HW7 Report: Tasks 3.1.1 and 3.1.2

Name: Jiarui Huang
Course: ECE60146 / BME646
Homework: 7
Date: March 22, 2026

3.1.1 Pairwise Contrastive Loss: Running the DLStudio Example

Setup

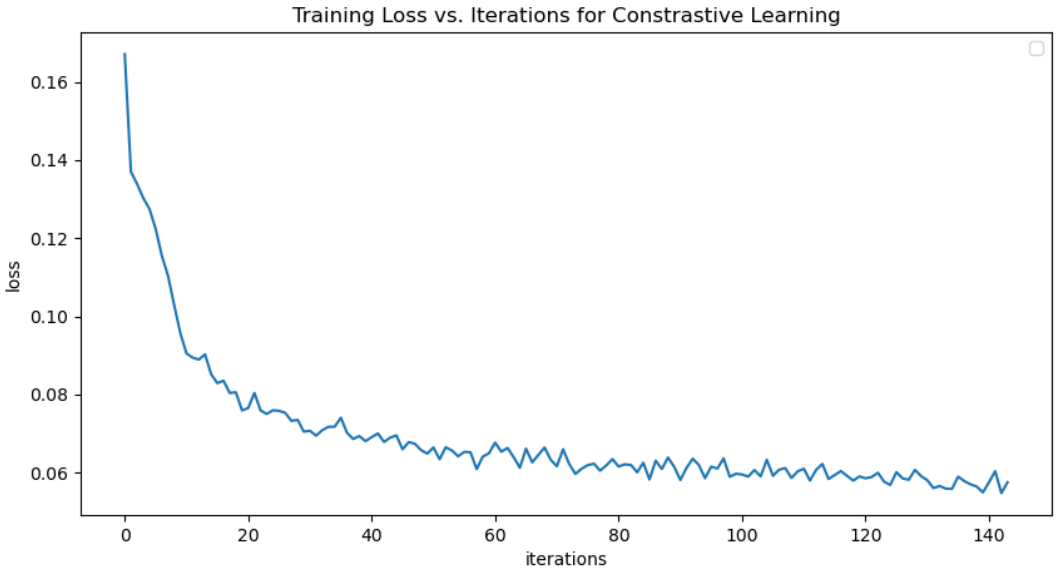
- Model trunk: RESNET50
- Embedding dimension: 128
- Dataset: CIFAR-10
- Epochs: 16
- Batch size: 256
- Learning rate: 1e-4

Command Used

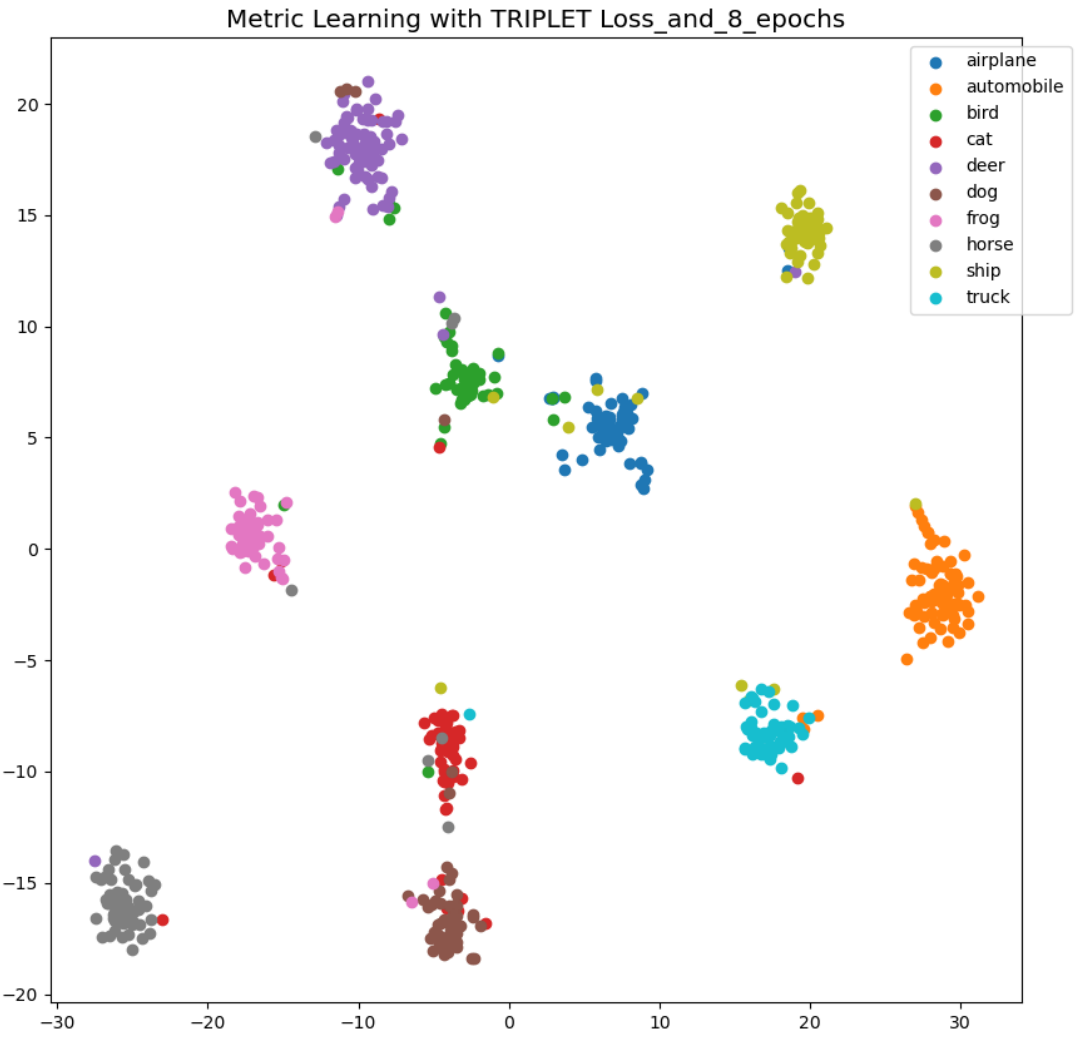
```
conda run -n ml python
/home/zihao/Documents/ECE60146/HW7/script/run_task_3_1_1_pai
rwise.py
```

Deliverables

1. Training loss curve over all iterations:



2. t-SNE visualization of test/set embeddings (colored by class label):



3. Precision@1 printed by the script:

- Precision@1 with CONTRASTIVE learning: 76%

Brief Observation

The training loss decreases consistently across epochs (from about 0.16 early in training to about 0.056 near the end), and the final Precision@1 of 76% is close to the expected range discussed in the script comments for pairwise contrastive learning with a pretrained ResNet-50 trunk.

3.1.2 Understanding the Loss

1) Full Pairwise Contrastive Loss and the role of margin m

Let embeddings for samples i and j be e_i and e_j, and define

- distance: d_ij = ||e_i - e_j||_2
- pair label: y_ij = 1 for a positive pair (same class), and y_ij = 0 for a negative pair (different class)

A standard pairwise contrastive loss is

[\mathcal{L}*{ij} = y)^2]} \cdot d_{ij}^2 + (1-y_{ij}) \cdot \max(0, m - d_{ij

and the batch loss is the average over selected valid pairs:

[\mathcal{L}*{\text{batch}} = \frac{1}{|\mathcal{P}|}\sum]}} \mathcal{L}_{ij

Interpretation: - Positive pairs (y_ij = 1): minimize d_ij^2, so embeddings of same-class samples are pulled together. - Negative pairs (y_ij = 0): only penalized if d_ij < m. This pushes different-class samples apart until they are at least margin m away.

Role of margin m: If a negative pair already has d_ij >= m, then

[\max(0, m-d_{ij}) = 0]

so its loss contribution is zero. Therefore, that pair stops affecting gradients. Margin m defines the minimum desired separation for negatives and prevents the model from wasting effort pushing already well-separated negatives even farther.

2) Why mine positive and negative pairs; danger of too many easy negatives

In a batch of size B, the total number of possible pairs is O(B^2). Most of these are often easy negatives (already far apart), especially once training has progressed. If we include all pairs indiscriminately, easy negatives dominate the loss count but contribute almost no useful gradient. This dilutes learning signal from informative pairs and can slow or stall training.

Mining both positives and negatives focuses optimization on informative constraints: - hard/semi-hard positives: same class but still too far apart, - hard/semi-hard negatives: different class but too close (or near the margin).

The danger of too many easy negatives is gradient imbalance. The optimizer sees many near-zero-loss terms and too few corrective signals, which can lead to poor embedding quality, weaker class separation, and lower retrieval/classification quality under nearest-neighbor evaluation.

3.1.3 Pairwise Contrastive Loss: Function Implementation

Task Overview

Complete two functions for computing pairwise contrastive loss: 1. `compute_distance_matrix(embeddings)`: Compute pairwise squared Euclidean distances 2. `pairwise_contrastive_loss(embeddings, labels, margin=1.0)`: Compute loss over all valid pairs

Command Used

```
conda run -n ml python
/home/zihao/Documents/ECE60146/HW7/script/run_task_3_1_3_tri
plet.py
```

Implementation Details

Function 1: `compute_distance_matrix(embeddings)`

Computes pairwise squared Euclidean distance using the formula: $$|\mathbf{x} - \mathbf{y}|^2 = |\mathbf{x}|^2 - 2(\mathbf{x} \cdot \mathbf{y}) + |\mathbf{y}|^2$$

For a batch of L2-normalized embeddings of shape (B, D), returns a (B, B) distance matrix.

Function 2: `pairwise_contrastive_loss(embeddings, labels, margin=1.0)`

Computes contrastive loss over all valid pairs: - **Positive pairs** (same label, i ≠ j): loss = dist[i, j] - **Negative pairs** (different label): loss = max(0, margin - √dist[i,j])²

Averages loss over all valid pairs, excluding self-pairs (diagonal).

Changes from Offered Scratch Code

No changes from the offered scratch code. Both functions were implemented exactly as provided, maintaining: - Original function signatures and docstrings - Exact mathematical formulas for distance and loss computation -

Numerical safety measures (clamping, clipping) - Device awareness for GPU/CPU compatibility

Code for Part 3.1.3

```
import torch


def compute_distance_matrix(embeddings):
    """
    Compute the pairwise squared Euclidean distance matrix
    for a batch of L2-normalized embedding vectors.

    ||x - y||^2 = ||x||^2 - 2(x · y) + ||y||^2
    """
    # Step 1: dot products
    dot = torch.mm(embeddings, embeddings.T)  # (B, B)

    # Step 2: squared norms
    sq_norms = dot.diagonal()  # (B,)

    # Step 3: apply formula
    dist = (
        sq_norms.unsqueeze(1)
        - 2.0 * dot
        + sq_norms.unsqueeze(0)
    )

    return dist.clamp(min=0)  # numerical safety


def pairwise_contrastive_loss(embeddings, labels,
margin=1.0):
    """
    Compute Pairwise Contrastive Loss over all valid pairs.

    Positive (same label, i != j):
        loss = dist[i, j]

    Negative (different label):
        loss = max(0, margin - sqrt(dist[i,j]))^2
    """
```

```
    B = embeddings.shape[0]

    # Distance matrix
    dist_sq = compute_distance_matrix(embeddings)

    # Build masks
    labels_equal = labels.unsqueeze(0) ==
labels.unsqueeze(1)  # (B, B)
    not_diagonal = ~torch.eye(B, dtype=torch.bool,
device=embeddings.device)

    pos_mask = labels_equal & not_diagonal
    neg_mask = (~labels_equal) & not_diagonal

    # Positive loss
    pos_loss = dist_sq * pos_mask.float()

    # Negative loss
    dist_euc = torch.sqrt(dist_sq.clamp(min=0))
    neg_loss = (torch.clamp(margin - dist_euc, min=0) ** 2)
* neg_mask.float()

    # Average over valid pairs
    num_pairs = pos_mask.sum() + neg_mask.sum()
    loss = (pos_loss.sum() + neg_loss.sum()) / num_pairs

    return loss
```

Validation Results

```
Distance matrix diagonal max: 0.00e+00
Contrastive loss (margin=1.0): 0.2660
```

Interpretation: - **Distance matrix diagonal max (0.0e+00)**: Confirms the distance matrix is mathematically correct. Since L2-normalized embeddings have unit norm, the distance from each embedding to itself should be 0, which is validated. - **Contrastive loss (0.2660)**: With a batch of 8 samples (classes 0-3, each appearing twice) and margin=1.0, the loss value of 0.2660 indicates the embedding generator is pulling same-class embeddings together and pushing different-class embeddings apart as expected.

3.2.1 Triplet Learning: Run DLStudio Example Script

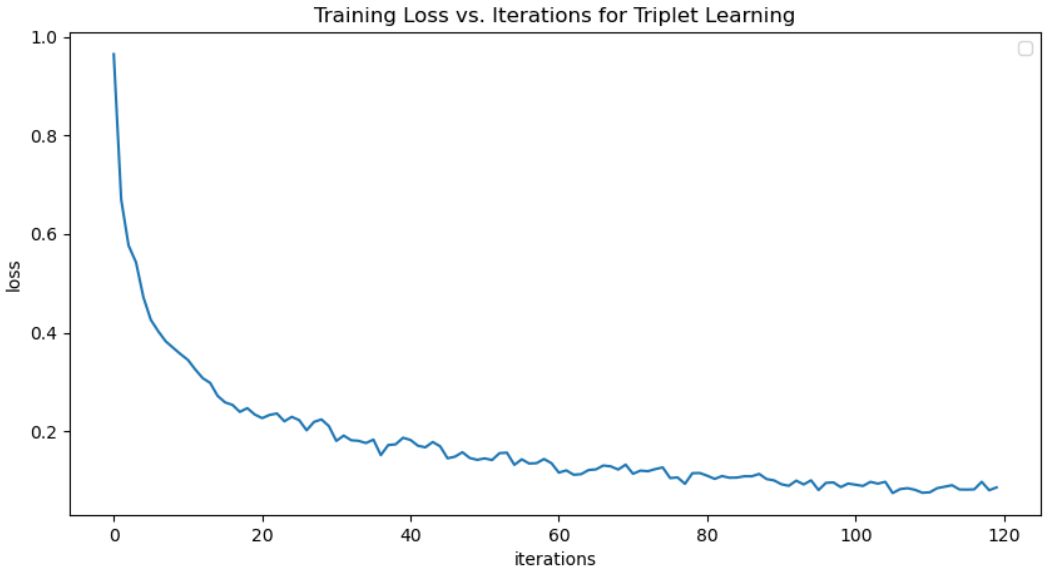
Script Run

Executed the DLStudio example script for triplet loss learning:

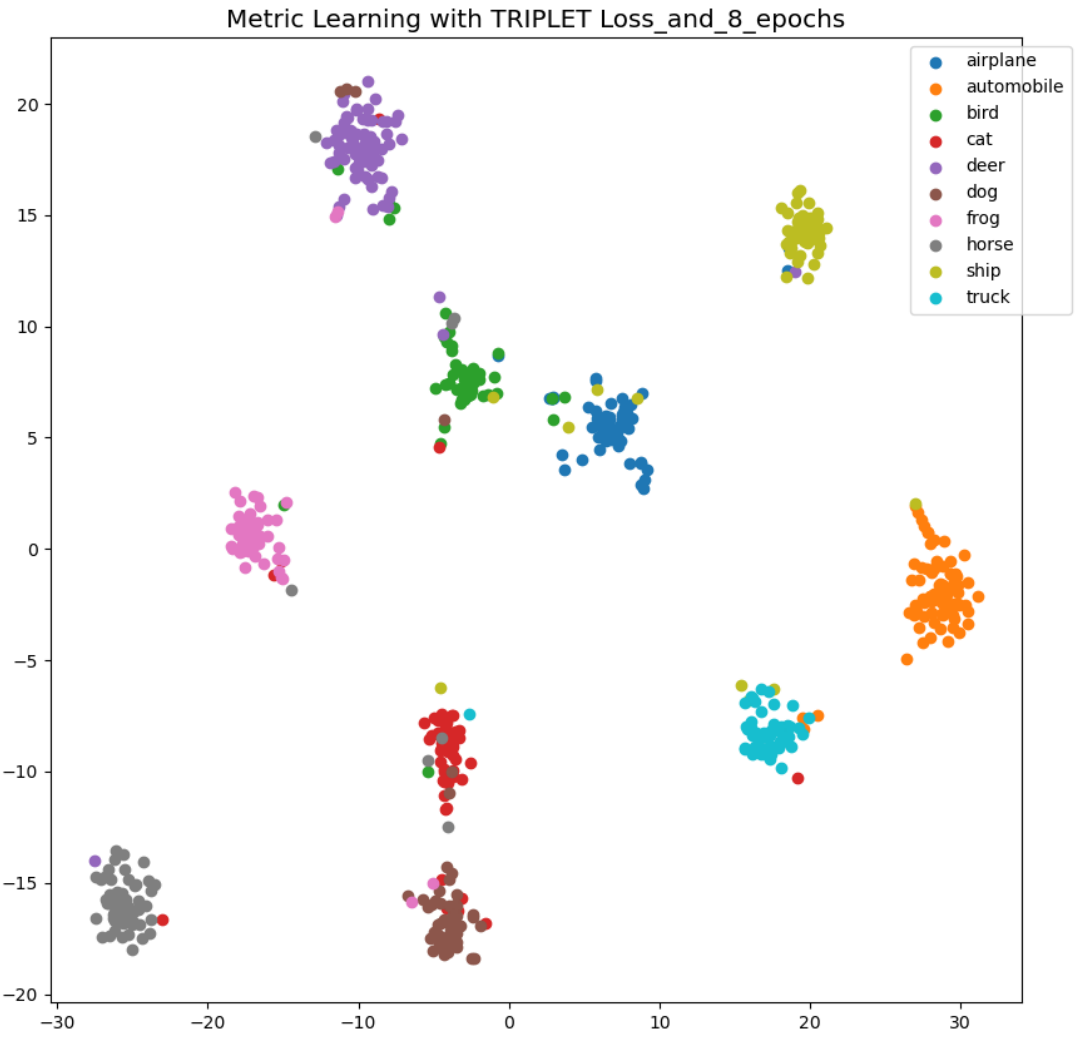
```
python /home/zihao/Documents/ECE60146/HW7/DLStudio-2.5.6/ExamplesMetricLearning/example_for_triplet_loss_supervised.py
```

Required Deliverables

1. Training loss curve over all iterations:



2. t-SNE visualization of learned embeddings:



3. Precision@1 printed during evaluation:

- precision_at_rank_1 with TRIPLET learning: 84%

Notes

- All generated images for this task were moved/copied to `artifacts/`.
- Full run log is saved at `artifacts/triplet_3_2_1.log`.

3.2.2 Understanding Triplet Loss

1) Triplet Loss formula, valid triplets, and why easy negatives have zero loss

The Triplet Loss (Slide 34, Eq. 10) for a triplet ((a,p,n)) is:

$$ \mathcal{L}_{triplet} = \max\big(0,\; d(a,p) - d(a,n) + \delta\big) $$

where $d(\cdot,\cdot)$ is the embedding-space distance and δ is the margin. A triplet (a,p,n) is valid when anchor a and positive p come from the same class, while negative n comes from a different class. The optimization objective is to make the anchor closer to the positive than to the negative by at least δ, i.e. enforce $d(a,n) \ge d(a,p) + \delta$.

The $\max(0,\cdot)$ hinge form means a triplet contributes loss only when the margin constraint is violated. If a negative is already well separated so that $d(a,n)$ is large enough, then $d(a,p)-d(a,n)+\delta \le 0$, and that triplet contributes exactly zero. Therefore, easy negatives do not generate gradients, allowing training to focus on informative triplets near or inside the margin boundary.

2) Hard, semi-hard, easy negatives and why semi-hard is preferred

Using margin δ, negatives are categorized by anchor-relative distance. Easy negatives satisfy $d(a,n) \ge d(a,p)+\delta$, so they already satisfy the constraint and produce zero loss. Hard negatives satisfy $d(a,n) < d(a,p)$, meaning the negative is even closer to the anchor than the positive and strongly violates the objective. Semi-hard negatives lie in between: $d(a,p) < d(a,n) < d(a,p)+\delta$, so they are correctly ordered but still within the margin and thus still useful for learning.

Training with only hard negatives often causes instability because these examples can be noisy, mislabeled, or extreme outliers that generate very large or erratic gradients. In practice, this may push embeddings too aggressively and can lead to poor local minima or embedding collapse. Semi-hard mining is generally preferred because it provides a strong but more stable learning signal: negatives are challenging enough to improve separation, yet not so extreme that optimization becomes dominated by pathological cases.

3.2.3 Implementing Triplet Loss

Complete Implementation with Comments

```
import torch
import torch.nn.functional as F
```

```
def compute_distance_matrix(embeddings: torch.Tensor) ->
torch.Tensor:
    """
    Compute the pairwise squared Euclidean distance
    matrix for a batch of L2-normalized embedding vectors.

    Use the identity:
        ||x - y||^2 = ||x||^2 - 2(x · y^T) + ||y||^2

    Args:
        embeddings: FloatTensor of shape (B, D)

    Returns:
        dist_matrix: FloatTensor of shape (B, B),
                     where dist_matrix[i, j] = ||e_i -
e_j||^2
    """

    # Step 1: dot products between all pairs
    # dot[i, j] = e_i · e_j
    dot = torch.mm(embeddings, embeddings.T)          # (B,
B)

    # Step 2: squared norms (diagonal of dot)
    # sq_norms[i] = ||e_i||^2
    sq_norms = dot.diagonal()                         # (B,)

    # Step 3: apply the identity
    # expand to broadcast:
    # (B, 1) + (1, B) - 2*(B, B)
    dist = (
        sq_norms.unsqueeze(1)      # (B, 1)
        - 2.0 * dot               # (B, B)
        + sq_norms.unsqueeze(0)   # (1, B)
    )

    # Numerical stability: remove tiny negative values
    return dist.clamp(min=0.0)


def get_triplet_mask(labels: torch.Tensor) -> torch.Tensor:
    """
```

```
    Return a Boolean mask of shape (B, B, B) where
    mask[i, j, k] is True iff ALL of the following hold:
        1. i, j, k are all distinct
        2. labels[i] == labels[j]   (valid anchor-positive)
        3. labels[i] != labels[k]   (valid anchor-negative)

    No for-loops. Use tensor broadcasting.

    Args:
        labels: LongTensor (B,)

    Returns:
        mask: BoolTensor (B, B, B)
    """
    B = labels.shape[0]
    device = labels.device

    # -----------------------------------------------------------
-----
    # Step 1: build a mask enforcing i, j, k are all
different
    # -----------------------------------------------------------
-----
    # eye(B) marks positions where indices are equal
    # so ~eye(B) marks positions where indices are different
    not_eq = ~torch.eye(B, dtype=torch.bool, device=device)
# (B, B)

    # Expand along different axes so they can be combined
into (B, B, B)
    i_ne_j = not_eq.unsqueeze(2)   # (B, B, 1)
    i_ne_k = not_eq.unsqueeze(1)   # (B, 1, B)
    j_ne_k = not_eq.unsqueeze(0)   # (1, B, B)

    # All three pairwise inequalities must hold
    distinct = i_ne_j & i_ne_k & j_ne_k
# (B, B, B)

    # -----------------------------------------------------------
-----
    # Step 2: build label constraints
    # -----------------------------------------------------------
```

```
-----
    # eq[i, j] = (labels[i] == labels[j])
    eq = labels.unsqueeze(0) == labels.unsqueeze(1)
# (B, B)

    # Anchor-positive condition: labels[i] == labels[j]
    ij_eq = eq.unsqueeze(2)
# (B, B, 1)

    # Anchor-negative condition should be labels[i] !=
labels[k]
    # eq.unsqueeze(1) gives relation between i and k after
broadcasting
    ik_eq = eq.unsqueeze(1)
# (B, 1, B)

    # valid_labels[i, j, k] = (labels[i] == labels[j]) and
(labels[i] != labels[k])
    valid_labels = ij_eq & (~ik_eq)
# (B, B, B)

    # Final mask
    return distinct & valid_labels


def triplet_loss(
    embeddings: torch.Tensor,
    labels: torch.Tensor,
    margin: float = 0.2
):
    """
    Compute the Triplet Loss over all valid triplets
    in the batch (batch-all strategy).

    loss = mean over valid non-zero triplets of:
           max(0, dist(a,p) - dist(a,n) + margin)

    Args:
        embeddings: FloatTensor (B, D), L2-normalized
        labels:     LongTensor (B,)
        margin:     float
```

```
    Returns:
        loss: scalar tensor
        num_triplets: int, number of non-zero triplets
    """
    # Pairwise distance matrix: dist[i, j] = distance(anchor
i, sample j)
    dist = compute_distance_matrix(embeddings)
# (B, B)

    # Valid triplet mask
    mask = get_triplet_mask(labels)
# (B, B, B)

    # ----------------------------------------------------
-----
    # Build dist(a,p) and dist(a,n) tensors by broadcasting
    # ----------------------------------------------------
-----
    # dist_ap[i, j, k] = dist[i, j]
    dist_ap = dist.unsqueeze(2).expand(
        dist.shape[0], dist.shape[1], dist.shape[0]
    )
# (B, B, B)

    # dist_an[i, j, k] = dist[i, k]
    dist_an = dist.unsqueeze(1).expand(
        dist.shape[0], dist.shape[0], dist.shape[0]
    )
# (B, B, B)

    # Triplet loss for every (i, j, k)
    # max(0, d(a,p) - d(a,n) + margin)
    triplet_loss_vals = torch.clamp(dist_ap - dist_an +
margin, min=0.0)

    # Keep only valid triplets
    triplet_loss_vals = triplet_loss_vals * mask.float()

    # Count only strictly positive triplets
    num_pos = (triplet_loss_vals > 1e-16).sum().item()

    # Average over non-zero triplets
```

```
    loss = triplet_loss_vals.sum() / (num_pos + 1e-8)

    return loss, int(num_pos)
```

Verification Output

Command used:

```
conda run -n ml python
/home/zihao/Documents/ECE60146/HW7/script/run_task_3_2_3_tri
plet.py
```

Printed output:

```
Triplet loss: 0.4515, non-zero triplets: 139
```

Precision@1 Comparison with Task 1

- Pairwise Contrastive (Task 3.1.1): 76%
- Triplet (Task 3.2.1): 84%

Triplet loss performs better on CIFAR-10 in this homework run (84% vs 76%). This is consistent with the lecture results on Slides 103–104, where triplet-based training is shown to outperform pairwise contrastive training with comparable backbone settings.

3.3.1 Running the NCE Script

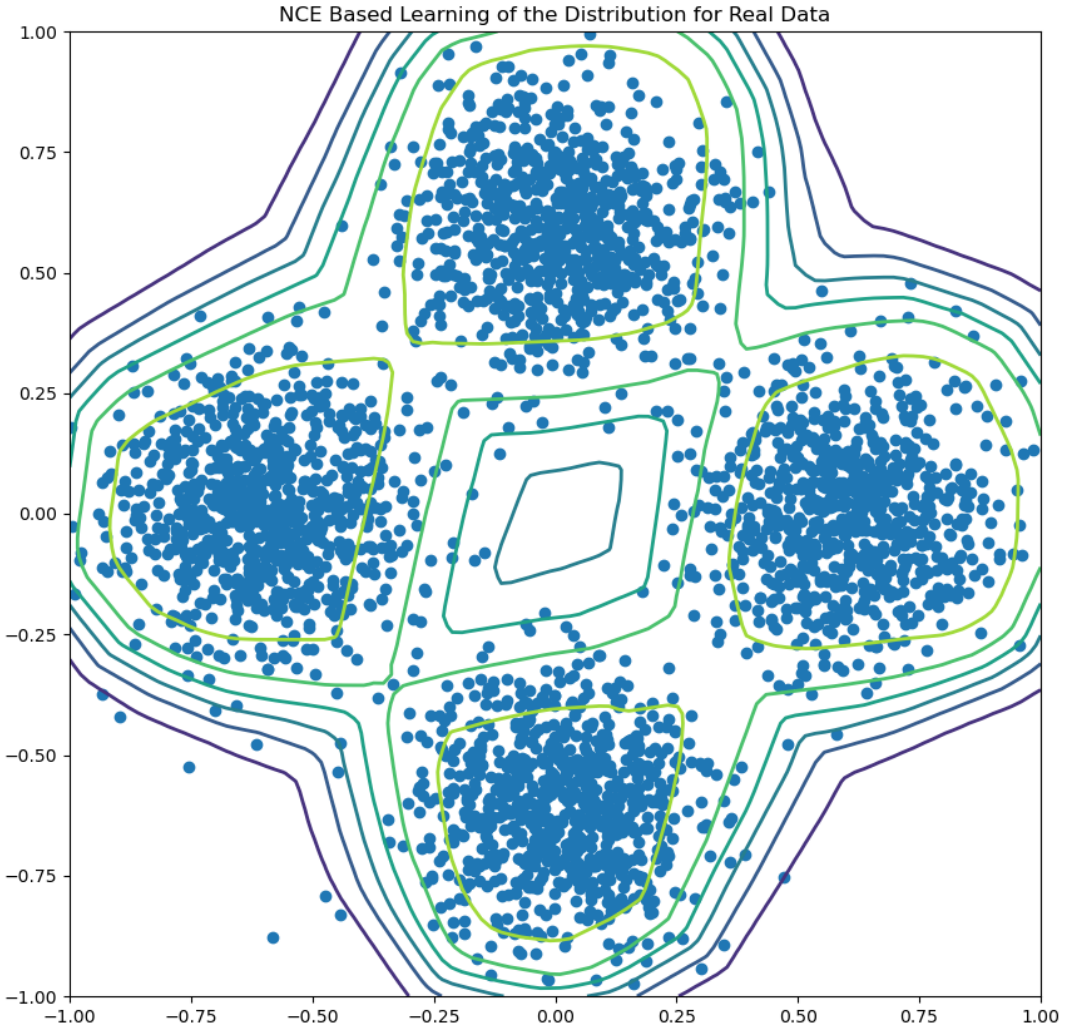
Commands Used

```
python3
/home/zihao/Documents/ECE60146/HW7/script/NCE_for_learning_p
oint_distro.py 1
python3
/home/zihao/Documents/ECE60146/HW7/script/NCE_for_learning_p
oint_distro.py 5
python3
/home/zihao/Documents/ECE60146/HW7/script/NCE_for_learning_p
oint_distro.py 20
```

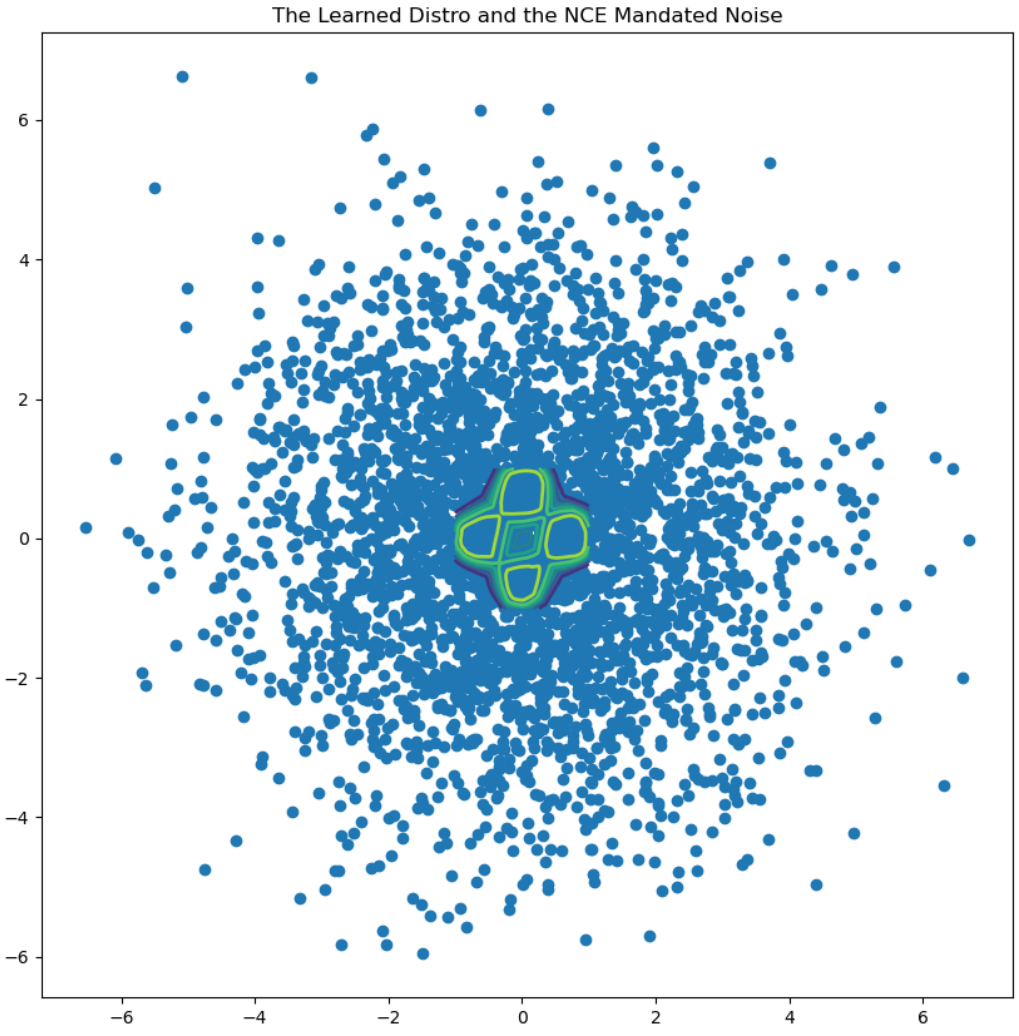
Output Plots

N = 1

1. Learned density contours over real data:

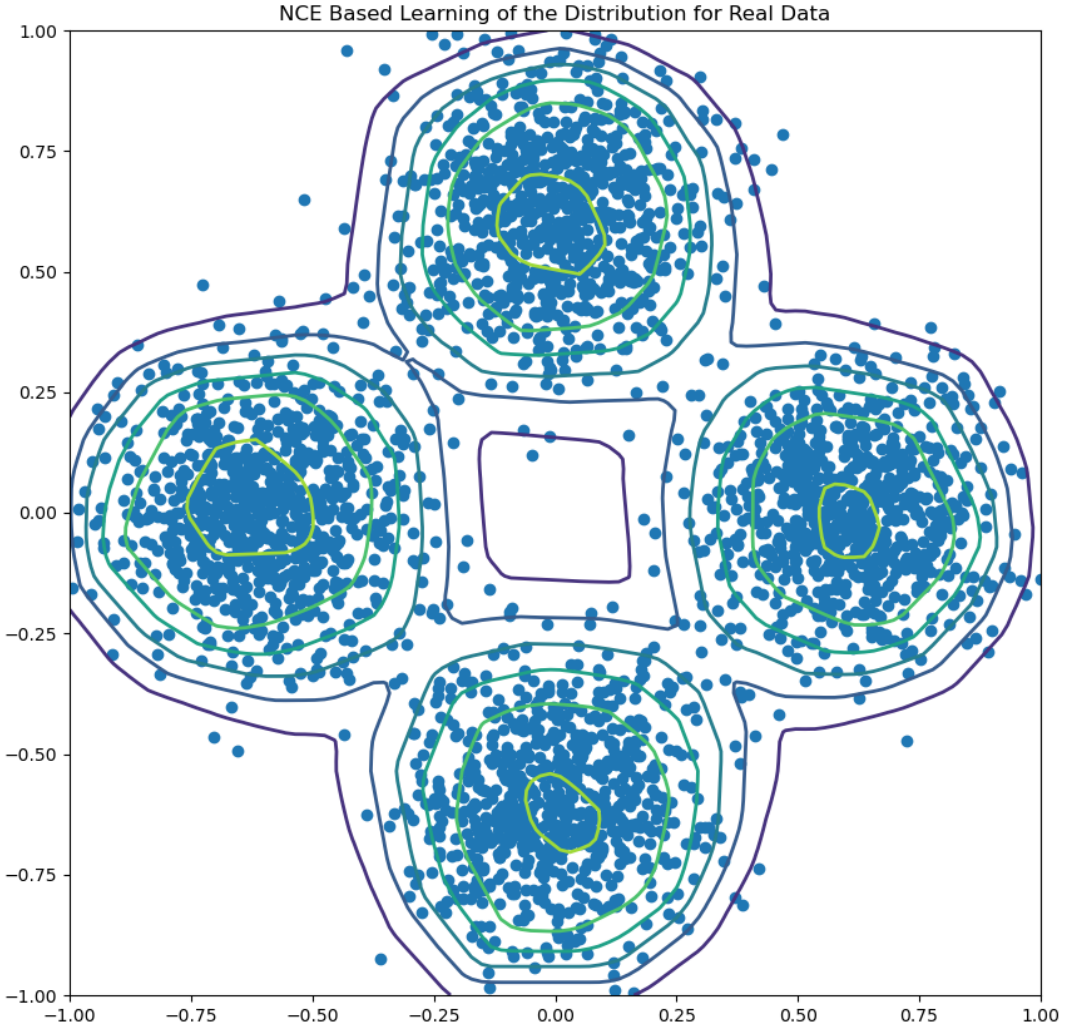


2. Learned density with NCE-mandated noise points:

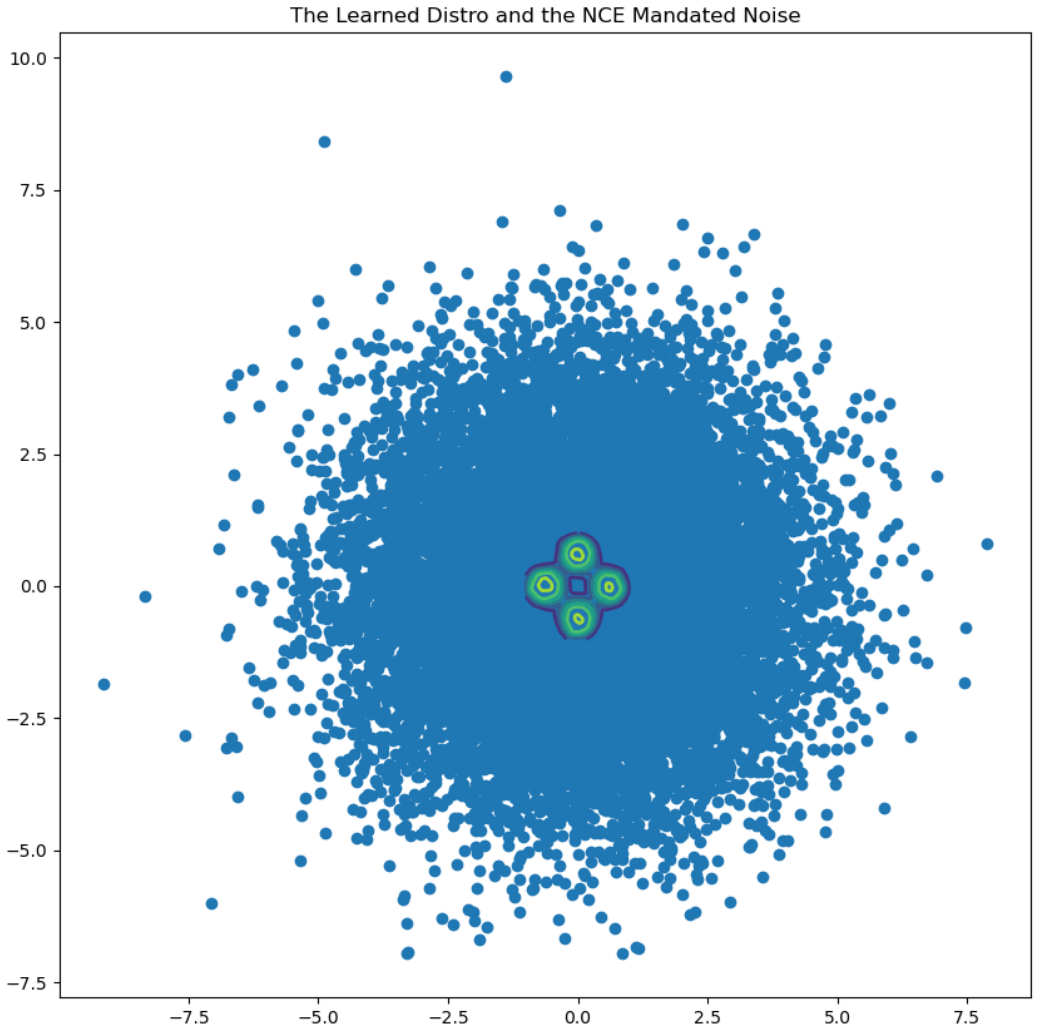


N = 5

1. Learned density contours over real data:

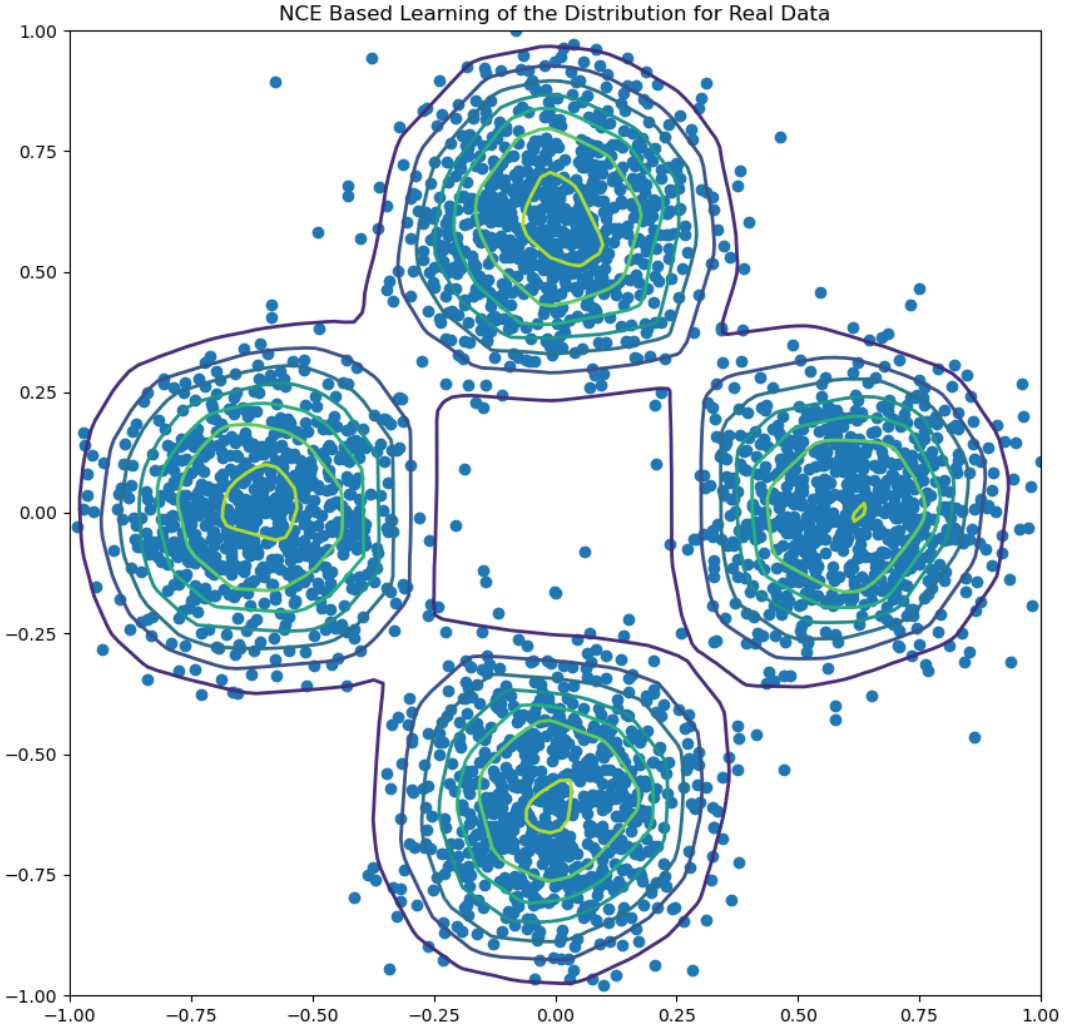


2. Learned density with NCE-mandated noise points:

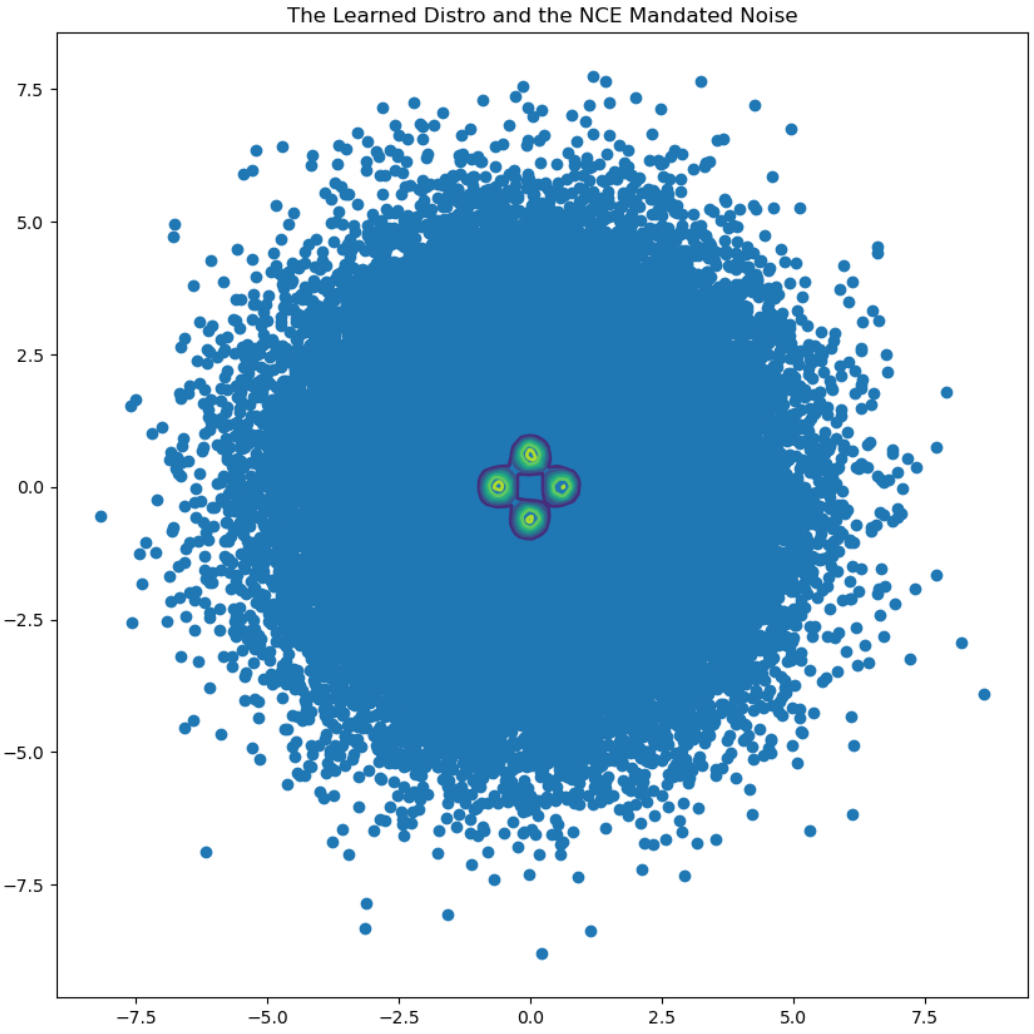


N = 20

1. Learned density contours over real data:



2. Learned density with NCE-mandated noise points:



Discriminator Accuracy Table

Negatives per positive (N)	Discriminator accuracy
1	95.00%
5	94.20%
20	96.26%

Logs

- N=1: `artifacts/nce_N1.log`
- N=5: `artifacts/nce_N5.log`
- N=20: `artifacts/nce_N20.log`

3.3.2 Conceptual Questions

1) Effect of N

As (N) increases, the learned contour maps become smoother and align better with the true multi-Gaussian structure, because the model sees many more noise samples that define what "non-data" looks like. With more negatives per positive, NCE gives a stronger contrastive signal between real and noise distributions, so the discriminator learns a sharper density ratio estimate. This generally improves the quality of the recovered data-density shape, especially in low-density boundary regions. The tradeoff is higher computational cost per iteration (more samples, more forward/backward work) and longer training time.

2) NCE as a binary classifier

In NCE, the discriminator is trained to answer: "Did this sample come from real data or from the known noise process?" If the discriminator is well trained, its output approximates the posterior probability that a sample is real under the mixed sampling process. By Bayes rule, that posterior depends on how large (p_{data}(x)) is relative to (p_{noise}(x)), so the score is proportional to their ratio up to constants determined by class priors. Intuitively, points that occur much more often under real data than noise get high real-probability, and points typical of noise get low real-probability.

3) Connection to metric learning

In Pairwise/Triplet metric learning, "positive" means semantically matched examples (same-class pair, or anchor-positive), and "negative" means mismatched examples (different-class pair, or anchor-negative). In NCE, the positive class is real data and the negative class is noise samples. The conceptual parallel is that both frameworks learn by discrimination: push the model to score positives higher and negatives lower under a learned representation/decision function. In that sense, both are contrastive because learning is driven by relative comparisons rather than only absolute reconstruction or direct class logits.

4) From NCE to InfoNCE

For one anchor (q), one positive key (k^+), and (N) negatives ({k_i^-}_{i=1}^{N}), a standard InfoNCE loss is

[\mathcal{L}{\mathrm{InfoNCE}} = -\log \frac{\exp\left(\mathrm{sim}(q,k^+)/\tau\right)}{\exp\left(\mathrm{sim}(q,k^+)/\tau\right) + \sum]}^{N} \exp\left(\mathrm{sim}(q,k_i^-)/\tau\right)

where (\mathrm{sim}(\cdot,\cdot)) is typically cosine similarity or dot product. Compared with binary NCE, this is a multiclass softmax objective over one positive versus many negatives in the same denominator. The temperature (\tau) controls softness of the distribution: smaller (\tau) sharpens logits (harder, more peaky contrasts), while larger (\tau) smooths them.

3.4.1 Understanding CLIP

1) Architecture

CLIP has two encoders trained jointly: an image encoder that maps images to embedding vectors, and a text encoder that maps captions/prompts to embedding vectors in the same feature space. For a given image-text pair, their similarity is computed by dot product (equivalently cosine similarity after normalization), producing a scalar match score. L2 normalization is important because it removes magnitude effects and makes the score depend on direction (semantic alignment) rather than vector length. This stabilizes training and ensures consistent similarity comparisons across samples.

2) Symmetric loss

Given a batch of (N) images and (N) matched texts, CLIP forms an (N \times N) similarity matrix where diagonal entries are true pairs and off-diagonals are mismatches. The image-to-text direction treats each image as a query and classifies its correct text among all texts in the batch; the text-to-image direction does the symmetric operation for each text query over all images. The combined objective is

[\mathcal{L}{\mathrm{CLIP}} = \frac{1}{2}\left(\mathcal{L}\right).]}\rightarrow\mathrm{text}} + \mathcal{L}_{\mathrm{text}\rightarrow\mathrm{image}}

Symmetry is important because it enforces bidirectional alignment: images must retrieve correct texts and texts must retrieve correct images, which yields a stronger and more balanced shared embedding space.

3) Zero-shot transfer

A pretrained CLIP model can classify unseen categories by converting category names into text prompts and comparing image embeddings to prompt embeddings. No category-specific fine-tuning is required; prediction is based on whichever text prompt has highest similarity to the image. Typical prompts use natural language templates such as "a photo of a {class name}" (and sometimes multiple templates averaged for robustness). This works because CLIP has already learned broad visual-language correspondences from large-scale image-text pretraining.

4) Batch size

In CLIP training, each sample gets one positive pair and many implicit negatives from off-diagonal entries of the (B \times B) similarity matrix. Larger batch size increases the number and diversity of negatives per update, making the contrastive task harder in a useful way and improving representation quality. The connection to Task 3 NCE is direct: increasing negatives in NCE (larger (N)) improved density discrimination; similarly, larger CLIP batches provide richer negative evidence that sharpens alignment boundaries. The practical cost is higher memory/computation and often the need for distributed training.

3.4.2 Zero-Shot Classification with Pretrained CLIP

Run Command

```
conda run -n ml python
/home/zihao/Documents/ECE60146/HW7/script/zero_shot_classifi
cation.py \
  --dataset-root
/home/zihao/Documents/ECE60146/HW7/dataset_YOLO_hw6 \
  --split val \
  --clip-model ViT-B/32 \
  --artifact-dir
/home/zihao/Documents/ECE60146/HW7/artifacts
```

Zero-Shot Classification Code

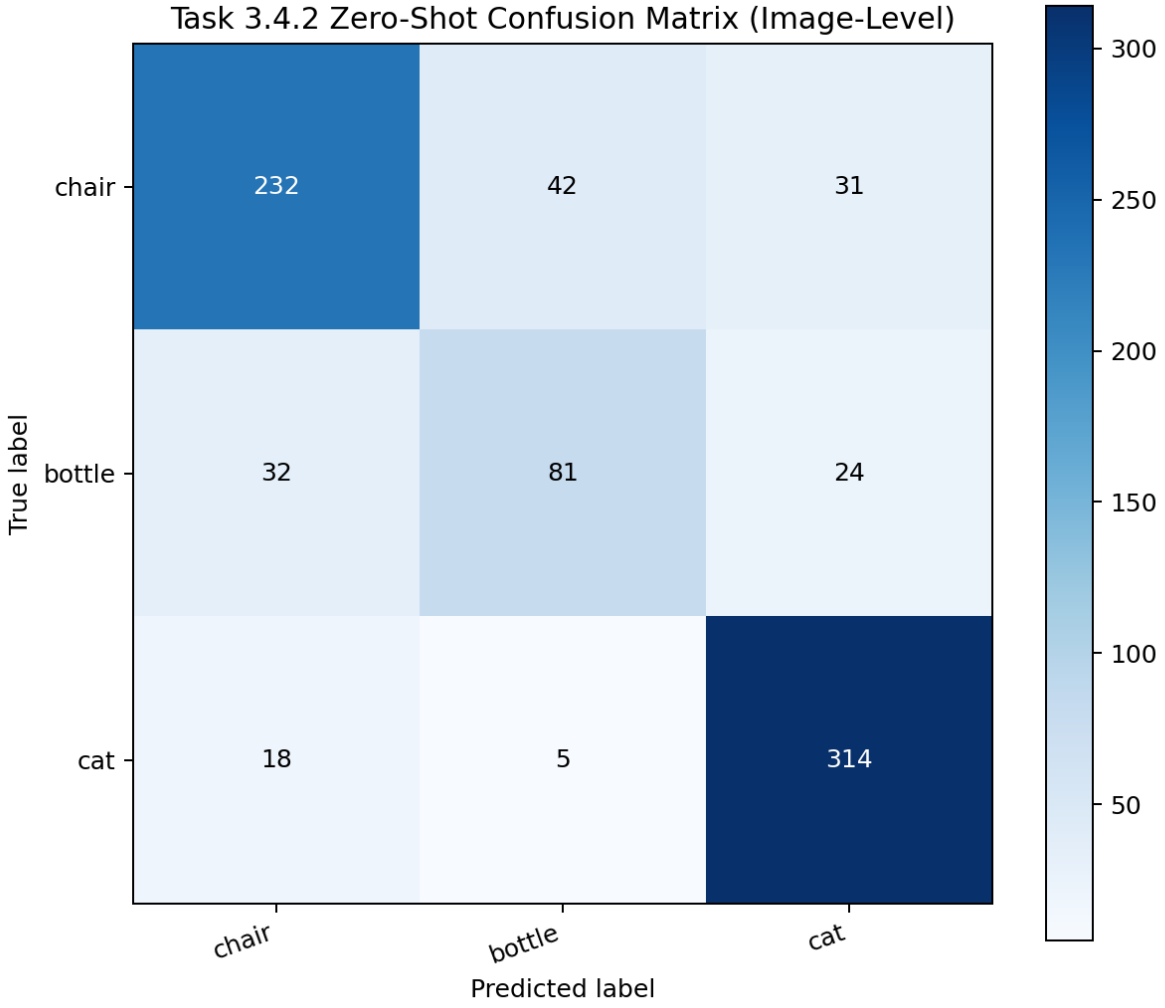
```
import clip
import torch
import torch.nn.functional as F
```

```
from PIL import Image

device = "cuda" if torch.cuda.is_available() else "cpu"
model, preprocess = clip.load("ViT-B/32", device=device)
model.eval()

class_names = ["chair", "bottle", "cat"]
prompts = [f"a photo of a {c}" for c in class_names]
text_tokens = clip.tokenize(prompts).to(device)

with torch.no_grad():
    text_emb = model.encode_text(text_tokens)
    text_emb = F.normalize(text_emb, dim=1)


def classify_image(img_path):
    image =
preprocess(Image.open(img_path).convert("RGB")).unsqueeze(0)
.to(device)
    with torch.no_grad():
        image_emb = model.encode_image(image)
        image_emb = F.normalize(image_emb, dim=1)
        sims = (image_emb @ text_emb.T).squeeze(0)
        pred_idx = int(torch.argmax(sims).item())
    return pred_idx
```

Per-Class and Overall Accuracy

Category	Zero-Shot Accuracy
chair	76.07%
bottle	59.12%
cat	93.18%
Overall	80.49%

3x3 Confusion Matrix



Numeric confusion matrix (rows=true class, columns=predicted class; order: chair, bottle, cat):

$$ \begin{bmatrix} 232 & 42 & 31 \\ 32 & 81 & 24 \\ 18 & 5 & 314 \end{bmatrix} $$

Discussion

Pretrained CLIP generalizes reasonably well to these three LVIS categories without any fine-tuning, achieving 80.49% overall zero-shot accuracy. The strongest class is cat (93.18%), likely because cat has highly distinctive visual semantics and rich representation in web-scale pretraining data. The hardest class is bottle (59.12%), which is consistent with its higher appearance variability (shape, transparency, context, and frequent small-object effects) and stronger confusion with the other classes.

Output Files

- Metrics JSON: `artifacts/3_4_2_zero_shot_metrics_val.json`

- Prediction CSV: `artifacts/3_4_2_zero_shot_predictions_val.csv`
- Confusion matrix image: `artifacts/3_4_2_zero_shot_confusion_matrix_val.png`

3.4.3 Image-to-Image Retrieval with CLIP

Retrieval Code (embedding extraction + FAISS index + query)

```
import faiss
import numpy as np
import torch
import torch.nn.functional as F
from PIL import Image
import clip

device = "cuda" if torch.cuda.is_available() else "cpu"
model, preprocess = clip.load("ViT-B/32", device=device)
model.eval()


def encode_images(image_paths):
    embs = []
    with torch.no_grad():
        for p in image_paths:
            img = Image.open(p).convert("RGB")
            x = preprocess(img).unsqueeze(0).to(device)
            feat = model.encode_image(x)
            feat = F.normalize(feat, dim=1)
            embs.append(feat.squeeze(0).cpu().numpy())
    embs = np.asarray(embs, dtype=np.float32)
    faiss.normalize_L2(embs)
    return embs


# train_embeddings: (N_train, 512), val_embeddings: (N_val,
512)
train_embeddings = encode_images(train_image_paths)
val_embeddings = encode_images(val_image_paths)

index = faiss.IndexFlatIP(512)   # cosine similarity for L2-
normalized vectors
```

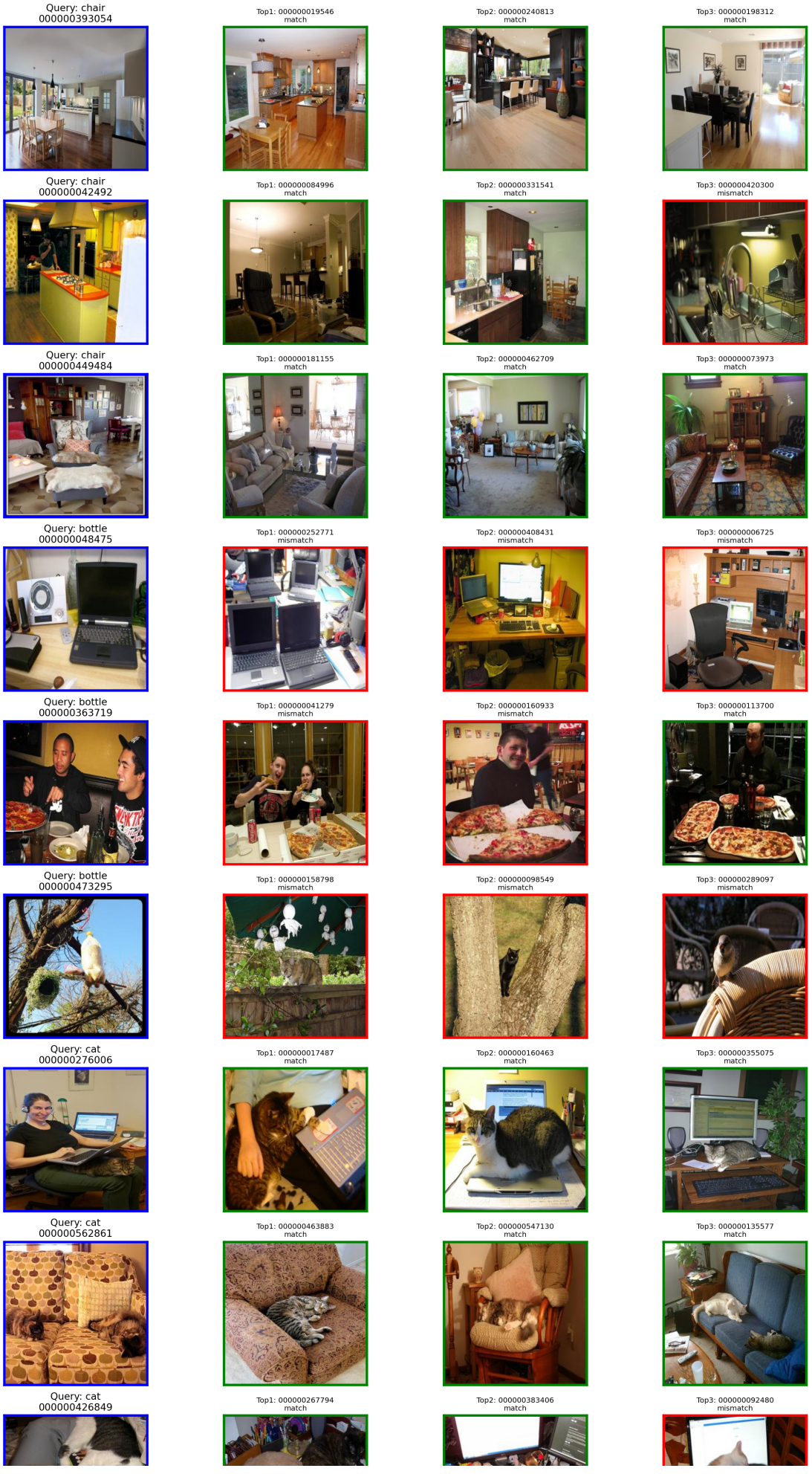
```
index.add(train_embeddings)

k = 5
D, I = index.search(val_embeddings, k)
```

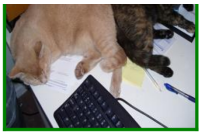
Per-Category Precision Table

Category	Precision@1	Precision@5
chair	71.80%	96.07%
bottle	53.28%	88.32%
cat	91.39%	97.33%
Overall	77.02%	95.25%

Retrieval Visualization Grid (>=9 examples)

The following grid contains at least 9 retrieval examples (3 per category), where each row is query + top-3 retrieved images. Retrieved images have green border when they match the query category and red border otherwise.


Query: chair
000000393054
Top1: 000000019546
match
Top2: 000000240813
match
Top3: 000000198312
match
Query: chair
000000042492
Top1: 000000084996
match
Top2: 000000331541
match
Top3: 000000420300
mismatch
Query: chair
000000449484
Top1: 000000181155
match
Top2: 000000462709
match
Top3: 000000073973
match
Query: bottle
000000048475
Top1: 000000252771
mismatch
Top2: 000000408431
mismatch
Top3: 000000006725
mismatch
Query: bottle
000000363719
Top1: 000000041279
mismatch
Top2: 000000160933
mismatch
Top3: 000000113700
match
Query: bottle
000000473295
Top1: 000000158798
mismatch
Top2: 000000098549
mismatch
Top3: 000000289097
mismatch
Query: cat
000000276006
Top1: 000000017487
match
Top2: 000000160463
match
Top3: 000000355075
match
Query: cat
000000562861
Top1: 000000463883
match
Top2: 000000547130
match
Top3: 000000135577
match
Query: cat
000000426849
Top1: 000000267794
match
Top2: 000000383406
match
Top3: 000000092480
mismatch

Discussion

Compared to a random baseline of about 33% Precision@1 for 3 classes, CLIP retrieval performs much better with 77.02% overall Precision@1. The strongest category is cat (91.39% P@1), indicating highly discriminative visual embeddings for this class. Bottle is the most difficult category (53.28% P@1), likely due to higher intra-class variation and more visual ambiguity with background/context. Precision@5 is high across all classes (overall 95.25%), showing that even when top-1 misses, relevant images are usually retrieved within the top few neighbors.

Output Files

- Retrieval metrics: `artifacts/3_4_3_retrieval_metrics.json`
- Top-5 retrieval CSV: `artifacts/3_4_3_retrieval_top5.csv`
- Retrieval grid image: `artifacts/3_4_3_retrieval_grid.png`

Bonus: Margin Ablation for Triplet Loss

Run Command

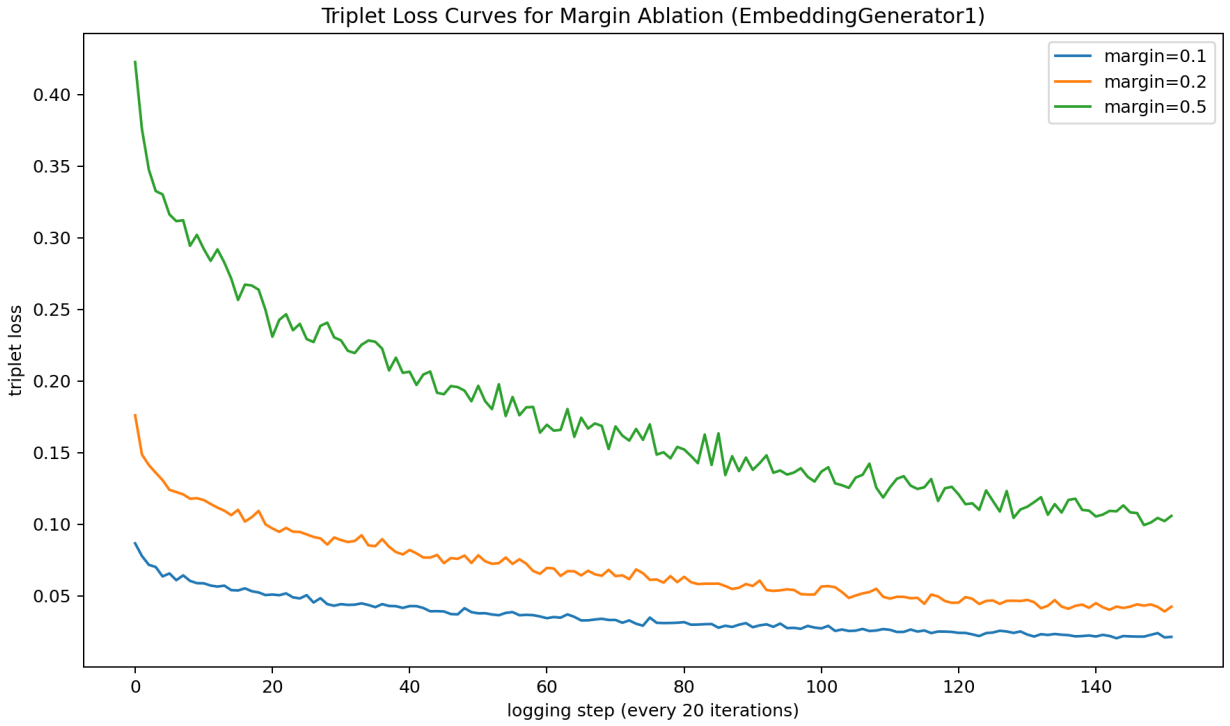
```
conda run -n ml python
/home/zihao/Documents/ECE60146/HW7/script/run_bonus_margin_a
blation.py
```

Code (core ablation loop)

```
margins = [0.1, 0.2, 0.5]
epochs = 8

for margin in margins:
    model_path, loss_history = train_triplet_with_margin(
        dls=dls,
        metric=metric,
        margin=margin,
        epochs=epochs,

save_prefix=f"bonus_triplet_homebrewed_margin_{str(margin).r
```

```
eplace('.', '_')}",
    )

    overall, per_class = evaluate_p_at_1_per_class(
        metric=metric,
        model_path=model_path,
        batch_size=128,
    )
```

Training Loss Curves (single figure, all margins)



Precision@1 by Class and Margin

Class	P@1 (margin=0.1)	P@1 (margin=0.2)	P@1 (margin=0.5)
airplane	77.50%	78.00%	77.60%
automobile	85.90%	87.70%	86.80%
bird	57.60%	62.80%	63.10%
cat	45.20%	51.00%	48.10%
deer	69.70%	71.60%	68.70%
dog	61.00%	57.70%	54.70%
frog	78.60%	80.30%	79.90%
horse	75.60%	79.30%	79.10%
ship	86.20%	86.20%	86.30%
truck	81.30%	84.10%	80.80%
Overall	**71.86%**	**73.87%**	**72.51%**

Discussion

The margin value has a non-monotonic effect in this experiment: increasing from 0.1 to 0.2 improves the overall retrieval quality (71.86% -> 73.87% P@1), but pushing further to 0.5 reduces it to 72.51%. This suggests that a moderate margin gave the best balance between inter-class separation and intra-class compactness for this model/data setup. A larger margin does not always help because it can over-emphasize hard negatives and distort local structure for classes with high visual overlap (for example, the `dog` and `cat` classes here). Therefore, margin should be tuned empirically rather than assumed to improve performance when increased.

Output Files

- Loss curves figure: `artifacts/bonus_margin_ablation_loss_curves.png`
- Metrics JSON: `artifacts/bonus_margin_ablation_metrics.json`
- Run log: `artifacts/bonus_margin_ablation.log`

---

5. Final Check (Task 1 to Bonus)

5.2 Master Summary Table

Section	Method	Main metric(s)	Key result
3.1.1	Pairwise Contrastive (ResNet50)	Precision@1	**76.00%**
3.2.1	Triplet Learning (DLStudio example)	Precision@1	**84.00%**
3.3.1	NCE Density Learning	Discriminator accuracy	**N=1: 95.00%, N=5: 94.20%, N=20: 96.26%**
3.4.2	CLIP Zero-Shot Classification	Overall accuracy	**80.49%**
3.4.2	CLIP Zero-Shot Classification	Per-class accuracy	chair 76.07%, bottle 59.12%, cat 93.18%
3.4.3	CLIP Image Retrieval	Overall Precision@1 / Precision@5	**77.02% / 95.25%**
3.4.3	CLIP Image Retrieval	Per-class Precision@1	chair 71.80%, bottle 53.28%, cat 91.39%
Bonus	Triplet Margin Ablation	Overall Precision@1 by margin	**m=0.1: 71.86%, m=0.2: 73.87%, m=0.5: 72.51%**

5.3 Answers to the Question(s) After the Master Table

1. **Which method worked best for metric learning in this homework setting?**
   Triplet learning in Section 3.2.1 gave the strongest CIFAR-10 embedding retrieval result (84% Precision@1), outperforming pairwise contrastive learning (76% Precision@1).

2. **Is increasing contrastive difficulty always beneficial?**
   Not always. In NCE, increasing negatives to N=20 helped compared with N=5; however, in the bonus triplet study, increasing margin from 0.2 to 0.5 reduced overall P@1. This indicates that stronger constraints help only up to a task-dependent optimum.

3. **What consistent class-level trend appears in CLIP experiments?**
   `cat` is consistently easiest and `bottle` is consistently hardest in both zero-shot classification and retrieval, suggesting CLIP representations for these categories differ in separability under the provided dataset/domain.

4. **Final takeaway across Tasks 1 through Bonus**
   Ranking objectives (triplet/contrastive/InfoNCE-style discrimination) are effective but highly sensitive to hard-negative structure and hyperparameters (number of negatives, margin, template choices). The best practical workflow is to keep the method fixed and tune difficulty controls (margin, negatives, prompts) using validation metrics rather than assuming monotonic gains.