ECE 60146 - HW7

Abin Mathew

3.1 Task 1: Pairwise Constrastive Loss

Objective: Understand the Pairwise Contrastive Loss by running DLStu- dio's example and implementing the loss function from scratch.

3.1.1 Running the DLStudio Example

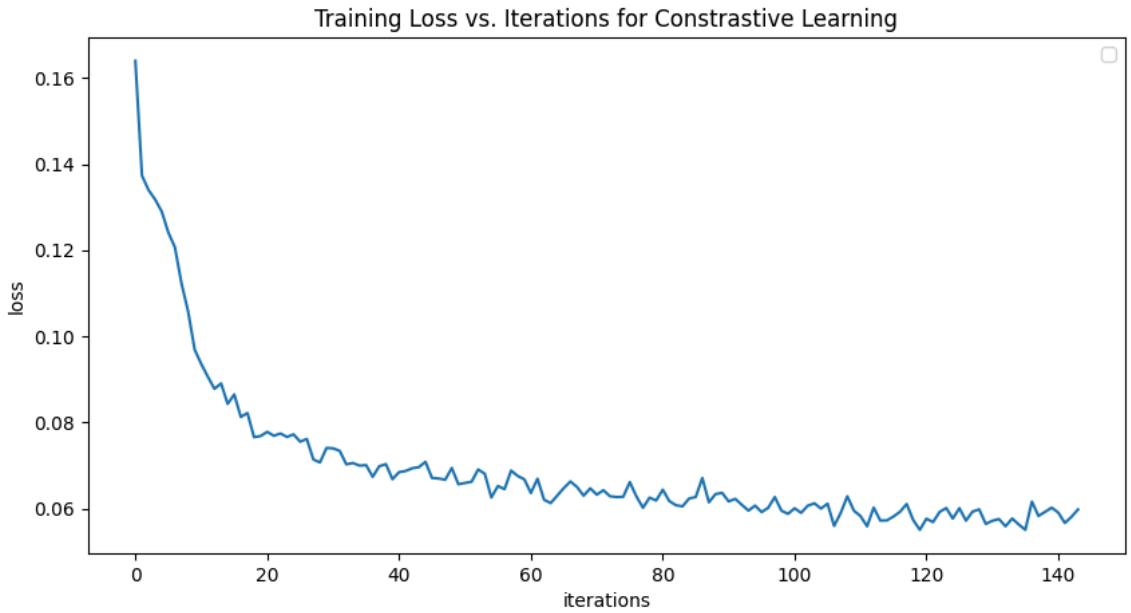
We run the script using the command:

```
uv run DLStudio-
2.5.6/ExamplesMetricLearning/example_for_pairwise_contrastive_loss_supervised.py
```

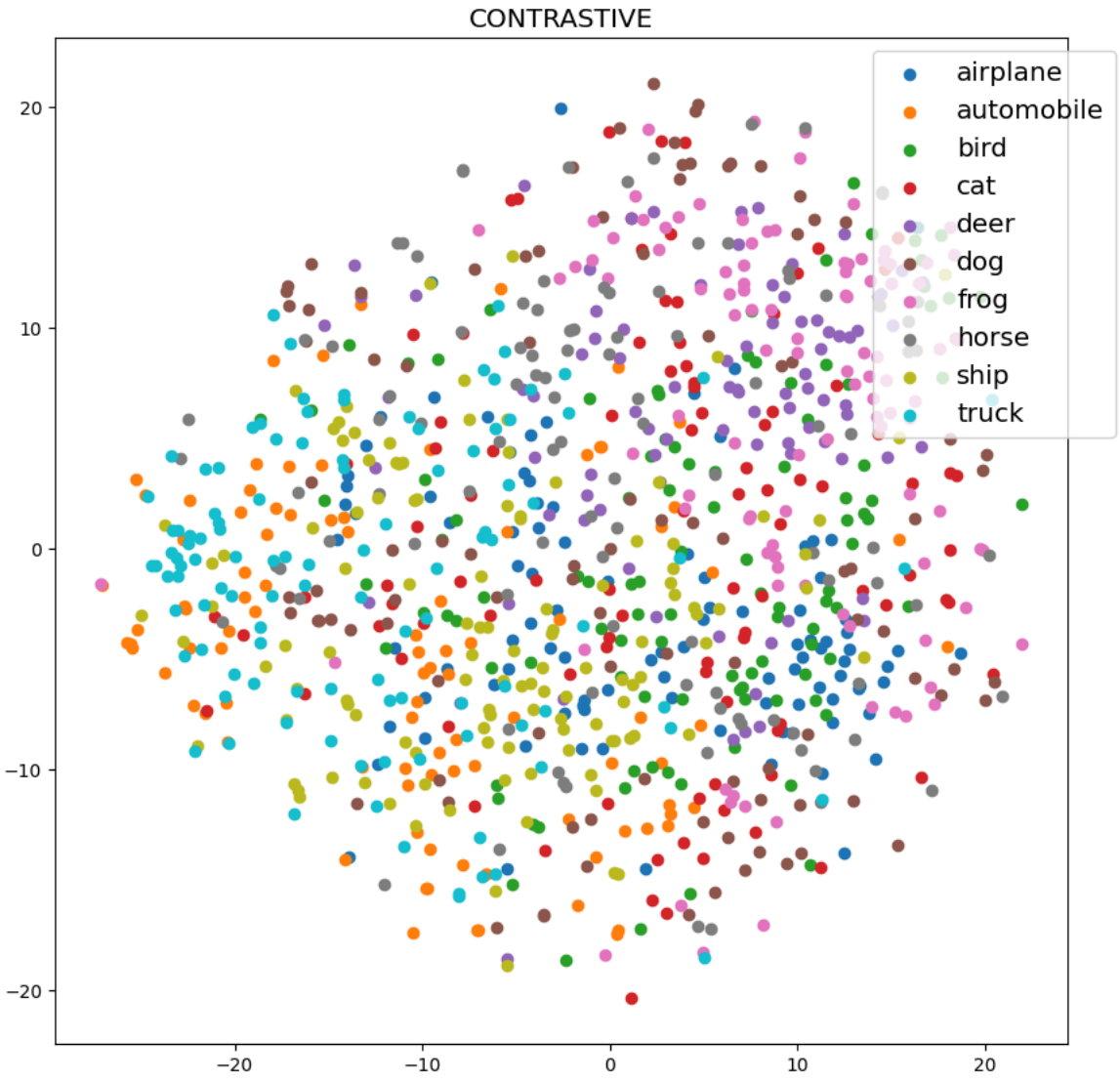
Output(selected lines):

```
The number of learnable parameters in the model: 23770304


The number of layers in the model: 161



Finished Training

[t-SNE] Computing 121 nearest neighbors...
[t-SNE] Indexed 1024 samples in 0.000s...
[t-SNE] Computed neighbors for 1024 samples in 0.210s...
[t-SNE] Computed conditional probabilities for sample 1000 / 1024
[t-SNE] Computed conditional probabilities for sample 1024 / 1024
[t-SNE] Mean sigma: 1.250517
[t-SNE] KL divergence after 250 iterations with early exaggeration: 64.231071
[t-SNE] KL divergence after 1000 iterations: 1.715109
[t-SNE] Computing 121 nearest neighbors...
[t-SNE] Indexed 1024 samples in 0.000s...
[t-SNE] Computed neighbors for 1024 samples in 0.176s...
[t-SNE] Computed conditional probabilities for sample 1000 / 1024
[t-SNE] Computed conditional probabilities for sample 1024 / 1024
[t-SNE] Mean sigma: 1.167731
[t-SNE] KL divergence after 250 iterations with early exaggeration: 44.675045
[t-SNE] KL divergence after 1000 iterations: 0.187814


precision_at_rank_1 with CONTRASTIVE learning: 76%
```

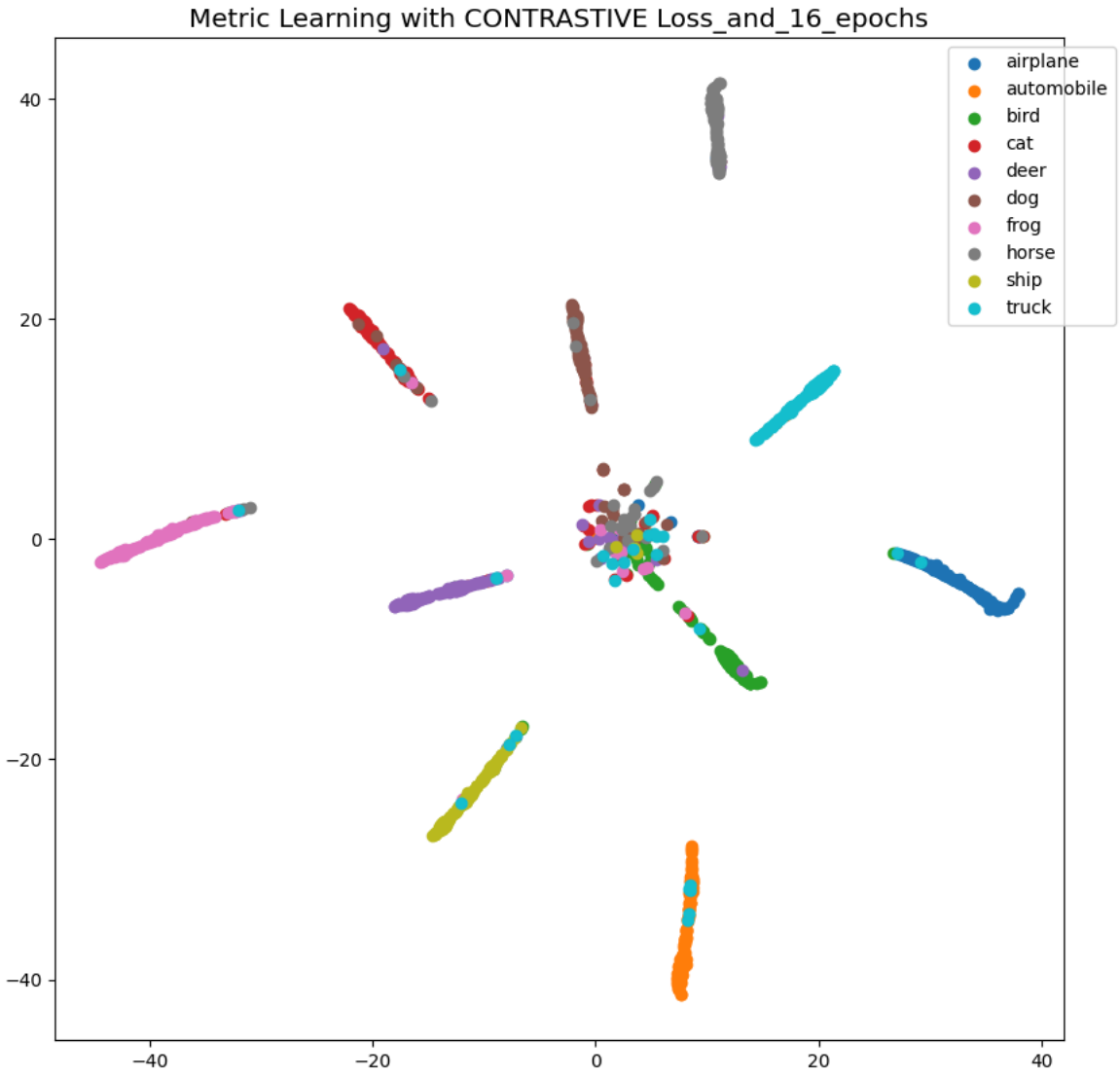
Training Loss Curve:



t-SNE visualization of raw data:



t-SNE visualization of the test-set embeddings:



3.1.2 Understanding the Loss

1. The PCL has two components, one for positive pairs and one for negative pairs. The loss is given by:

$$\mathcal{L} = \sum_i \left[(1 - y_i)\, d\big(f(x_1^i), f(x_2^i)\big)^2 + y_i \big(\max\{0,\, m - d\big(f(x_1^i), f(x_2^i)\big)\}\big)^2 \right].$$

The first term pulls positive pairs together by penalizing large embedding distance. The second term pushes negative pairs apart only when they are too close. The margin m acts as a cutoff: if a negative pair already has distance greater than m, then $\max(0, m - d) = 0$, so that pair contributes zero loss and zero gradient.

2. It is important to mine both positive and negative pairs because informative pairs are a small subset of all possible pairs. If we compute loss over all pairs, the number of pairs grows quadratically with batch size, and most negative pairs quickly become easy negatives that already satisfy the margin. Those

easy pairs contribute little useful gradient but can dominate the loss statistics and slow meaningful learning. Also, using too many easy negatives creates a gradient dilution problem: the model appears to optimize the objective while still failing to separate hard, boundary-level examples. Mining hard or semi-hard positives/negatives focuses optimization on the pairs that still violate the objective, which improves embedding quality and metrics like Precision@1.

3.1.3 Implementing Pairwise Contrastive Loss

Implementation of PCL using tensor operations.

Code

```
def compute_distance_matrix(embeddings):
    """
    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 of all pairs
    dot = torch.mm(embeddings, embeddings.T)

    # Step 2: squared norms (the diagonal of dot)
    sq_norms = dot.diagonal()

    # Step 3: apply the identity: ||x-y||^2 = ||x||^2 - 2*(x.y^T) + ||y||^2
    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 in the batch.

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

    Negative pair (different labels):
        loss_ij = max(0, margin - sqrt(dist[i,j]))^2

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

```
        margin: float
    Returns:
        loss: scalar tensor (mean over all pairs)
    """
    B = embeddings.shape[0]
    dist_sq = compute_distance_matrix(embeddings)

    # Build positive / negative masks using broadcasting (no for-loops)
    labels_equal = (labels.unsqueeze(0) == labels.unsqueeze(1))
    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: squared distance
    pos_loss = dist_sq * pos_mask.float()

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

    # Average over all pairs (positive + negative)
    num_pairs = pos_mask.sum() + neg_mask.sum()
    loss = (pos_loss.sum() + neg_loss.sum()) / (num_pairs.float() + 1e-8)

    return loss

if __name__ == '__main__':
    # Verification block: Task 1 (Pairwise Contrastive)
    torch.manual_seed(42)
    B, D = 8, 16
    emb = F.normalize(torch.randn(B, D), dim=1)
    labels = torch.tensor([0, 0, 1, 1, 2, 2, 3, 3])

    dm = compute_distance_matrix(emb)
    assert dm.shape == (B, B)
    assert dm.diagonal().abs().max().item() < 1e-5, "Diagonal should be zero"

    loss_c = pairwise_contrastive_loss(emb, labels)
    assert loss_c.item() >= 0
    print(f"Distance matrix diagonal max:
{dm.diagonal().abs().max().item():.2e}")
    print(f"Contrastive loss (margin = 1.0): {loss_c.item():.4f}")
```

Output:

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

Observations:

- The labels.unsqueeze(0) == labels.unsqueeze(1) operation creates a (B, B) boolean matrix where entry (i, j) is True if labels[i] == labels[j], and False otherwise. This allows us to identify which pairs of samples

belong to the same class (positive pairs) and which belong to different classes (negative pairs) without using explicit loops.

3.2 Task 2: Triplet Loss

Objective: Understand the Triplet Loss and its mining strategies by run- ning DLStudio's example and implementing the loss.

3.2.1 Running the DLStudio Example

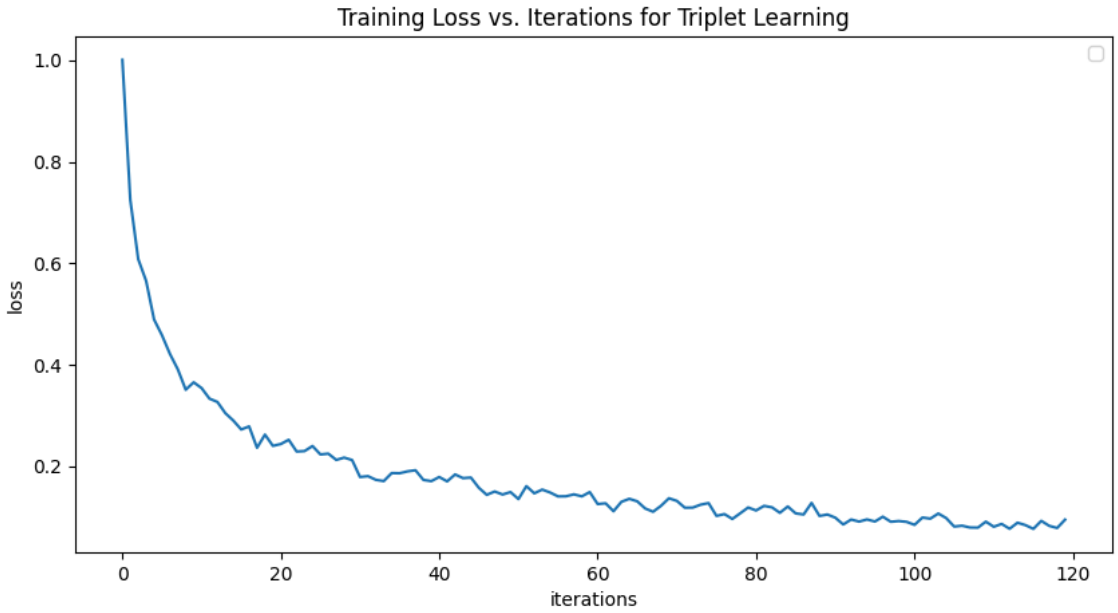
We run the script using the command:

```
uv run DLStudio-
2.5.6/ExamplesMetricLearning/example_for_triplet_loss_supervised.py
```

Output(selected lines):

```
Finished Training

[t-SNE] Computing 121 nearest neighbors...
[t-SNE] Indexed 640 samples in 0.000s...
[t-SNE] Computed neighbors for 640 samples in 0.173s...
[t-SNE] Computed conditional probabilities for sample 640 / 640
[t-SNE] Mean sigma: 1.323029
[t-SNE] KL divergence after 250 iterations with early exaggeration: 43.051052
[t-SNE] KL divergence after 1000 iterations: 0.277072
True



precision_at_rank_1 with TRIPLET learning: 82%
```

Training Loss Curve:



t-SNE visualization of the test-set embeddings:

3.2.2 Understanding Triplet Loss

Triplet Loss eqn:

$$\mathcal{L} = \sum_{i=1}^{N} \max\left\{ \left\| f\left(x_i^a\right) - f\left(x_i^p\right) \right\|_2^2 - \left\| f\left(x_i^a\right) - f\left(x_i^n\right) \right\|_2^2 + \delta,\, 0 \right\}.$$

A triplet (a, p, n) is valid when the anchor a and positive p share the same class, while the negative n belongs to a different class. The objective enforces that the anchor-negative distance should exceed the anchor-positive distance by at least margin δ. If that condition is already satisfied, i.e., $\|f(a) - f(n)\|_2^2 \geq \|f(a) - f(p)\|_2^2 + \delta$, the expression inside max becomes non-positive, so the loss for that triplet is exactly zero. This is why easy negatives (already far from the anchor) do not contribute gradients.

Using margin δ, negatives are commonly categorized as hard, semi-hard, and easy. Hard negatives are those closer to the anchor than the positive (roughly $d(a,n) < d(a,p)$), so they strongly violate the ranking and produce large gradients. Semi-hard negatives satisfy $d(a,p) < d(a,n) < d(a,p) + \delta$: they are farther than the positive but still inside the margin, so they provide useful corrective signal without being extreme. Easy negatives satisfy $d(a,n) \geq d(a,p) + \delta$ and therefore contribute zero loss.

In practice, training only with hard negatives often causes instability because these examples can be outliers, mislabeled cases, or overly difficult samples that generate very large and noisy gradients. The model may collapse embeddings toward degenerate configurations. Semi-hard mining is usually preferred because it keeps informative violations, giving smoother optimization and better class-structured embedding spaces.

3.2.3 Implementing Triplet Loss

Implementation of Triplet Loss using tensor operations.

Code

```python
def get_triplet_mask(labels):
    """
    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.
    See Slides 52-60 of the lecture.

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

    # All-distinct indices mask
    not_eq = ~torch.eye(B, dtype=torch.bool, device=device)
    i_ne_j = not_eq.unsqueeze(2)
    i_ne_k = not_eq.unsqueeze(1)
    j_ne_k = not_eq.unsqueeze(0)
    distinct = i_ne_j & i_ne_k & j_ne_k

    # Label equality mask
    eq = (labels.unsqueeze(0) == labels.unsqueeze(1))
    ij_eq = eq.unsqueeze(2)      # i == j (anchor-positive)
    ik_eq = eq.unsqueeze(1)      # i == k
    valid_labels = ij_eq & (~ik_eq)

    return distinct & valid_labels
```

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

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

    Args:
        embeddings: FloatTensor (B, D), L2-normalized
        labels: LongTensor (B,)
        margin: float (delta in the lecture)
    Returns:
        loss: scalar tensor
        num_triplets: int, number of non-zero triplets
    """
    dist = compute_distance_matrix(embeddings)
    mask = get_triplet_mask(labels)

    # dist_ap[i,j,k] = dist[i,j] (anchor-positive)
    dist_ap = dist.unsqueeze(2).expand(dist.shape[0], dist.shape[1],
dist.shape[0])

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

    triplet_loss_vals = torch.clamp(dist_ap - dist_an + margin, min=0.0)

    # Apply mask and average over non-zero losses
    triplet_loss_vals = triplet_loss_vals * mask.float()
    num_pos = (triplet_loss_vals > 1e-16).sum().item()
    loss = triplet_loss_vals.sum() / (num_pos + 1e-8)

    return loss, int(num_pos)

# Verification block: Task 2 (Triplet Loss)
torch.manual_seed(7)
B, D = 12, 32
emb = F.normalize(torch.randn(B, D), dim=1)
labels = torch.tensor([0,0,0,1,1,1,2,2,2,3,3,3])

mask = get_triplet_mask(labels)
assert mask.shape == (B, B, B)
assert mask.diagonal(dim1=0, dim2=1).any() == False

loss_t, n = triplet_loss(emb, labels)
assert loss_t.item() >= 0
print(f"Triplet loss: {loss_t.item():.4f}, non-zero triplets: {n}")
```

Output:

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

Observations:

Comparing the Precision@1 from the PCL and Triplet Loss examples, we see that the Triplet Loss achieves a higher Precision@1 (82%) compared to the Pairwise Contrastive Loss (76%). This is similar to the results observed in the slides(84% vs 74%).

3.3 Task 3: Noise Contrastive Estimation

Objective: Understand how NCE trains a discriminator to learn an implicit probability density model by distinguishing real data from injected noise

3.3.1 Running the NCE Script

Code

```
uv run hw7_AbinMathew/NCE_for_learning_point_distro.py 1
uv run hw7_AbinMathew/NCE_for_learning_point_distro.py 5
uv run hw7_AbinMathew/NCE_for_learning_point_distro.py 20
```

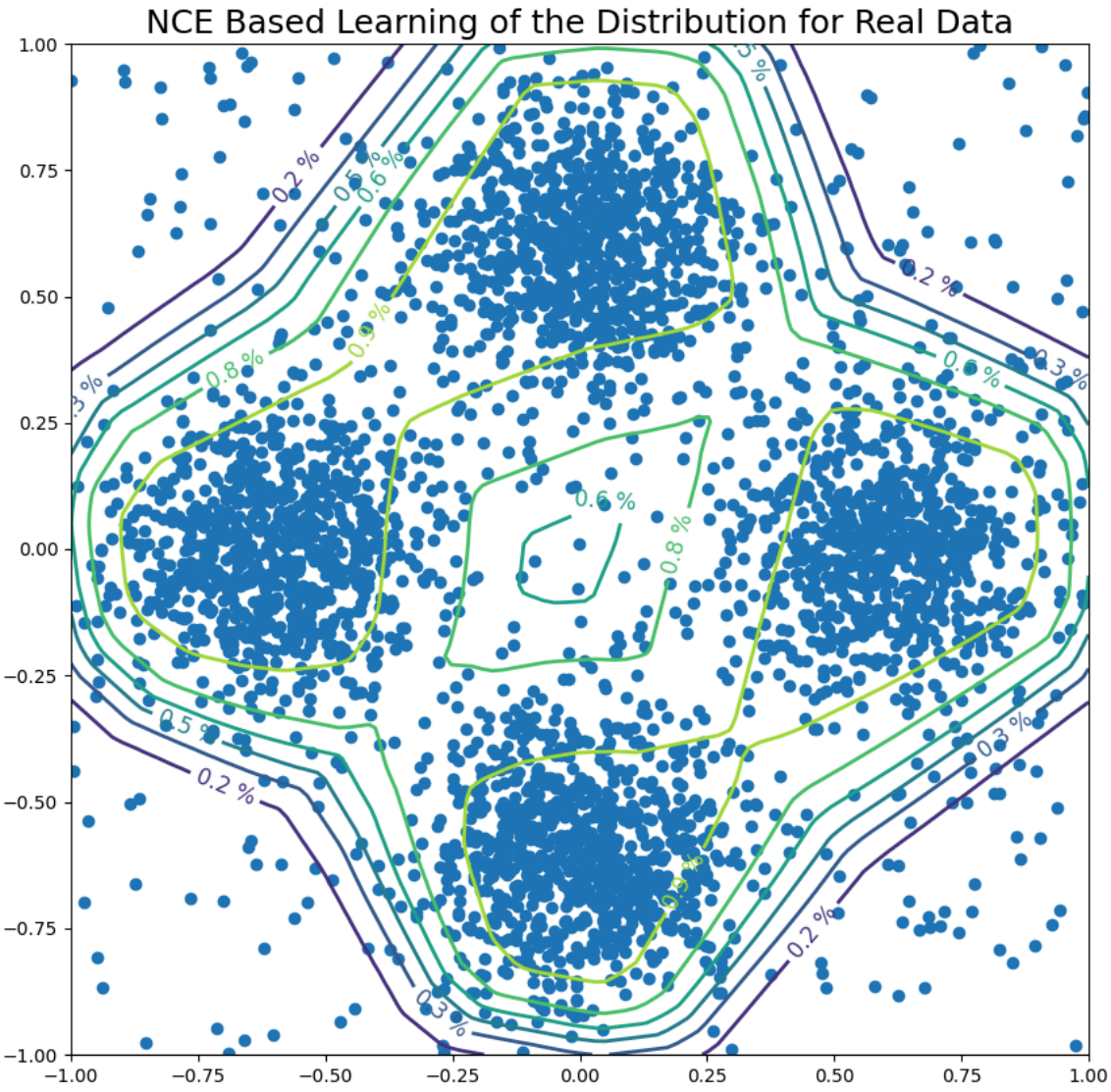
Discriminator Accuracy:

Negatives per positive(N)	Discriminator Accuracy
1	94.91
5	94.06
20	96.25

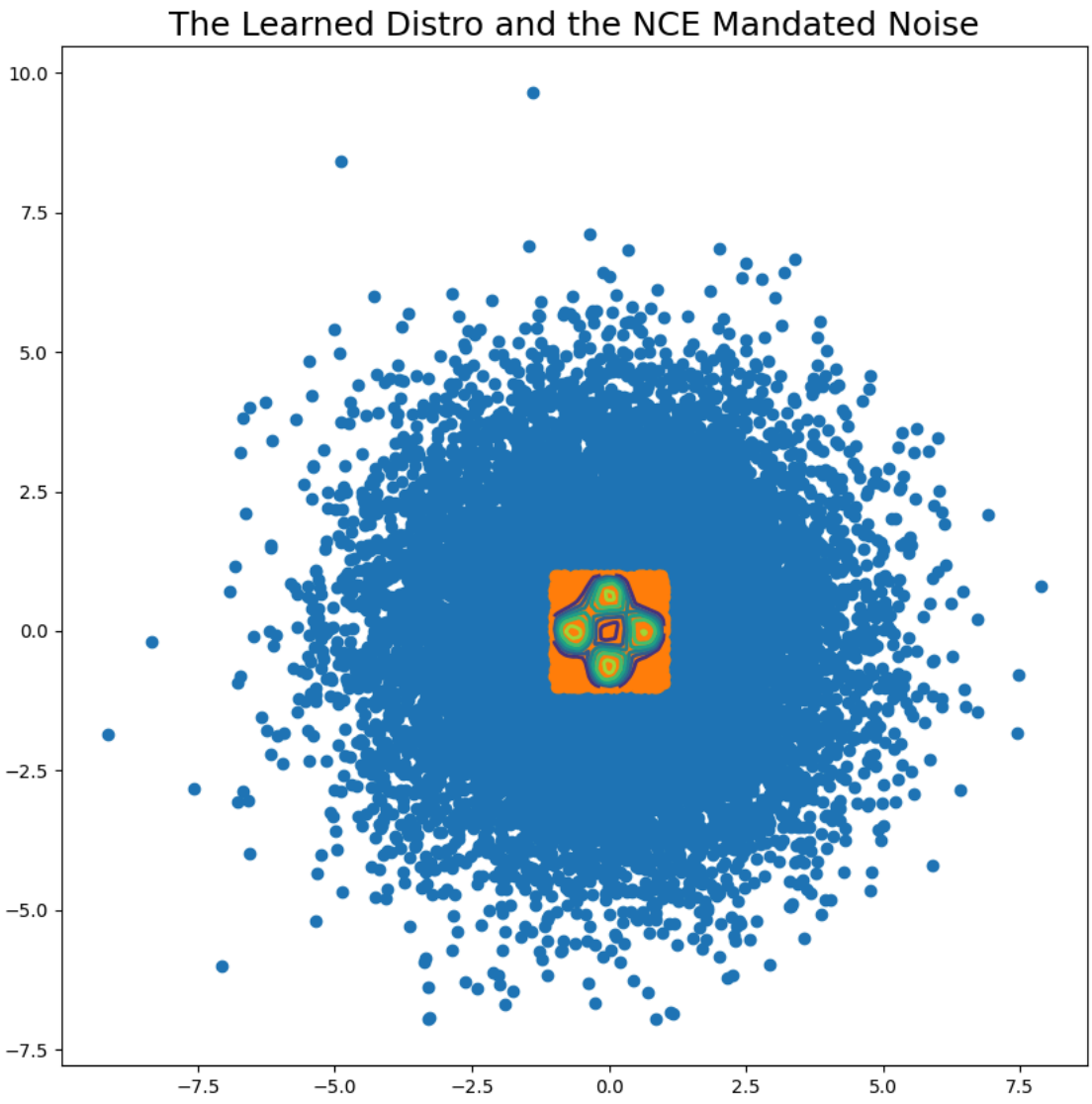
Output Plots:

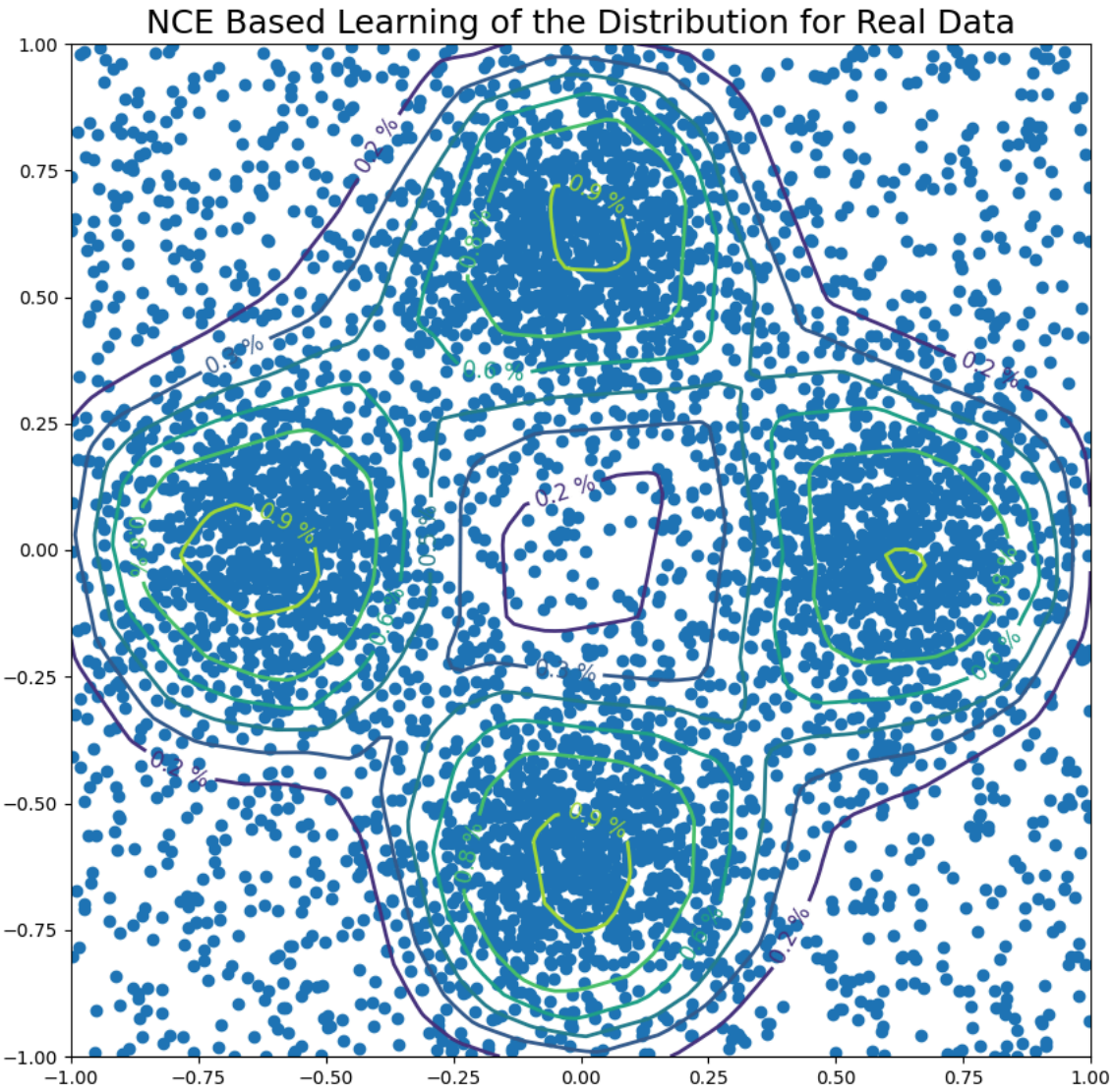
N = 1:


The Learned Distro and the NCE Mandated Noise
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

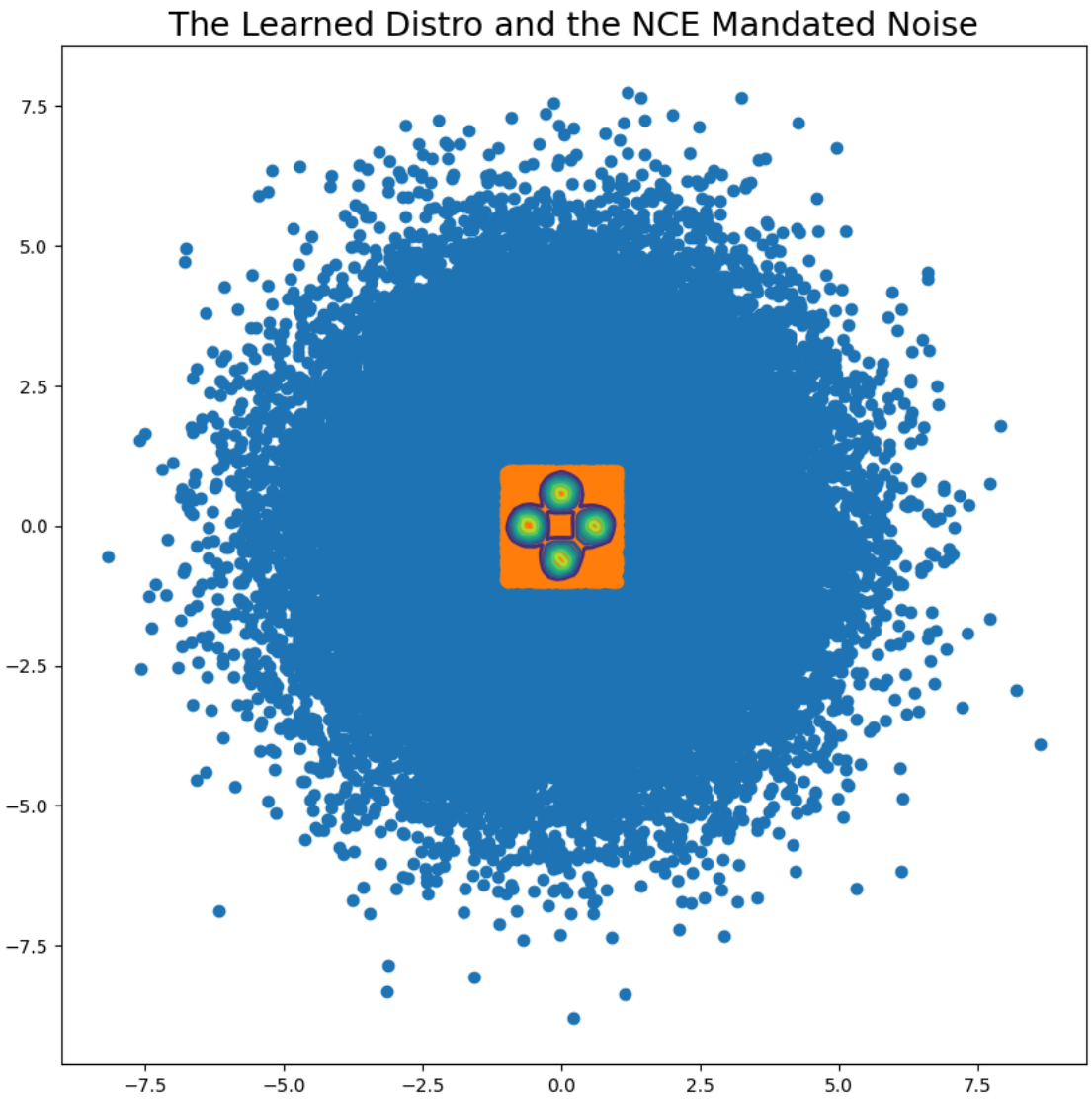

NCE Based Learning of the Distribution for Real Data
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
0.2 %
0.5 %
0.6 %
0.9 %
0.8 %
0.6 %
0.8 %
0.6 %
0.2 %
0.3 %
0.5 %
0.2 %
0.3 %
0.3 %
0.2 %
0.3 %

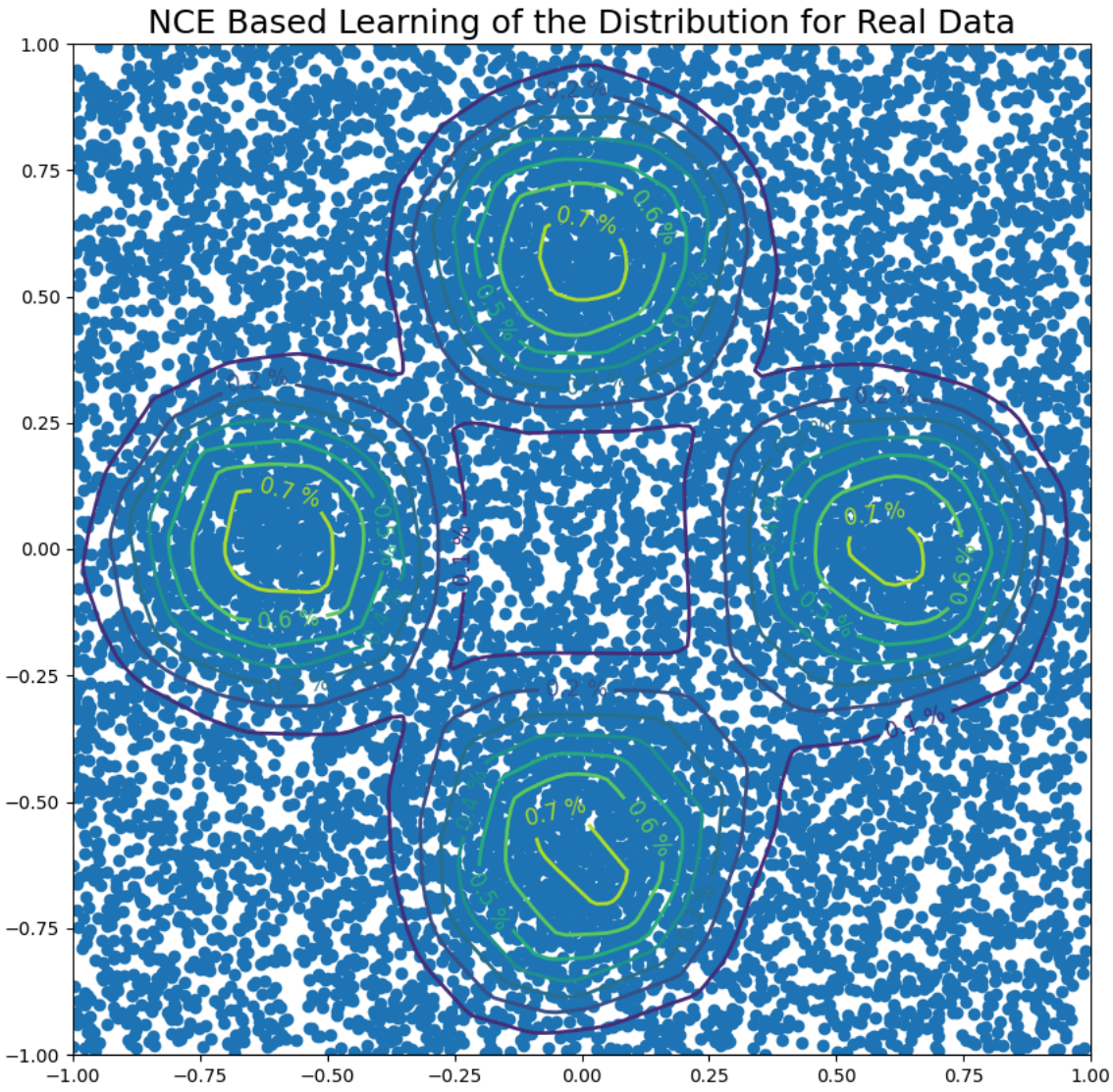
N = 5:


The Learned Distro and the NCE Mandated Noise
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5


NCE Based Learning of the Distribution for Real Data
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
0.2 %
0.6 %
0.9 %

N = 20:


The Learned Distro and the NCE Mandated Noise
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5



3.3.2 Conceptual Questions

1. Effect of N: As N increases, the dicriminator is trained with much higher number of noise(negatives) samples for each data point, as a result, the estimated density is closer to true data distribution and the contours become smoother. The reason this happens is because NCE trains the discrimnator to distinguish real data from noise. With more noise samples, the discriminator gets a richer set of negative examples, which forces it to learn a more accurate decision boundary. The cost of increasing N is that compute cost grows linearly as well as there can be diminishing returns after a point, since the discriminator may already be performing well with fewer negatives.

2. NCE as a binary classifier: if x is more typical under real data than noise, then discrimator says "real" with high confidence. If x is typical under noise, the discriminator says "noise" with high confidence. The confidence level is determined by how large one density is relative to the other, which is why NCE can recover the data density ratio from binary classification.

3. Connection to metric learning: In NCE the positive role is played by the real data distribution, while the negative role is played by the noise distribution. both approaches are doing "contrastive learning" by pushing the model to assign higher scores to real data (positives) and lower scores to noise (negatives).

4. The eqn for InfoNCE loss is given by:

$$\mathcal{L}_{\text{InfoNCE}} = -\mathbb{E}_x \left[\log \frac{\exp(\mathbf{v} \cdot \mathbf{v}^+ / \tau)}{\exp(\mathbf{v} \cdot \mathbf{v}^+ / \tau) + \sum_{n=1}^{N} \exp\left(\mathbf{v} \cdot \mathbf{v}_n^- / \tau\right)} \right].$$

The temperature parameter τ scales the dot products (logits) before the softmax operation, which controls the sharpness of the resulting probability distribution and dictates how heavily the model penalizes "hard" negative samples.

3.4 Task 4 CLIP Experiments on LVIS Dataset

Objective: Understand CLIP's multi-modal metric learning framework and evaluate the pretrained CLIP model on your own LVIS data from HW6, both for zero-shot classification and image retrieval

3.4.1 Understanding CLIP

1. CLIP uses one text encoder and one image encoder to map both modalities into a single shared embedding space. Their outputs are L2-normalized, so that the dot product between an image embedding and a text embedding directly calculates the cosine similarity, ensuring the score measures purely the angular alignment between the image and text representations, independent of their original vector magnitudes.
2. In the $N \times N$ similarity matrix (where rows are images and columns are texts in a batch), the "image-to-text" direction calculates the loss across the columns for a given row—meaning for a given image, it tries to identify the correct text caption out of N options. The "text-to-image" direction calculates the loss across the rows for a given column—meaning for a given text caption, it tries to identify the correct image out of N options. The combined loss is written as:

$$\mathcal{L}_{CLIP} = \frac{1}{2}(\mathcal{L}_{image \to text} + \mathcal{L}_{text \to image})$$

Symmetry is important because CLIP aims to build a joint embedding space that works robustly in both directions. The model must be penalized equally for failing to retrieve the right image for a text prompt as it is for failing to retrieve the right text for an image prompt.

3. A trained CLIP model can classify an image into a category it has never explicitly been trained to classify by projecting the target image into the shared embedding space alongside text embeddings of all candidate classes. It then assigns the image to the class whose text embedding yields the highest cosine similarity. To maximize accuracy, the text prompts typically don't just use the raw category name (e.g., "car"), but are placed into natural language templates like "a photo of a {class_name}", which better matches the distribution of the text the model saw during pretraining.

4. In CLIP's $B \times B$ similarity matrix, the diagonal contains the positive pairs, and the $B - 1$ off-diagonal entries automatically serve as negative samples. A larger batch size significantly improves the quality of CLIP training because it exposes the model to a much larger pool of negative examples ($B - 1$ negatives per positive) in every step, forcing it to learn highly discriminative features to tell them apart. Conceptually, this is identical to the NCE result in Task 3: just as increasing N (the number of noise samples) provided a denser background to yield a more accurate density estimate, increasing the batch size in CLIP provides a richer set of negatives to define much sharper boundaries in the multi-modal embedding space.

3.4.2 Zero Shot Classification with Pretrained CLIP

Code for Zero-Shot Classification

```
def perform_zero_shot_classification():
    # ------------------------------------------------------------
    # Setup & Zero-Shot Classification
    # ------------------------------------------------------------
    device = "cuda" if torch.cuda.is_available() else "cpu"
    model, preprocess = clip.load("ViT-B/32", device=device)
    model.eval()


    class_names = ["car", "chair", "bottle"]
    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):
        """Zero-shot classification for a single image"""
        img =
preprocess(Image.open(img_path).convert('RGB')).unsqueeze(0).to(device)
        with torch.no_grad():
            img_emb = model.encode_image(img)
            img_emb = F.normalize(img_emb, dim=1)

            # Cosine similarity to each text prompt
            sims = (img_emb @ text_emb.T).squeeze(0)
            return sims.argmax().item()
    # Create a confusion matrix based on the classification results.
    def create_confusion_matrix(results):
        """Creates, prints, and plots a labeled confusion matrix."""
        num_classes = len(class_names)
        conf_matrix = np.zeros((num_classes, num_classes), dtype=int)

        for _, pred_idx, actual_idx in results:
            conf_matrix[actual_idx, pred_idx] += 1

        # Row-normalized matrix for easier comparison across classes.
```

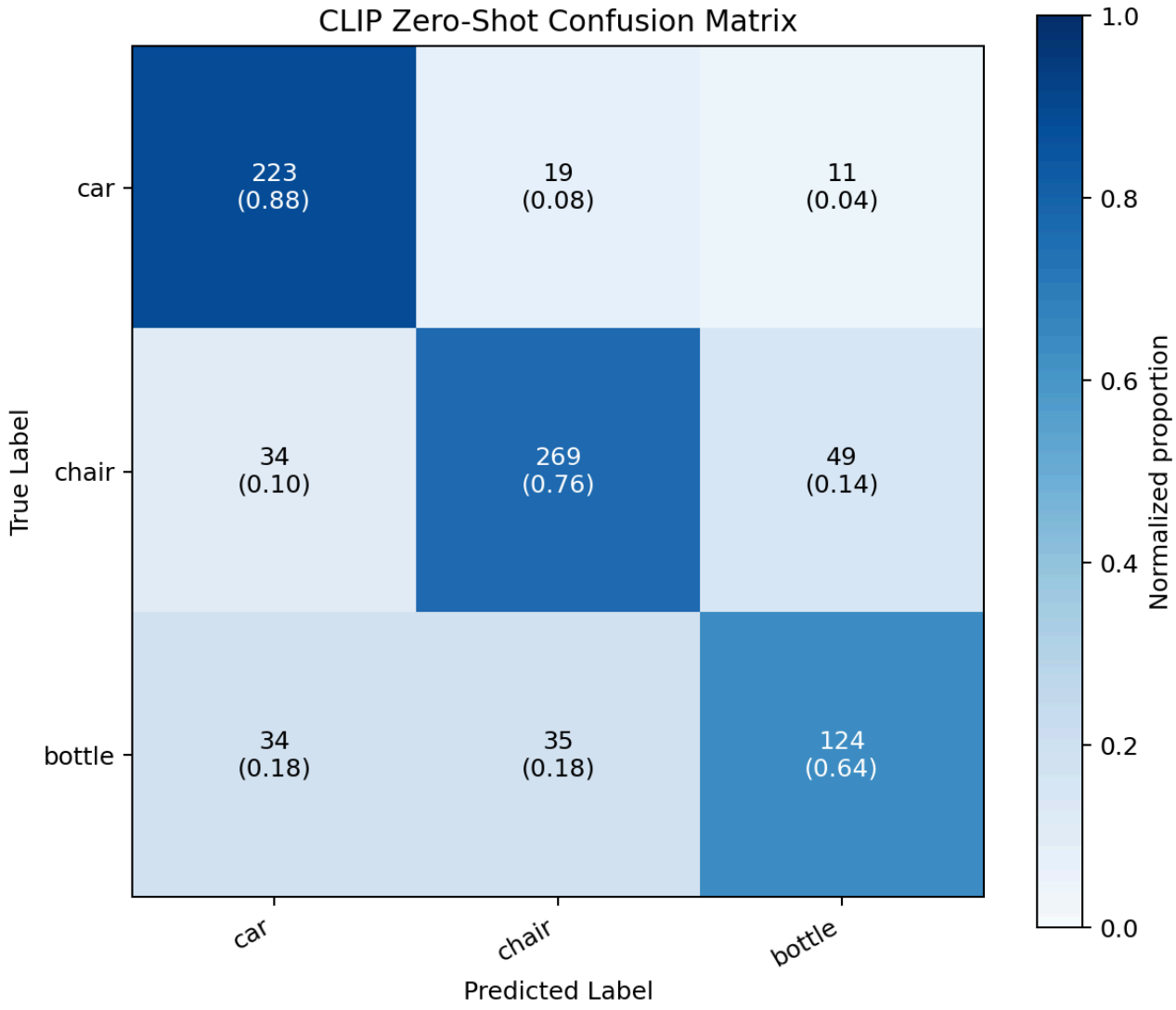
```
        row_sums = conf_matrix.sum(axis=1, keepdims=True)
        conf_matrix_norm = np.divide(
            conf_matrix,
            np.maximum(row_sums, 1),
            where=row_sums != 0,
        )

        fig, ax = plt.subplots(figsize=(7, 6))
        im = ax.imshow(conf_matrix_norm, cmap="Blues", vmin=0.0, vmax=1.0)
        cbar = fig.colorbar(im, ax=ax)
        cbar.set_label("Normalized proportion", rotation=90)

        ax.set_title("CLIP Zero-Shot Confusion Matrix")
        ax.set_xlabel("Predicted Label")
        ax.set_ylabel("True Label")
        ax.set_xticks(np.arange(num_classes), labels=class_names)
        ax.set_yticks(np.arange(num_classes), labels=class_names)

        # Rotate x-axis labels for readability.
        plt.setp(ax.get_xticklabels(), rotation=30, ha="right")

        # Annotate each cell with count and normalized value.
        for i in range(num_classes):
            for j in range(num_classes):
                count = conf_matrix[i, j]
                frac = conf_matrix_norm[i, j]
                text_color = "white" if frac > 0.5 else "black"
                ax.text(
                    j,
                    i,
                    f"{count}\n({frac:.2f})",
                    ha="center",
                    va="center",
                    color=text_color,
                    fontsize=10,
                )

        fig.tight_layout()
        plt.savefig("hw7_AbinMathew/clip_confusion_matrix.png", dpi=180)
        plt.show()

        # Print a simple accuracy report table.
        print("\nAccuracy Report:")
        print(f"{'Class':<12} {'Correct':>8} {'Total':>8} {'Accuracy':>10}")
        print("-" * 42)
        for idx, class_name in enumerate(class_names):
            correct = int(conf_matrix[idx, idx])
            total = int(row_sums[idx, 0])
            acc = (correct / total) if total > 0 else 0.0
            print(f"{class_name:<12} {correct:>8d} {total:>8d} {acc:>9.2%}")

        overall_correct = int(np.trace(conf_matrix))
        overall_total = int(conf_matrix.sum())
        overall_acc = (overall_correct / overall_total) if overall_total > 0
else 0.0
```

```
        print("-" * 42)
        print(f"{'Overall':<12} {overall_correct:>8d} {overall_total:>8d}
{overall_acc:>9.2%}")


    results_for_confusion_matrix = []
    for img_file in os.listdir(OUTPUT_DIR_IMAGES):
        img_path = os.path.join(OUTPUT_DIR_IMAGES, img_file)
        actual_class_label_file_path = os.path.join(OUTPUT_DIR_LABELS,
img_file.replace('.jpg', '.json'))
        actual_class_label_idx = json.load(open(actual_class_label_file_path))
[0]['category_index']
        actual_class_name = class_names[actual_class_label_idx]
        pred_class_idx = classify_image(img_path)
        print(f"Image: {img_file}, Predicted Class:
{class_names[pred_class_idx]}, Actual Class: {actual_class_name}")
        results_for_confusion_matrix.append((img_file, pred_class_idx,
actual_class_label_idx))

    create_confusion_matrix(results_for_confusion_matrix)
```

Per class accuracy and Overall accuracy:

```
Accuracy Report:
Class         Correct    Total   Accuracy
-----------------------------------------
car               223      253    88.14%
chair             269      352    76.42%
bottle            124      193    64.25%
-----------------------------------------
Overall           616      798    77.19%
```

Confusion Matrix:



Observations:

The pretrained CLIPE generalize pretty well to the LVIS dataset, achieving overall accruacy of 77%. The highest was for "car" class (88%), while the lowest was for "bottle" class (64%). The confusion matrix shows that the misclassifications for "bottle" were predicted as "chair" and "car" equally, which may be due to some visual similarities in the dataset or the model's bias from pretraining.

3.4.3 Image-toIMage Retrieval with CLIP

Code for Image Retrieval

```
# Image-to-Image Retrieval with CLIP
def perform_image_to_image_retrieval():
    device = "cuda" if torch.cuda.is_available() else "cpu"
    model, preprocess = clip.load("ViT-B/32", device=device)
    model.eval()

    train_image_paths, train_labels = load_image_paths_and_labels(
        OUTPUT_DIR_IMAGES_TRAIN, OUTPUT_DIR_LABELS_TRAIN
    )
```

```
    val_image_paths, val_labels = load_image_paths_and_labels(
        OUTPUT_DIR_IMAGES_VAL, OUTPUT_DIR_LABELS_VAL
    )

    # 1. Extract Embeddings
    print("Extracting training embeddings...")
    train_embeddings = extract_embeddings(model, preprocess, train_image_paths,
device,batch_size=64)
    print("Extracting validation embeddings...")
    val_embeddings = extract_embeddings(model, preprocess, val_image_paths,
device,batch_size=64)

    # 2. Build FAISS index
    embed_dim = 512 # ViT-B/32 embedding dimension
    index = faiss.IndexFlatIP(embed_dim) # Inner product = Cosine sim for L2-
normalized vectors
    index.add(train_embeddings.astype('float32')) # Add training embeddings to
index

    # 3. Query FAISS
    k = 5
    D, I = index.search(val_embeddings, k=k)

    # 4. Calculate Precision@1 and Precision@5
    correct_p1 = {c: 0 for c in range(len(CLASS_NAMES))}
    correct_p5 = {c: 0 for c in range(len(CLASS_NAMES))}
    total_queries = {c: 0 for c in range(len(CLASS_NAMES))}

    # Track examples for visualization: (query_path, [retrieved_paths],
query_label, [retrieved_labels])
    viz_examples = {c: [] for c in range(len(CLASS_NAMES))}

    for idx, (query_label, neighbors_indices) in enumerate(zip(val_labels, I)):
        total_queries[query_label] += 1
        neighbor_labels = [train_labels[ni] for ni in neighbors_indices]

        # Precision @ 1
        if neighbor_labels[0] == query_label:
            correct_p1[query_label] += 1

        # Precision @ 5
        if query_label in neighbor_labels:
            correct_p5[query_label] += 1

        # Collect examples for the grid (we need 3 per category)
        if len(viz_examples[query_label]) < 3:
            retrieved_paths = [train_image_paths[ni] for ni in
neighbors_indices[:3]]
            viz_examples[query_label].append((
                val_image_paths[idx], retrieved_paths, query_label,
neighbor_labels[:3]
            ))

    print("\nRetrieval Precision:")
    print(f"{'Category':<15} | {'P@1':<10} | {'P@5':<10}")
```

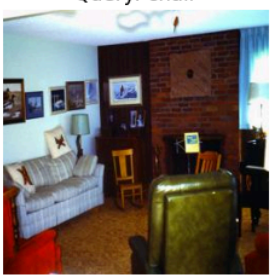
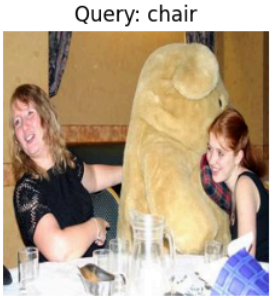
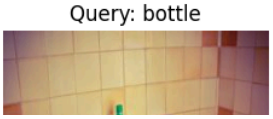
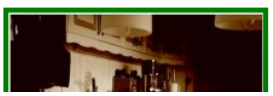
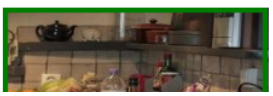
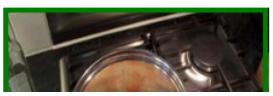
```
    print("-" * 40)
    for c in range(len(CLASS_NAMES)):
        p1 = correct_p1[c] / max(1, total_queries[c])
        p5 = correct_p5[c] / max(1, total_queries[c])
        print(f"{CLASS_NAMES[c]:<15} | {p1*100:.2f}%     | {p5*100:.2f}%")

    overall_p1 = sum(correct_p1.values()) / max(1, sum(total_queries.values()))
    overall_p5 = sum(correct_p5.values()) / max(1, sum(total_queries.values()))
    print(f"{'Overall':<15} | {overall_p1*100:.2f}%     |
{overall_p5*100:.2f}%")

    # 5. Visualization Grid (9 queries total: 3 per category)
    print("\nGenerating Retrieval Visualization Grid...")
    fig, axes = plt.subplots(9, 4, figsize=(12, 24))
    fig.suptitle('Image-to-Image Retrieval (Query vs Top-3 Retrieved)',
fontsize=16)

    row = 0
    for c_idx in range(len(CLASS_NAMES)):
        for example in viz_examples[c_idx]:
            q_path, ret_paths, q_label, ret_labels = example

            # Plot Query
            ax_q = axes[row, 0]
            ax_q.imshow(Image.open(q_path))
            ax_q.set_title(f"Query: {CLASS_NAMES[q_label]}")
            ax_q.axis('off')

            # Plot Top 3 Retrieved
            for i in range(3):
                ax_r = axes[row, i+1]
                ax_r.imshow(Image.open(ret_paths[i]))
                ax_r.axis('off')

                # Green border if correct, Red if incorrect
                is_correct = (ret_labels[i] == q_label)
                color = 'green' if is_correct else 'red'

                # Add bounding box to indicate correctness
                rect = patches.Rectangle((0, 0), 1, 1,
transform=ax_r.transAxes,
                                         linewidth=6, edgecolor=color,
facecolor='none')
                ax_r.add_patch(rect)
                ax_r.set_title(f"Match {i+1}")

            row += 1

    plt.tight_layout()
    plt.subplots_adjust(top=0.95)
    plt.savefig('retrieval_grid.png', bbox_inches='tight')
    print("Saved visualization grid to 'retrieval_grid.png'")
```

Per catefory Precision@1 and Precision@5:

```
Retrieval Precision:
Category        | P@1        | P@5
----------------------------------------
car             | 90.12%     | 97.63%
chair           | 78.12%     | 98.58%
bottle          | 69.43%     | 92.75%
Overall         | 79.82%     | 96.87%
```

Visualization of Image Retrieval Results:

Image-to-Image Retrieval (Query vs Top-3 Retrieved)

Query: car	Match 1	Match 2	Match 3
			
Query: car	Match 1	Match 2	Match 3
			
Query: car	Match 1	Match 2	Match 3
			
Query: chair	Match 1	Match 2	Match 3
			
Query: chair	Match 1	Match 2	Match 3
			
Query: chair	Match 1	Match 2	Match 3
			
Query: bottle	Match 1	Match 2	Match 3
			

Observations:

When we compare to a random baseline(33%), the CLIP based Precision@1 is significantly higher across all categories, with "car" achieving the highest P@1 (90%) and "bottle" the lowest (69%). The visualization grid shows that most of the top retrieved images are indeed correct matches (green borders), with a few incorrect ones (red borders), which aligns with the quantitative metrics.

4.1 Margin Ablation Study for Triplet Loss

We re-train the DLStudio EmbeddingGenerator1 using Task 2's Triplet Loss with different margin values: $\delta = 0.1, 0.5, 1.0$. We then evaluate the Precision@1 for each margin setting.

Code for Margin Ablation Study

```
def main():
    args = parse_args()
    set_seed(args.seed)

    output_dir = Path(args.output_dir)
    output_dir.mkdir(parents=True, exist_ok=True)

    metric_learner = build_dlstudio(
        epochs=args.epochs,
        batch_size=args.batch_size,
        learning_rate=args.learning_rate,
    )
    class_names = get_class_names(metric_learner)
```

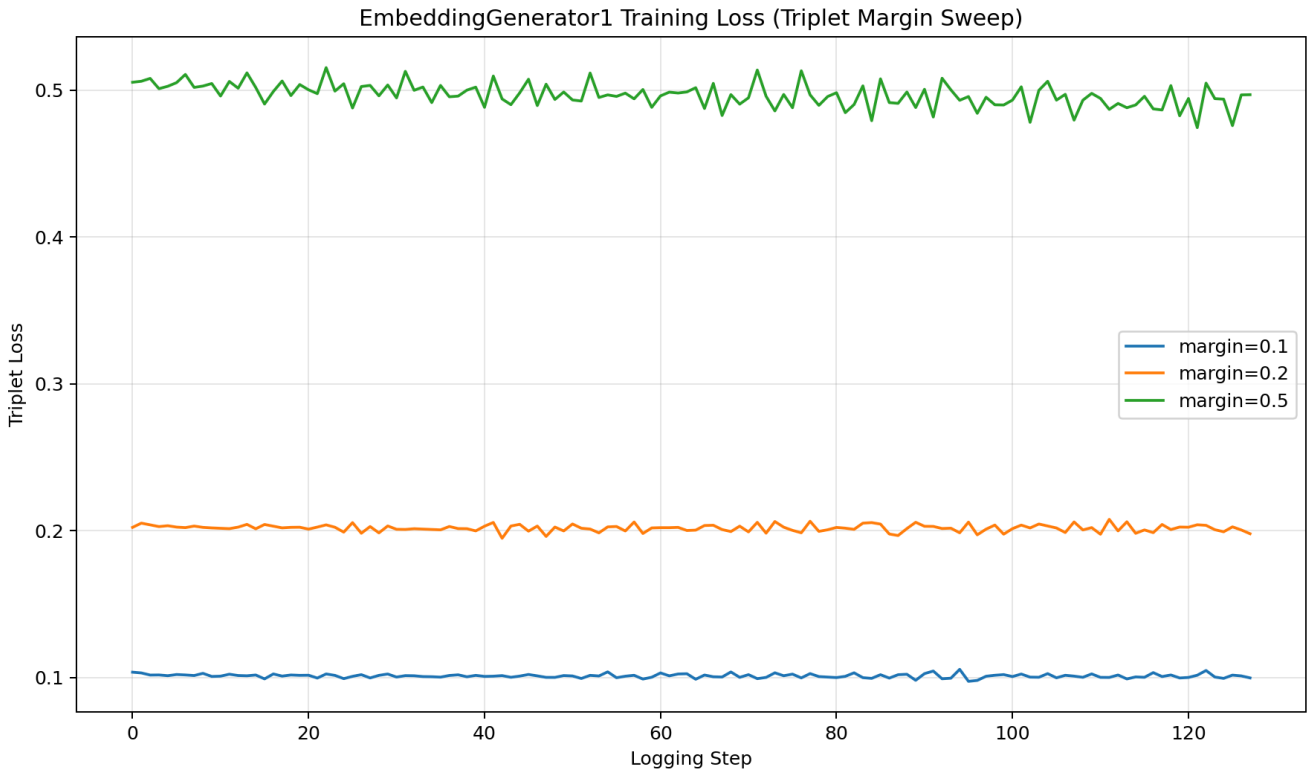
```
    loss_curves: dict[float, list[float]] = {}
    metrics: dict[str, dict] = {}

    for margin in args.margins:
        # Train and evaluate each margin independently.
        print(f"\n===== Training margin={margin:.2f} =====")
        model, losses = train_one_margin(
            metric_learner=metric_learner,
            margin=margin,
            log_every=args.log_every,
        )

        loss_curves[margin] = losses

        ckpt_path = output_dir /
f"embedding_gen1_triplet_margin_{margin:.2f}.pt"
        torch.save(model.state_dict(), ckpt_path)

        result = evaluate_precision_at_1(
            model=model,
            metric_learner=metric_learner,
            class_names=class_names,
            train_batches=args.train_eval_batches,
            test_batches=args.test_eval_batches,
        )
        metrics[f"{margin:.2f}"] = result

    fig_path = save_loss_plot(loss_curves, output_dir)
    json_path, csv_path = save_metrics(metrics, output_dir)
    print_metrics_table(metrics)

    print("\nSaved artifacts:")
    print(f"- Loss curves figure: {fig_path}")
    print(f"- Metrics JSON: {json_path}")
    print(f"- Metrics CSV: {csv_path}")
```

Output:

```
Precision@1 Summary (DLStudio-style subset evaluation)
================================================================================
Margin 0.10
Class            Correct    Total        P@1
------------------------------------------------
airplane              44       68     64.71%
automobile            44       50     88.00%
bird                  45       62     72.58%
cat                   36       59     61.02%
deer                  37       54     68.52%
dog                   43       66     65.15%
frog                  61       76     80.26%
horse                 50       61     81.97%
ship                  61       69     88.41%
truck                 68       75     90.67%
------------------------------------------------
OVERALL              489      640     76.41%
```

```
==========================================================================
Margin 0.20
Class            Correct    Total        P@1
------------------------------------------------
airplane              49       68     72.06%
automobile            48       50     96.00%
bird                  41       62     66.13%
cat                   34       59     57.63%
deer                  33       54     61.11%
dog                   40       66     60.61%
frog                  58       76     76.32%
horse                 51       61     83.61%
ship                  55       69     79.71%
truck                 62       75     82.67%
------------------------------------------------
OVERALL              471      640     73.59%
==========================================================================
Margin 0.50
Class            Correct    Total        P@1
------------------------------------------------
airplane              52       68     76.47%
automobile            44       50     88.00%
bird                  38       62     61.29%
cat                   29       59     49.15%
deer                  40       54     74.07%
dog                   34       66     51.52%
frog                  63       76     82.89%
horse                 44       61     72.13%
ship                  60       69     86.96%
truck                 58       75     77.33%
------------------------------------------------
OVERALL              462      640     72.19%
==========================================================================
```

Training Loss Curves for Different Margins:



Observations:

The margin δ controls how strongly the model enforces a gap between positive and negative pairs in the embedding space through $d(a, n) \geq d(a, p) + \delta$. A smaller margin allows more flexible local structure, while a larger margin pushes different classes farther apart and can also force tighter within-class clustering. However, if δ is too large, many triplets remain active for longer, optimization becomes harder, and the model may over-emphasize separation at the expense of preserving fine-grained neighborhood structure. In our CIFAR-10 results, this behavior is visible in Precision@1: $\delta = 0.10$ gave the best overall P@1 (76.41%), while $\delta = 0.20$ (73.59%) and $\delta = 0.50$ (72.19%) performed worse. This shows that a larger margin does **not** always improve retrieval performance. The practical takeaway is that margin is hyperparameter that needs to be tuned based on the dataset.

5.2 Master Summary Table

Network	Dataset	Loss / Setting	Key Metric
EmbeddingGenerator1	CIFAR-10	Pairwise Contrastive	P@1: 76%
EmbeddingGenerator1	CIFAR-10	Triplet	P@1: 82%
NCE Discriminator	2D Gaussians	BCE	Accuracy: 96.25% (best at N = 20)
CLIP ViT-B/32 (pretrained)	LVIS (3 cat)	Zero-shot classification	Zero-shot acc: 77.19%
CLIP ViT-B/32 (pretrained)	LVIS (3 cat)	Image-to-image retrieval	P@1 retrieval: 79.82%

1. Pairwise Contrastive Loss achieved 76% P@1 on CIFAR-10, while Triplet Loss reached 82% P@1, so triplet-based metric learning performed better in this setup. The gap is consistent with the intuition

that triplets provide a stronger relative ranking signal by explicitly comparing anchor-positive and anchor-negative relationships.

2. In the NCE experiments, increasing the number of negatives improved the discriminator quality, with the best BCE classification accuracy reaching 96.25% at N = 20. This supports the idea that richer negative sampling helps the model learn a sharper boundary between data and noise distributions.

3. For transfer learning, pretrained CLIP worked well without any fine-tuning, achieving 77.19% zero-shot classification accuracy. Retrieval behavior was also strong, with overall P@1 of 79.82%, showing that CLIP embeddings preserved useful semantic structure in the new domain.

4. Overall, these experiments show a consistent pattern: stronger contrastive supervision and richer negatives generally improve representation quality and downstream matching performance.