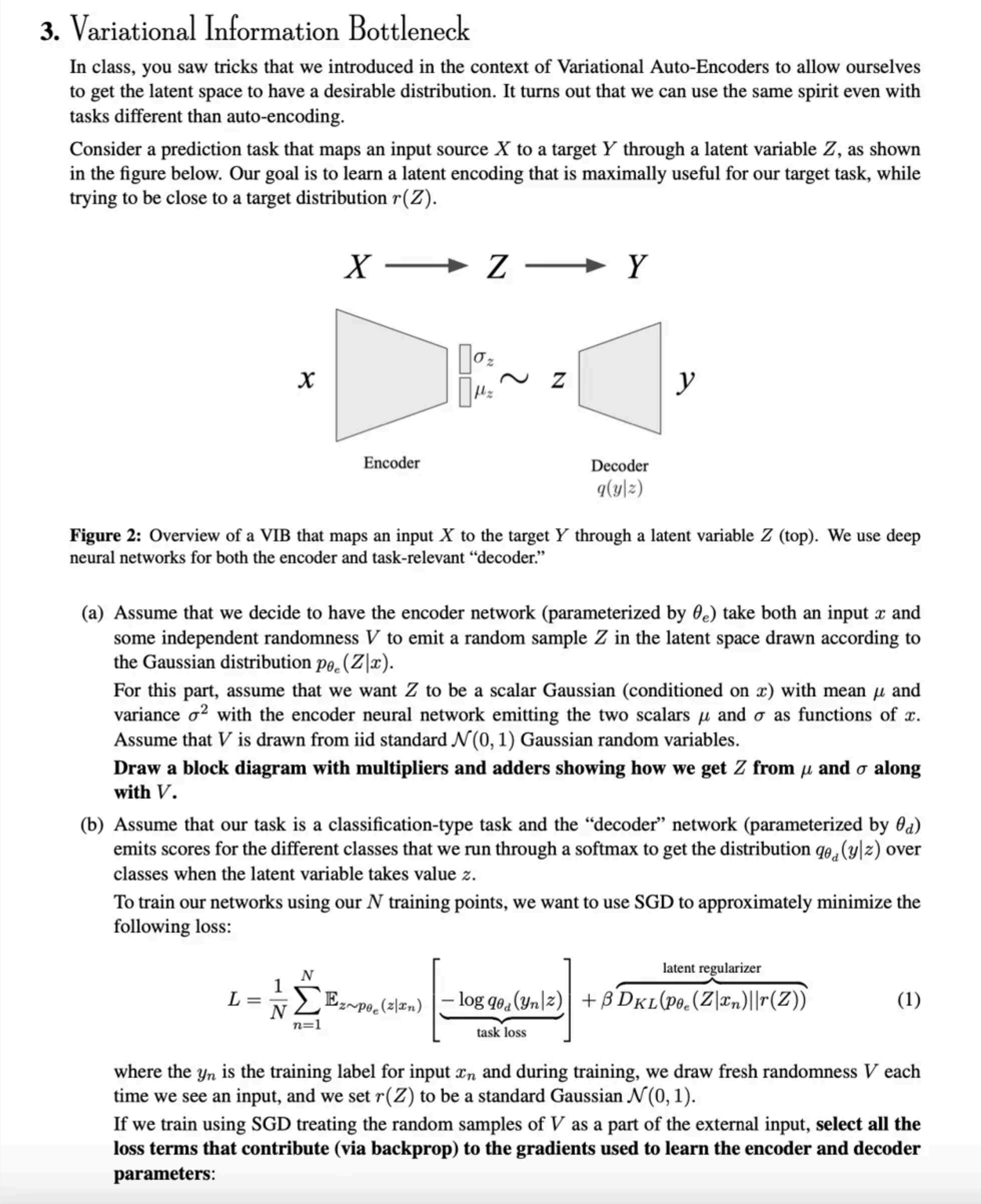
The correct way to reason about this problem from first principles is to think about the magnitude of P(x) / Q(x) vs Q(x) / P(x) depending on how the probability mass is concentrated; from this we can derive claims such as “forward KL is zero-forcing” or “reverse KL is mode-seeking.”

During the reasoning phase, Sonnet seems to start from these statements that it knows to be true, and instead reasons backwards using those “theorems” to describe the desired distribution of mass. However, when generating an explanation for the user, it takes the correct approach of starting from first principles.

In summary: forward KL is inclusive/covering (Plot A), while reverse KL is exclusive/seeking (Plot B).

11/30/2025, 11:49:25 PM

QUESTION 3



3. Variational Information Bottleneck

In class, you saw tricks that we introduced in the context of Variational Auto-Encoders to allow ourselves to get the latent space to have a desirable distribution. It turns out that we can use the same spirit even with tasks different than auto-encoding.

Consider a prediction task that maps an input source X to a target Y through a latent variable Z, as shown in the figure below. Our goal is to learn a latent encoding that is maximally useful for our target task, while trying to be close to a target distribution $r(Z)$.

Figure 2: Overview of a VIB that maps an input X to the target Y through a latent variable Z (top). We use deep neural networks for both the encoder and task-relevant "decoder."

(a) Assume that we decide to have the encoder network (parameterized by θ_e) take both an input x and some independent randomness V to emit a random sample Z in the latent space drawn according to the Gaussian distribution $p_{\theta_e}(Z|x)$.

For this part, assume that we want Z to be a scalar Gaussian (conditioned on x) with mean μ and variance σ^2 with the encoder neural network emitting the two scalars μ and σ as functions of x. Assume that V is drawn from iid standard $\mathcal{N}(0,1)$ Gaussian random variables.

Draw a block diagram with multipliers and adders showing how we get Z from μ and σ along with V.

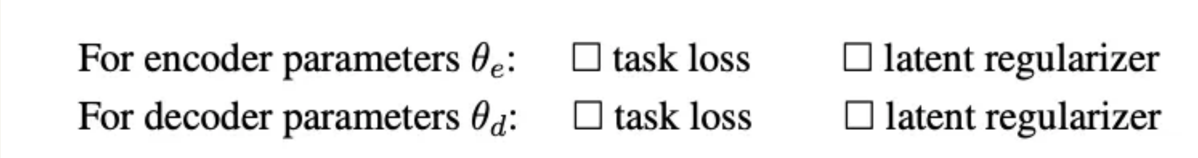
(b) Assume that our task is a classification-type task and the "decoder" network (parameterized by θ_d) emits scores for the different classes that we run through a softmax to get the distribution $q_{\theta_d}(y|z)$ over classes when the latent variable takes value z.

To train our networks using our N training points, we want to use SGD to approximately minimize the following loss:

$$L = \frac{1}{N}\sum_{n=1}^{N} \mathbb{E}_{z\sim p_{\theta_e}(z|x_n)}\left[\underbrace{-\log q_{\theta_d}(y_n|z)}_{\text{task loss}}\right] + \beta\,\overbrace{D_{KL}(p_{\theta_e}(Z|x_n)||r(Z))}^{\text{latent regularizer}} \tag{1}$$

where the y_n is the training label for input x_n and during training, we draw fresh randomness V each time we see an input, and we set $r(Z)$ to be a standard Gaussian $\mathcal{N}(0,1)$.

If we train using SGD treating the random samples of V as a part of the external input, **select all the loss terms that contribute (via backprop) to the gradients used to learn the encoder and decoder parameters**:



For encoder parameters θ_e: ☐ task loss ☐ latent regularizer
For decoder parameters θ_d: ☐ task loss ☐ latent regularizer

I have attached another question below: can you please solve it?

11/30/2025, 11:52:28 PM

```
Thought process: Analyzed variational information bottleneck architecture and gradient flow mechanisms.

Let me carefully analyze this Variational Information Bottleneck (VIB) question.

**Part (a)**: Draw a block diagram showing how we get Z from μ and σ along with V.

This is the reparameterization trick! We have:
- Encoder outputs μ(x) and σ(x)
- V ~ N(0, 1) (standard Gaussian)
- We want Z ~ N(μ, σ²)

Using the reparameterization trick:
Z = μ + σ * V

So the block diagram should show:
- μ and σ as inputs
- V as another input
- A multiplication of σ * V
- An addition of μ + (σ * V) to get Z

**Part (b)**: Which loss terms contribute to gradients for encoder and decoder parameters?

Let me analyze the loss function:
L = (1/N) Σ_{n=1}^N E_{z~p_{θe}(z|x_n)} [-log q_{θd}(y_n|z)] + β D_{KL}(p_{θe}(Z|x_n)||r(Z))

The loss has two terms:
1. Task loss: E_{z~p_{θe}(z|x_n)} [-log q_{θd}(y_n|z)]
```

2. Latent regularizer: β D_{KL}(p_{θe}(Z|x_n)||r(Z))

Now, the key question is: when we use SGD and treat V as part of the external input (not a learnable parameter), which terms contribute gradients via backprop?

For **encoder parameters θ_e**:
- The task loss involves sampling z from p_{θe}(z|x_n), which depends on θ_e
- Using the reparameterization trick: z = μ(x) + σ(x) * V, where μ and σ depend on θ_e

- Gradients flow through the reparameterization to the encoder
- The latent regularizer directly depends on encoder parameters θ_e (through μ and σ)

For **decoder parameters θ_d**:
- The task loss involves q_{θd}(y_n|z), which directly depends on θ_d
- The latent regularizer doesn't involve θ_d at all

I'll break down the gradient flow for each component. The encoder receives gradients from both the task loss and the KL divergence term, enabling end-to-end learning. The reparameterization trick allows smooth backpropagation through the stochastic layer by treating V as a fixed noise source.

For the encoder parameters θ_e, the gradient computation involves two key components: the task loss and the latent regularizer. Each term contributes to updating the encoder's parameters, ensuring that the learned representation captures both predictive and distributional information.

The decoder parameters θ_d are more focused, with the task loss

```
directly influencing their optimization while remaining
unaffected by the latent regularizer. This selective gradient
flow allows for precise control over the decoder's learning
process.
```

Looking at this Variational Information Bottleneck question:

Part (a): Block Diagram for Reparameterization Trick

Here's the block diagram showing how to get Z from μ, σ, and V:

```
     μ(x) ────────────────┐
                          │
                          ├─▶ (+) ───▶ Z
                          │
     σ(x) ───┐            │
             │            │
             ├─▶ (×) ─────┘
             │
     V ──────┘
```

Or more explicitly:

Z = μ + σ ⊙ V

Where:

- The encoder outputs **μ(x)** and **σ(x)** for input x
- **V ~ N(0, 1)** is sampled from standard Gaussian
- We **multiply** σ element-wise with V
- We **add** the result to μ to get Z

This part has been the worst explanation so far; Sonnet answers the question correctly, but describes this as a mere consequence of the “reparameterization trick”. It seems unlikely that a student who struggles to solve this question would be familiar with the reparameterization trick; hence, there is a need to explain HOW the trick allows us to propagate gradients rather than just citing it.

This is the reparameterization trick, which makes Z differentiable with respect to μ and σ (and thus θ_e), allowing gradients to flow through the stochastic sampling operation.

Part (b): Which Loss Terms Contribute to Gradients

Let me analyze each parameter set:

For encoder parameters θ_e:

- ☑ **task loss** - Gradients flow through the reparameterization: z = μ(x) + σ(x)·V depends on θ_e
- ☑ **latent regularizer** - D_KL(p_{θe}(Z|x_n)||r(Z)) directly depends on θ_e through μ and σ

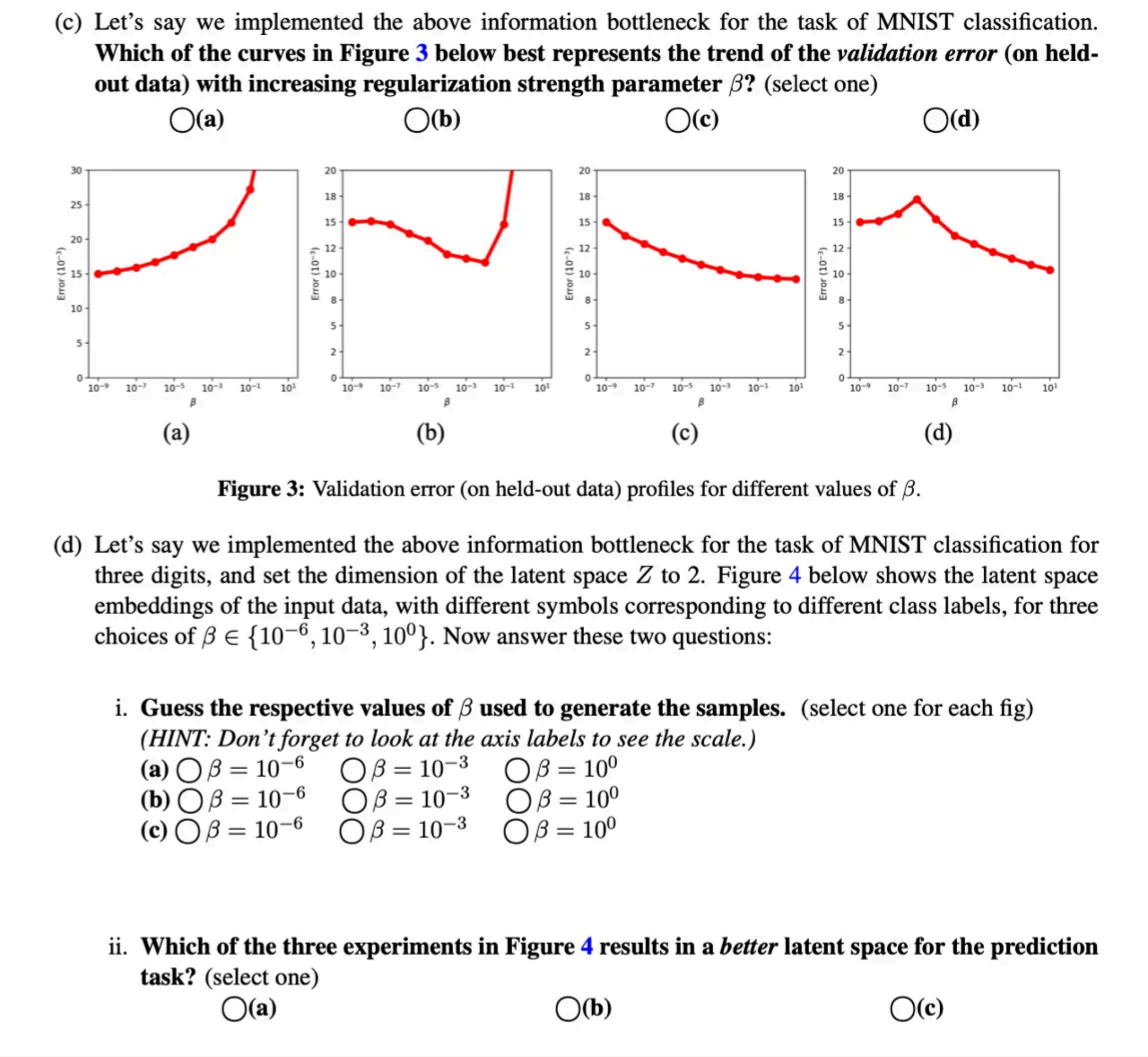
For decoder parameters θ_d:

- ☑ **task loss** - The term -log q_{θd}(y_n|z) directly depends on θ_d
- ☐ **latent regularizer** - D_KL term only involves the encoder distribution, not decoder parameters

Answer:

```
For encoder parameters θ_e:  ☑ task loss    ☑ latent regularizer
For decoder parameters θ_d:  ☑ task loss    ☐ latent regularizer
```

11/30/2025, 11:52:56 PM



(c) Let's say we implemented the above information bottleneck for the task of MNIST classification. **Which of the curves in Figure 3 below best represents the trend of the *validation error* (on held-out data) with increasing regularization strength parameter β?** (select one)

○**(a)** ○**(b)** ○**(c)** ○**(d)**

(a) (b) (c) (d)

Figure 3: Validation error (on held-out data) profiles for different values of β.

(d) Let's say we implemented the above information bottleneck for the task of MNIST classification for three digits, and set the dimension of the latent space Z to 2. Figure 4 below shows the latent space embeddings of the input data, with different symbols corresponding to different class labels, for three choices of $\beta \in \{10^{-6}, 10^{-3}, 10^{0}\}$. Now answer these two questions:

i. **Guess the respective values of β used to generate the samples.** (select one for each fig) *(HINT: Don't forget to look at the axis labels to see the scale.)*

(a) ○ $\beta = 10^{-6}$ ○ $\beta = 10^{-3}$ ○ $\beta = 10^{0}$
(b) ○ $\beta = 10^{-6}$ ○ $\beta = 10^{-3}$ ○ $\beta = 10^{0}$
(c) ○ $\beta = 10^{-6}$ ○ $\beta = 10^{-3}$ ○ $\beta = 10^{0}$

ii. **Which of the three experiments in Figure 4 results in a *better* latent space for the prediction task?** (select one)

○**(a)** ○**(b)** ○**(c)**

Can you please solve the remaining parts of the problem in the image I attached?

11/30/2025, 11:55:20 PM

```
Thought process: Analyzed regularization effects on validation
error and latent space structure.

Let me analyze these questions carefully.

## Part (c): Validation error trend with increasing β
```

```
The question asks which curve best represents the trend of validation error as β (regularization strength) increases.

Let me think about what β does:
- β controls the strength of the KL divergence regularizer
- Higher β means we force the latent distribution p(Z|x) to be closer to r(Z) = N(0,1)
- This acts as a bottleneck, compressing information

Expected behavior:
- Very small β (10^-8): Almost no regularization, model can overfit, may have high validation error
- Moderate β: Good balance, reduces overfitting, validation error should be low
- Very large β (10^0 = 1): Too much regularization, destroys task-relevant information, underfits, high validation error

This is a classic U-shaped curve or bias-variance tradeoff:
- Too small β: overfitting (high variance)
- Too large β: underfitting (high bias)
- Optimal β in the middle

Looking at the curves:
- (a): Monotonically increasing - doesn't show the U-shape
- (b): U-shaped with minimum in the middle, then shoots up - this matches!
- (c): Monotonically decreasing - doesn't make sense
- (d): Inverted U-shape (decreasing then increasing) - this is backwards

Answer: **(b)** - shows the U-shaped curve with best performance at moderate β

## Part (d): Analyzing latent space embeddings
```

Sonnet has an interesting multi-step reasoning approach to this problem. First it tries to think about general deep learning principles to understand what beta does at a high-level; then it dives more specifically into the specific data provided to apply this thinking to the task at hand.

Part (d)i: Guess the β values for each figure

The hint says to look at the axis labels to see the scale. The question is asking about β ∈ {10^-6, 10^-3, 10^0}.

Key insight:
- Low β (10^-6): Weak regularization, latent space can spread out widely, less constrained to N(0,1)

- Medium β (10^-3): Moderate regularization, some structure emerges
- High β (10^0 = 1): Strong regularization, forces latent space into tight N(0,1) distribution

I'll map the visual characteristics to β values:
- Widest spread suggests weakest regularization (10^-6)
- Moderate clustering indicates intermediate regularization (10^-3)
- Tightest, most compact clustering represents strongest regularization (10^0)

The progression shows how increasing β progressively constrains the latent space representation.

I notice the options present a puzzle. Each option set includes the same β values, but requires selecting one per figure. This suggests a nuanced matching between β values and specific figure characteristics. The challenge is determining which β value best describes each individual figure's distribution.

The key is carefully examining how the latent space transforms with different regularization strengths, matching the visual spread to the corresponding β parameter.

```
I'll need to analyze the spread of each figure systematically,
connecting the visual clustering pattern to the appropriate
regularization intensity.

The β values represent different regularization strengths: 10^-6
allows maximum spread, 10^-3 provides moderate constraint, and
10^0 creates tight clustering. Each subfigure (a), (b), (c)
requires selecting one β value.

Examining the latent space quality involves three key criteria:
class separation, within-class compactness, and preserving task-
relevant information. The β = 10^-3 appears most promising -
offering a balanced approach between spreading out and over-
constraining the representation.

The progression suggests a nuanced trade-off: extremely weak
regularization risks excessive dispersion, while strong
regularization risks collapsing distinctions. I'm anticipating
the middle value will demonstrate the most effective
representation.
```

Looking at the remaining parts of this Variational Information Bottleneck problem:

Part (c): Validation Error Trend with Increasing β

Answer: (b)

Reasoning: As β increases, we get a classic bias-variance tradeoff (U-shaped curve):

This is a concise, but still thorough explanation to the question.

- **Small β (10^{-8})**: Weak regularization → model can overfit to training data → poor generalization → **high validation error**
- **Moderate β (10^{-4} to 10^{-2})**: Good balance → bottleneck removes noise but preserves task-relevant information → **low validation error**

- **Large β ($10^0 = 1$)**: Too strong regularization → forces Z too close to N(0,1) → destroys task-relevant information → underfitting → **high validation error**

Curve (b) shows this U-shaped pattern with a minimum at moderate β values, which is the expected behavior.

Part (d)i: Guess the β Values for Each Figure

Answer:

- **(a) $\beta = 10^{-6}$** (weakest regularization → largest spread)
- **(b) $\beta = 10^{-3}$** (moderate regularization → medium spread)
- **(c) $\beta = 10^0$** (strongest regularization → tightest clustering near origin)

Reasoning:

- **Low β (10^{-6})**: Weak KL penalty → latent space can spread widely → large scale on axes
- **Medium β (10^{-3})**: Balanced compression → moderate spread with some structure
- **High β (10^0)**: Strong penalty forces $p(Z|x) \approx N(0,1)$ → tight clustering around origin → small scale

Part (d)ii: Which Experiment Gives a Better Latent Space?

Answer: (b) (corresponding to $\beta = 10^{-3}$)

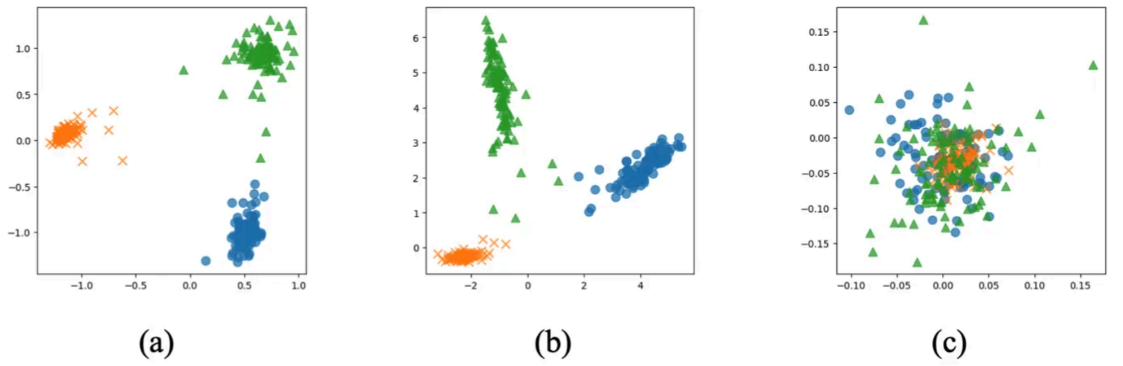
Reasoning: From part (c), we know the optimal β is at the minimum of the U-curve (moderate values). This corresponds to $\beta = 10^{-3}$, which:

- Provides good class separation (task-relevant information preserved)
- Maintains compact, structured representations (some regularization)
- Achieves the best validation performance

The extreme values are suboptimal:

- $\beta = 10^{-6}$: Too dispersed, potential overfitting
- $\beta = 10^{0}$: Over-compressed, poor class separation

11/30/2025, 11:55:58 PM



(a) (b) (c)

Figure 4: MNIST VIB with 2D latent space.

For part (d), here is the Figure being referenced; given this, answer the question

11/30/2025, 11:58:32 PM

```
Thought process: Analyzed latent space visualizations and matched
regularization parameters to plots.

Now I can see Figure 4 with the actual latent space
visualizations. Let me analyze each plot:

**Plot (a):**
- Axis range: approximately -1.5 to 1.5
- Three classes are well-separated into distinct clusters
- Orange X's cluster around (-1, 0)
- Blue circles cluster around (0.5, -1)
- Green triangles cluster around (0.5, 1)
- Clear separation between classes
- Moderate spread

**Plot (b):**
- Axis range: approximately -3 to 7
- Much larger spread/scale
```

- Classes are more spread out
- Orange X's around (-2, 0)
- Blue circles spread from (2, 1.5) to (5, 3)
- Green triangles very spread from (0, 2) to (2, 6)
- Large variance within each class

Plot (c):
- Axis range: approximately -0.15 to 0.15
- Very small scale, tightly clustered
- All three classes heavily overlap near the origin
- Poor class separation
- Points are compressed together

Now, relating to β values:
- **Low β (10^{-6})**: Weak regularization → encoder can use large values → large spread → **Plot (b)**
- **Medium β (10^{-3})**: Balanced regularization → good separation with reasonable scale → **Plot (a)**
- **High β (10^{0})**: Strong regularization forces Z close to N(0,1) → tight clustering near origin → **Plot (c)**

So my revised answer:
- (a) $\beta = 10^{-3}$
- (b) $\beta = 10^{-6}$
- (c) $\beta = 10^{0}$

For part ii, which is better:
- Plot (a) shows the best class separation while maintaining structure
- Plot (b) is too spread out (overfitting risk)

- Plot (c) has poor class separation (underfitting)

I'm carefully reconsidering the labeling for part (d)i. The

```
challenge is ensuring I match the correct β values to the correct
plots while avoiding confusion between the plot labels and the
experiment labels.

For part (d)i, the β values are:
- (a) β = 10⁻³ shows good separation with moderate scale
- (b) β = 10⁻⁶ demonstrates a large spread
- (c) β = 10⁰ reveals tight clustering

For part (d)ii, plot (a) emerges as the most effective latent
space representation.
```

Now that I can see Figure 4, let me revise my answers based on the actual visualizations:

Sonnet does a good job here of simply transferring the reasoning and ideas it came up with in part (c) to this specific plot.

Part (d)i: Guess the β Values for Each Figure

Looking at the scales and clustering patterns:

Answer:

- **(a) $\beta = 10^{-3}$** (moderate regularization)
- **(b) $\beta = 10^{-6}$** (weakest regularization)
- **(c) $\beta = 10^{0}$** (strongest regularization)

Reasoning:

Plot (a):

- Axis scale ~[-1.5, 1.5]
- Three classes are well-separated with clear structure
- Moderate spread with good clustering
- → **$\beta = 10^{-3}$** (optimal balance)

Plot (b):

- Axis scale ~[-3, 7] - much larger!
- Classes are very spread out with high variance
- Weak constraint on latent space
- → **$\beta = 10^{-6}$** (minimal regularization, allows large deviations from prior)

Plot (c):

- Axis scale ~[-0.15, 0.15] - very small!
- All classes heavily overlap near origin
- Strong compression toward N(0,1)
- Poor class separation
- → **$\beta = 10^{0}$** (strong regularization forces tight clustering)

Part (d)ii: Which Experiment Gives a Better Latent Space?

Answer: (a)

Reasoning:

- **(a)** shows excellent class separation with three distinct, well-defined clusters while maintaining reasonable structure - this is ideal for downstream classification
- **(b)** is too dispersed with excessive variance, indicating potential overfitting and poor regularization
- **(c)** has severe class overlap due to over-compression, making it difficult to distinguish between classes

Plot (a) with $\beta = 10^{-3}$ achieves the best balance between preserving task-relevant information (good separation) and regularization (structured representations).

11/30/2025, 11:58:55 PM