Metric learning, product quantization, approximate search

Jan Sedmidubský Masaryk University

sedmidubsky@mail.muni.cz

Outline

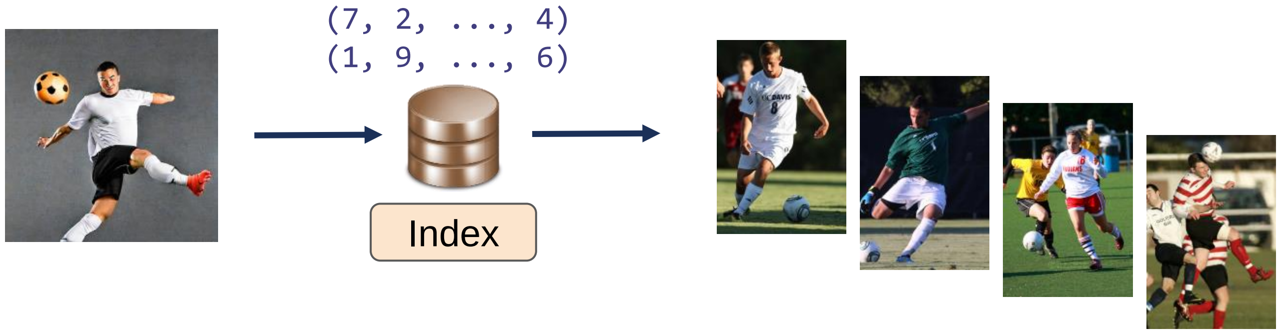
- Metric learning

→ `(0.6, 0.5, ..., 0.1)`

- Vector/Product quantization

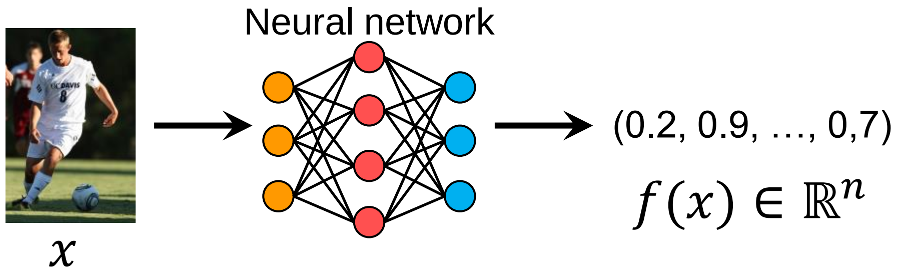
`(0.6, 0.5, ..., 0.1)` → `(7, 2, ..., 4)` (e.g., 64x compression)

- Approximate similarity search (e.g., using FAISS)



Metric learning

- Metric learning goal – representing data objects, such as images, text or whatever, with numerical vectors
  - Vectors = embeddings or embedding vectors
  - Function f transforms a given object (e.g., image x) into an *n*-dimensional vector $f(x) \in \mathbb{R}^n$



  - Former approach – individual features of the vector representation had to be manually specified
  - Current approach – the vector representation is learned automatically

Metric learning

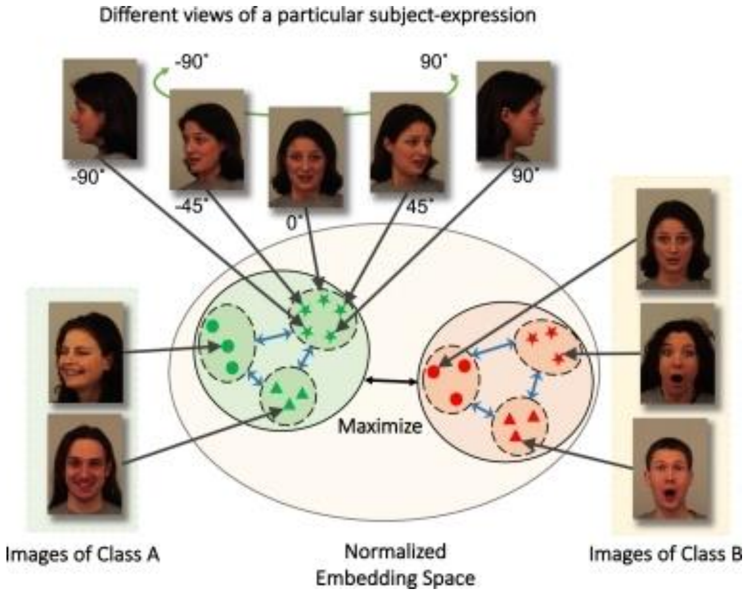
- Desired properties:
  - Similar data objects → vectors that are close together
  - Dissimilar data objects → vectors that are far apart
- Quantification of similarity/closeness:
  - Requires a distance measure in the underlying vector space
  - Commonly used measure – Euclidean distance function (L2 norm)
    - $dist\left(f(x), f(y)\right) = \|f(x) - f(y)\|_2$
- Metric learning process – pulling together the embeddings for similar objects and pushing apart those for dissimilar objects
- What exactly is meant by similar and dissimilar objects?

Metric learning

- Examples of similar and dissimilar objects on identity-based similarity:
    - Face recognition
    - Retail-product recognition
- Object identities (products, persons) lead to supervised clustering of the learned embeddings → why not just use a classifier?
    - Extreme classification – a very large number of classes (e.g., tens of thousands) with highly unbalanced training data
    - Stanford Online Products dataset (scraped from eBay) contains 120 K images for 23 K product classes
        - Output layer of some deep neural network with 23 K nodes and 4 images/class → you will get an unsatisfactory result

Metric learning

- Embedding vectors are:
  - As close together as they can be for the images in each class
  - As far as they can be from the embeddings for the other classes
- Example of recognition of facial expressions


Different views of a particular subject-expression
-90°
90°
-90°
-45°
0°
45°
90°
Maximize
Images of Class A
Normalized Embedding Space
Images of Class B

[Roy at al.: Contrastive Learning of View-invariant Representations for Facial Expressions Recognition, ACM TOMM 2023]

- Basic ideas in metric learning revolve around:
  - Pairwise contrastive loss
    - [Hadsell et al.: Dimensionality Reduction by Learning an Invariant Mapping, CVPR 2006]
  - Triplet loss
    - [Schroff et al.: FaceNet: A Unified Embedding for Face Recognition and Clustering, CVPR 2015]

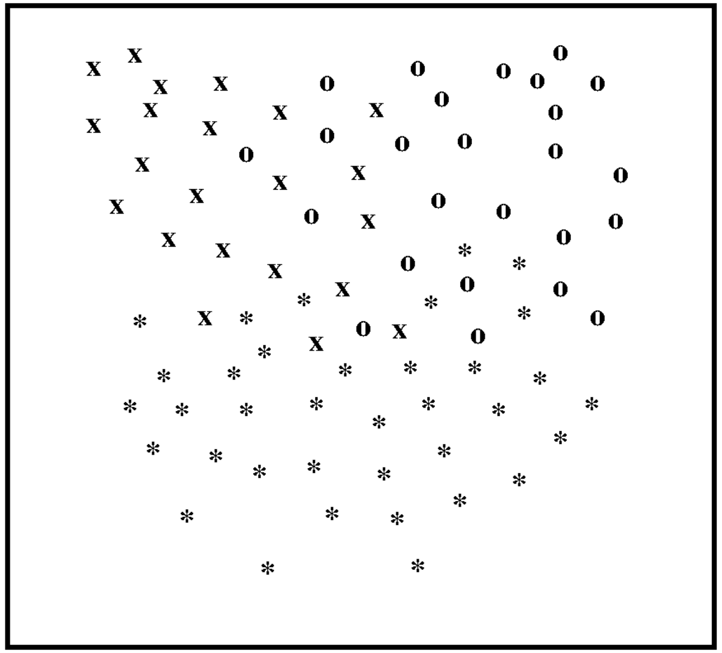
Pairwise Contrastive (PC) loss

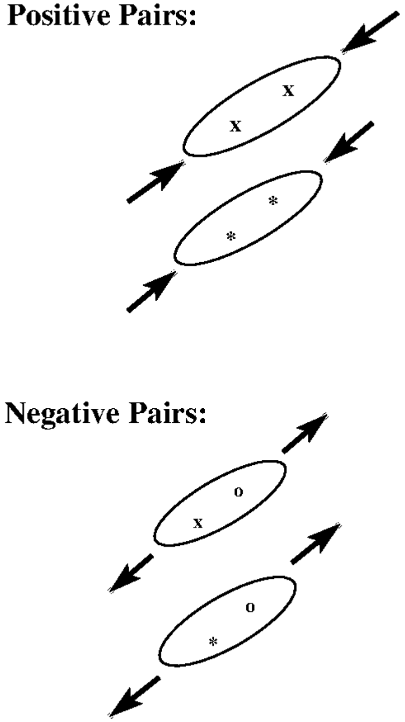
- Training a neural network in batches
- Goal – extract positive and negative pairs of training samples from a batch (batch – list of training samples)
  - Positive pairs – carry the same class label
  - Negative pairs – carry different labels

Positive pair

Negative pair

Three classes: x o *





PC loss – idea of calculation

- Loss (cost or objective) function L measures the discrepancy between the predicted output of the model and the actual target values
  - Purpose – to give the network feedback on how well it is performing so that it can adjust its parameters (weights and biases) to improve over time
  - During the training process, the loss should gradually decrease (up to 0)

- PC loss calculation:
  - A sum of the values calculated separately from positive pairs and negative pairs
  - Contribution to the loss by positive pairs (L_p) + contribution to the loss by negative pairs (L_n)

PC loss – positive pairs

- Contribution to the loss by positive pairs
  - Positive pair $\left(x_1^i, x_2^i\right)$ – pairwise distances as small as possible
  - Positive loss – sum over all positive pairs from the batch:

$$L_p = \sum_i \left[dist\left(f\left(x_1^i\right), f\left(x_2^i\right)\right)\right]^2$$

  - i – indexes all the positive pairs from the batch
  - Square of the distance because it is differentiable everywhere

PC loss – negative pairs

- Contribution to the loss by negative pairs
  - Negative pair (x_1^j, x_2^j) – pairwise distances as large as possible
    - *j* – indexes all the negative pairs from the batch
  - But very dissimilar items amount to wasting the learning effort
    - Two well-separated samples in a negative pair should not even participate in learning
    - Threshold on maximal dissimilarity quantified by margin m
      - If $dist(x_1^j, x_2^j) > m$ then the contribution to the loss should be 0
      - If $dist(x_1^j, x_2^j) \leq m$: $L_n = m - dist(x_1^j, x_2^j)$

  - Negative loss – sum over all negative pairs from the batch:

$$L_n = \sum_j \left[\max\left\{0, m - dist\left(f(x_1^j), f(x_2^j)\right)\right\}\right]^2$$

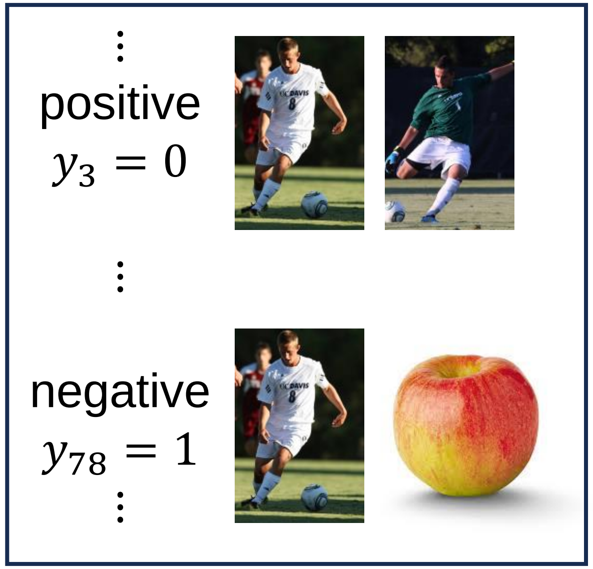
PC loss

- Overall loss for all pairs in batch by combining L_p and L_n
  - Binary variable $y \in \{0,1\}$:
    - $y = 0$ → positive pair
    - $y = 1$ → negative pair

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

  - i – goes over all the pairs from the batch

Batch example



⋮

positive

$y_3 = 0$

⋮

negative

$y_{78} = 1$

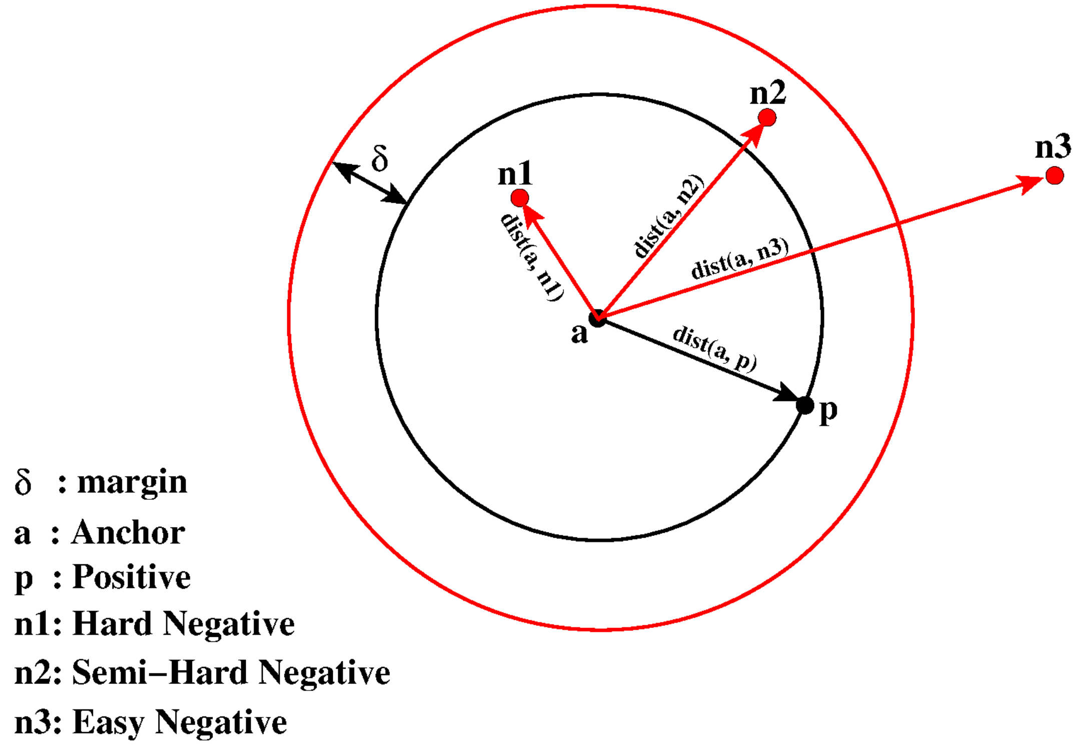
⋮

Triplet loss

- Creating triplets (Anchor, Positive, Negative) from a batch
  - (Anchor, Positive) – carry the same class label
  - (Anchor, Negative) – carry different labels
- Different mining strategies with different computational properties:
  - Negative-hard mining
  - Negative semi-hard mining

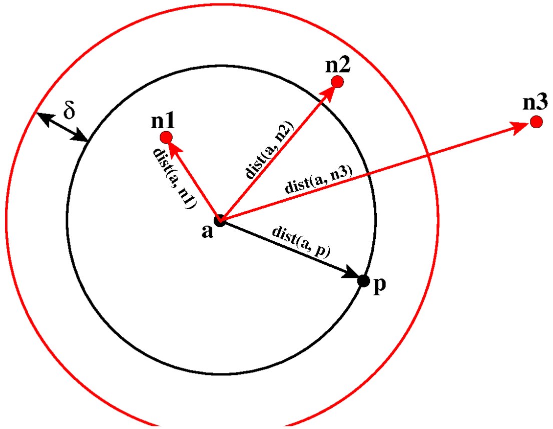
Triplet loss – creating triplets

- For every pair having the same class label:
  - One selected as Anchor, the other as Positive: (Anchor, Positive)
  - For every (Anchor, Positive) pair:
    - Negative objects are identified – objects with a different class than Anchor/Positive
      - $n1$ (hard negative) – must be pushed further out
      - $n2$ (semi-hard negative)
      - $n3$ (easy negative)



Triplet loss – determining negatives

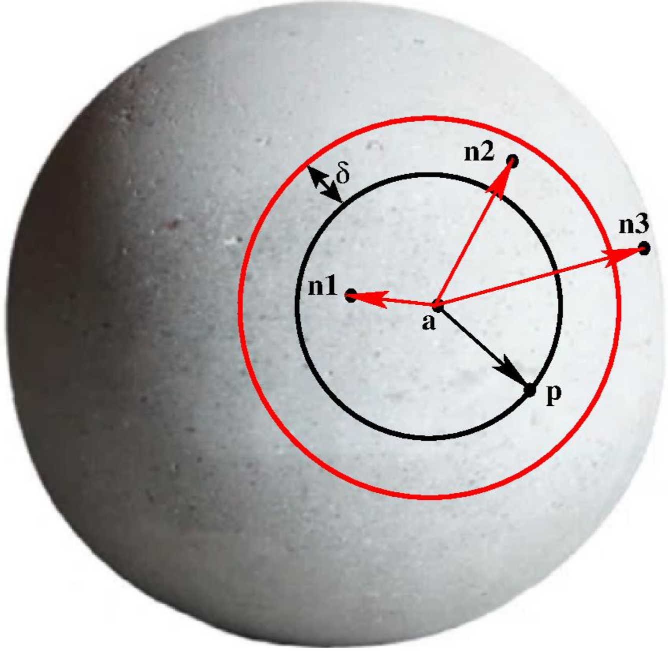
- Criteria for dividing a set of negatives:
  - Hard-negative mining ($n \sim n1$): negatives closer to anchor than positive
    - $dist(a, n) < dist(a, p)$
  - Semi-hard negative mining ($n \sim n2$): negatives fall within margin δ
    - $dist(a, p) < dist(a, n) < dist(a, p) + \delta$
  - Easy negatives ($n \sim n3$): negatives that lie beyond the margin
    - $dist(a, p) + \delta < dist(a, n)$



- Suitability of negatives for training:
  - Only easy negatives for training – insignificant role in learning, if any at all
  - Only hard negatives for training – network:
    - Converges to a local minimum at best, or
    - Collapses to a state in which all the embeddings are zero
  - The most suitable for training are semi-hard negatives

Triplet loss – determining negatives

- Semi-hard negative mining ($n2$)
  - Negatives sufficiently close to anchor
  - Margin δ has to be carefully set:
    - Appropriate value – network correctly distinguishes between positive/negative samples
    - Too large/small value – network optimization process may get stuck in a local minimum
  - To properly set the margin, all embeddings are normalized to be of size unity
    - Margin typically set to δ $= 0.2$

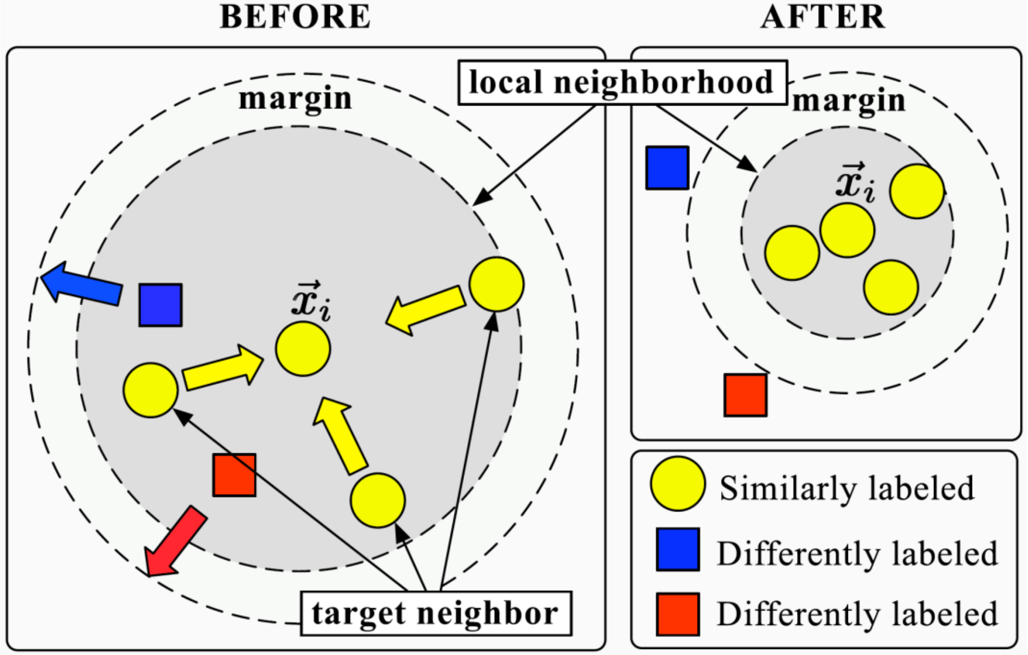


Triplet loss – calculation

- List of triplets (x_i^a, x_i^p, x_i^n) of a batch of cardinality N

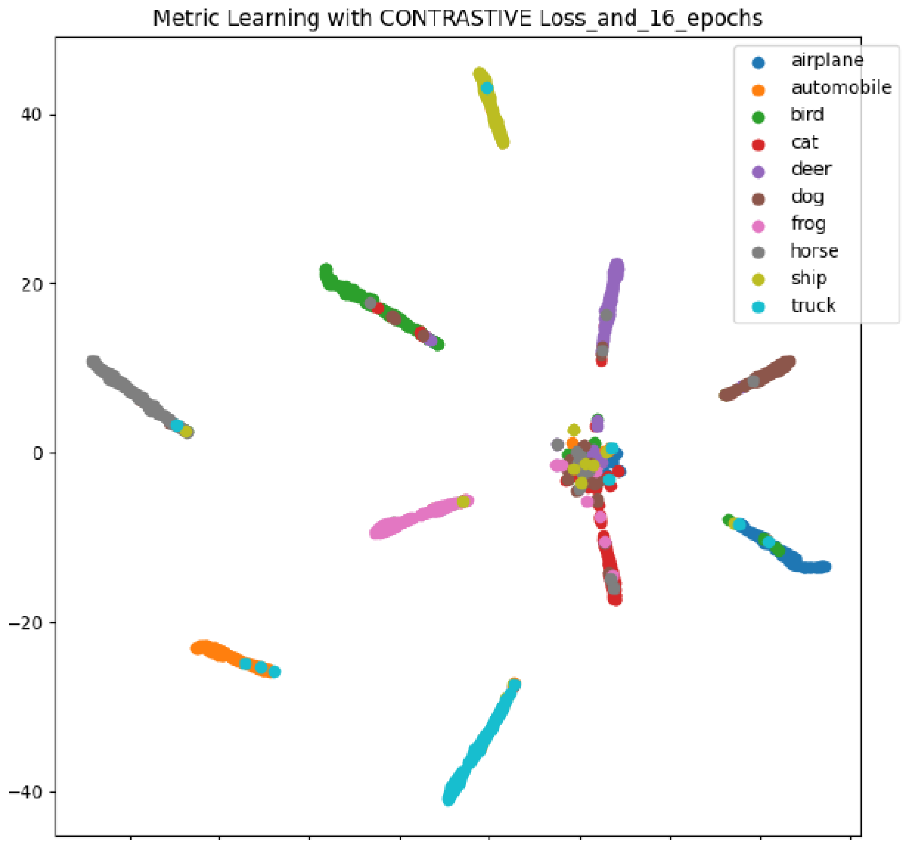
$$L = \sum_{i=1}^{N} \max\{0, dist^2(f(x_i^a), f(x_i^p)) - dist^2(f(x_i^a), f(x_i^n)) + \delta\}$$

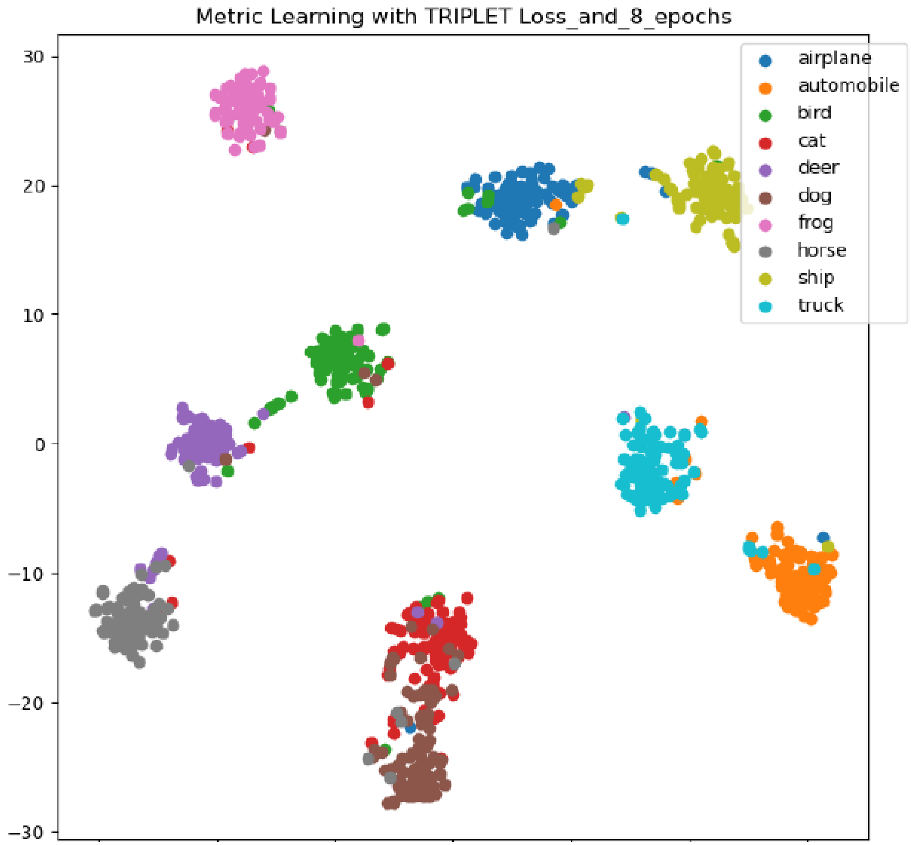
  - No contributions from the negatives that are outside the margin (*max* returns 0)



Results on CIFAR-10

- Training on the CIFAR-10 dataset
  - Contrastive loss learning: Precision@1 = 74%
  - Triplet loss learning: Precision@1 = 84%





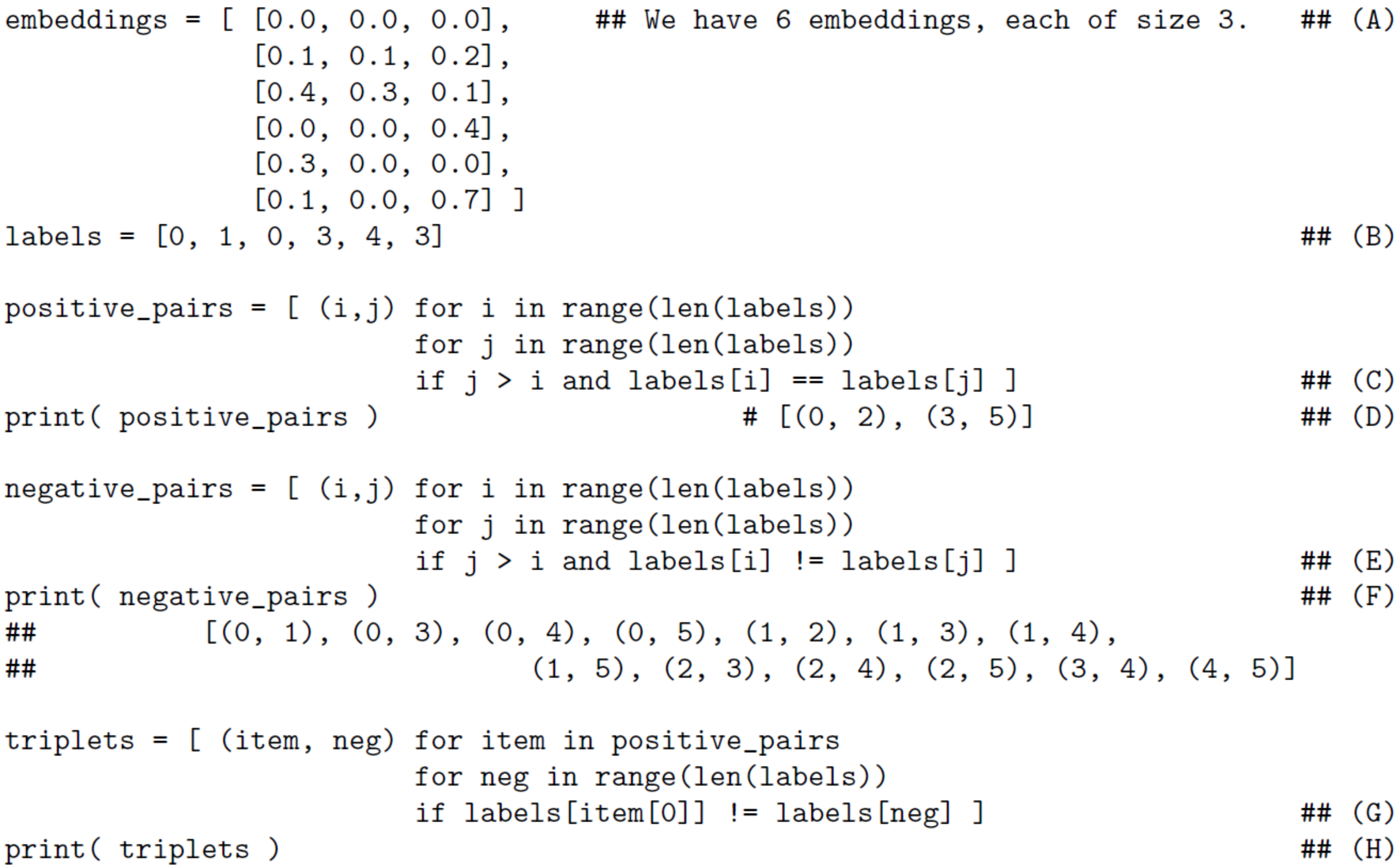
Coding issues

- Goal – determine pairs/triplets → calculate pair-wise distances
- Variables:
  - Batch of size B
  - Dimensionality of embeddings: M
  - Embeddings-data array X of shape $B \times M$
  - Labels of embeddings: $B \times 1$

- Easy solution – iterative processing (for-loops) to determine distances
  - The cost of iterative processing is simply too much great
- GPU solution – matrix multiplications – not a friend with for-loops
  - Thousand-fold speedup when you eliminate the loops that you would otherwise need for estimating

Coding issues

- Easy solution – iterative processing

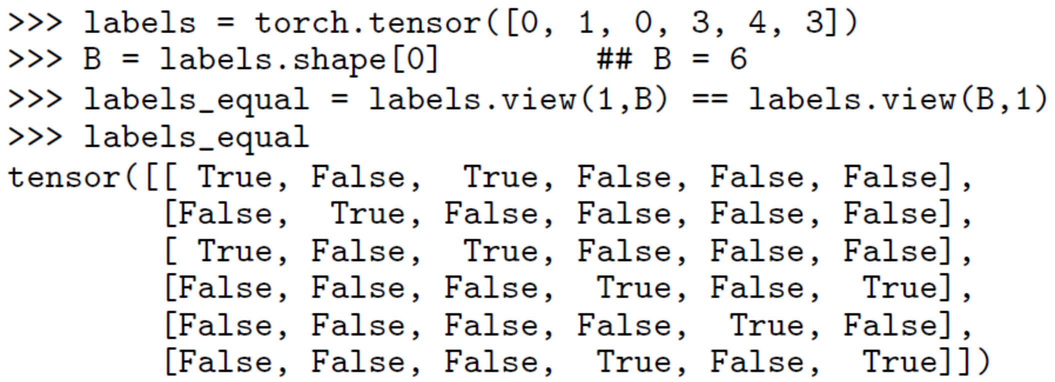


```
embeddings = [ [0.0, 0.0, 0.0],     ## We have 6 embeddings, each of size 3.   ## (A)
               [0.1, 0.1, 0.2],
               [0.4, 0.3, 0.1],
               [0.0, 0.0, 0.4],
               [0.3, 0.0, 0.0],
               [0.1, 0.0, 0.7] ]
labels = [0, 1, 0, 3, 4, 3]                                                    ## (B)

positive_pairs = [ (i,j) for i in range(len(labels))
                         for j in range(len(labels))
                         if j > i and labels[i] == labels[j] ]                 ## (C)
print( positive_pairs )                      # [(0, 2), (3, 5)]                ## (D)

negative_pairs = [ (i,j) for i in range(len(labels))
                         for j in range(len(labels))
                         if j > i and labels[i] != labels[j] ]                 ## (E)
print( negative_pairs )                                                        ## (F)
##          [(0, 1), (0, 3), (0, 4), (0, 5), (1, 2), (1, 3), (1, 4),
##                              (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (4, 5)]

triplets = [ (item, neg) for item in positive_pairs
                         for neg in range(len(labels))
                         if labels[item[0]] != labels[neg] ]                   ## (G)
print( triplets )                                                              ## (H)
```

Coding issues

- GPU solution – matrix multiplications
  - Implementation based on tensors
  - In case of iterative processing: if batch size is 128, this results in 8,192 fetches from the GPU memory ($128^2/2$)
  - Solution using 1 GPU fetch → >8 K speedup

    1) Determining pairs using the structure with labels (of B dimensionality):



```
>>> labels = torch.tensor([0, 1, 0, 3, 4, 3])
>>> B = labels.shape[0]        ## B = 6
>>> labels_equal = labels.view(1,B) == labels.view(B,1)
>>> labels_equal
tensor([[ True, False,  True, False, False, False],
        [False,  True, False, False, False, False],
        [ True, False,  True, False, False, False],
        [False, False, False,  True, False,  True],
        [False, False, False, False,  True, False],
        [False, False, False,  True, False,  True]])
```

Coding issues

- GPU solution – matrix multiplications

  2) Calculating the distance matrix:

  - Euclidean distance between vectors $\vec{x}$ and $\vec{y}$: $\|\vec{x}-\vec{y}\|_2 = \|\vec{x}\|_2 - 2\vec{x}\vec{y}^T + \|\vec{y}\|_2$
    - The square of the norm of each of the vectors
    - The value of the dot product between the two vectors

  - Calculating a dot product of every pair of embedding vectors in *X*:

    $$dot\_products = X@X.T$$

    - Vector norms for the embedding vectors are on diagonal

      $squared\_norms\_embedding\_vecs \; = \; torch.diagonal(dot\_products)$
    - Dot products between pairs of embedding vectors are in the off-diagonal elements

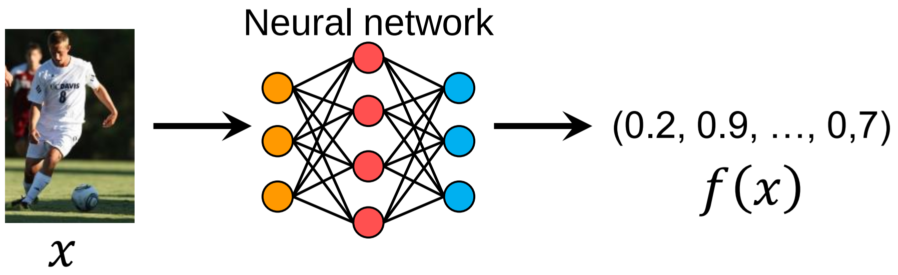
  - Calculating the Euclidean distance matrix $B \times B$ $(B = 6)$:

    $$\begin{aligned} &distance\_matrix \\ &= squared\_norms\_embedding\_vecs.view(1,6) - \; 2.0 \cdot dot\_products \\ &+ \; squared\_norms\_embedding\_vecs.view(6,1) \end{aligned}$$

    - Note: operators for tensors are overloaded to add (or subtract) three tensors of different shapes

Image embeddings – what we know

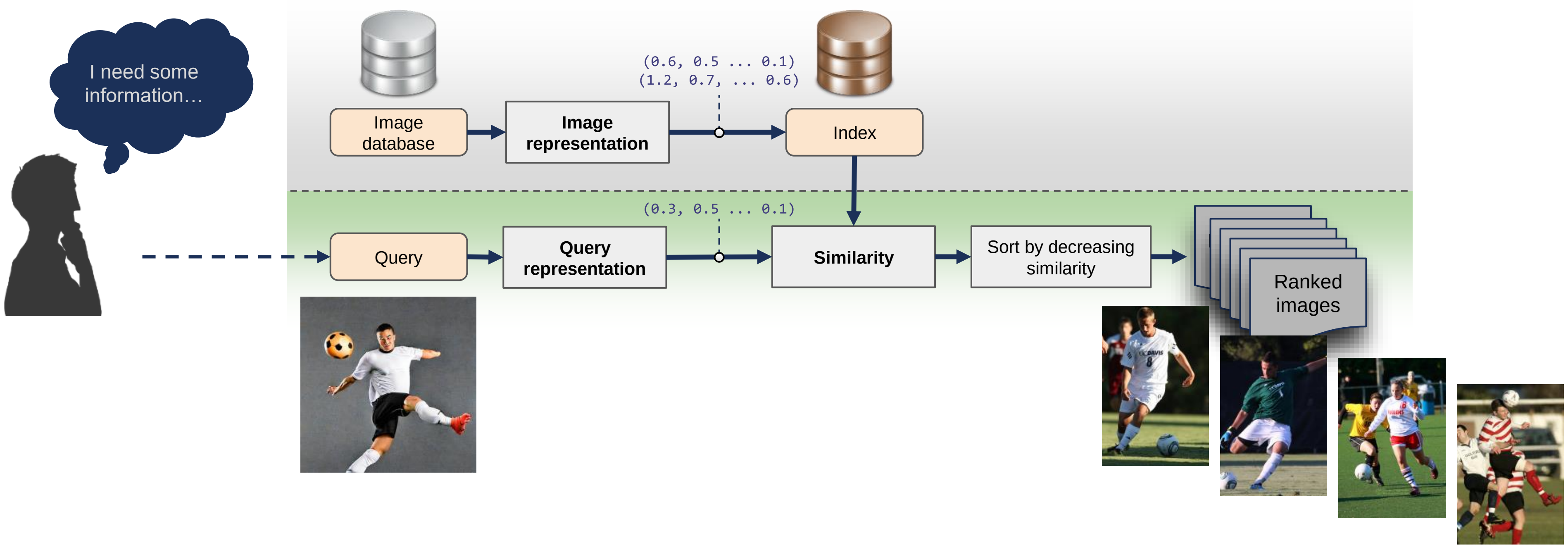
- Training a neural network based on PC/Triplet loss learning
- Transform query and each database image into an embedding
  - Mapping function $f()$ extracting embedding $f(x)$ for object x



  - Given two images x and y, their closeness can be quantified by the Euclidean distance: $dist\big(f(x), f(y)\big) = \|f(x) - f(y)\|_2$

Search over image embeddings

- Query image ~ query embedding ~ query
- *k*-nearest neighbor (*k*-NN) query
  - Finding the *k* database images that are the most similar to the query image
  - Similarity between query and database images based on the Euclidean/cosine distance between their embeddings



Search over image embeddings

- Many applications of image search, e.g.:
  - Photo organization – grouping images into albums automatically by recognizing similar faces, locations, or objects
  - Fashion and e-commerce – outfit pairing in online stores
    - Recommending matching items (e.g., shoes, bags, or accessories) by comparing product images based on style, color, and texture
    - Visual similarity in product search – enabling users to upload an image of a product to find similar items in the store
  - Cultural heritage and art preservation
    - Artifact identification – comparing images of newly found artifacts with existing ones to determine origin or classification
    - Style similarity matching – finding paintings with similar styles for study or curation
  - Duplicate image detection – removing similar or exact copies of an image in large databases to ensure unique content

Search over large image databases

- Brute-force approach:
  - Comparing the query embedding against each database embedding
  - $O(N)$ complexity (N is the size of the dataset) – not scalable
  - How to search when the database of embeddings is very large?

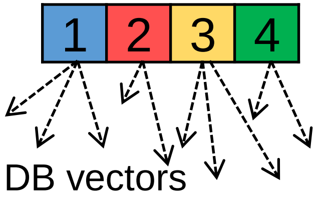
- Solution – Approximate Nearest Neighbor (ANN) algorithms
  - Many algorithms but leading ones are:
    - Locality Sensitive Hashing (LSH) – PA212 course
    - Product Quantization (PQ)

Product Quantization (PQ)

- PQ – extension of the very old Vector Quantization (VQ) idea
- VQ idea:
  - Compressing a high-dimensional image vector (embedding) into a single codeword, typically an integer value

    (0.2, 0.9, …, 0,7) → $\boxed{4}$

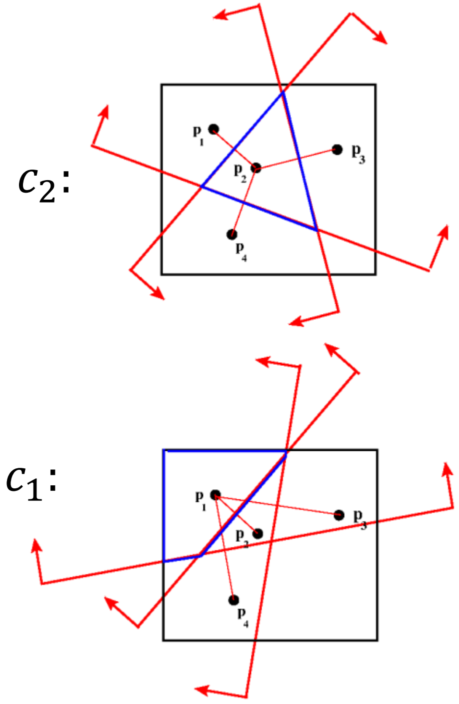
  - Pre-processing phase:
    - Create a mapping from all codewords to the original image vectors within a database
      - Each codeword points to the list of all the database images with the same codeword
      - This mapping structure is referred to as the lookup table or inverted index



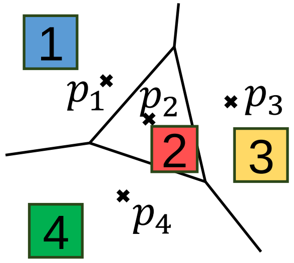
  - Search phase:
    - Transform the query into a single codeword
    - Use the lookup table to get the candidate image vectors of the same codeword as the query
      - It is also possible to consider candidate vectors having the codeword "relevant" (not strictly the same)
    - Compute the distance between the original query vector and all candidate vectors
    - Return the *k*-nearest candidate vectors with respect to the query vector

VQ

- Compression idea:
  - Partitioning a vector space into Voronoi cells with respect to a set of points
    - Points are typically called the centroids (pivots) of the cells in which they reside
    - Voronoi diagram – partitioning the *D*-dimensional space of embeddings into cells $\{c_1, c_2, \dots, c_n\}$ determined by pivots $\{p_1, p_2, \dots, p_n\}$; each cell c_i gets its ID ~ codeword
      - All the vectors in cell c_i are closer to the pivot p_i than to any other pivot



An example of the Voronoi diagram for the case of four pivots $\{p_1, p_2, p_3, p_4\}$ in the 2D plane



VQ

- Pre-processing phase:
  - Creating a codebook of $n = 2^B$ codewords
    - The n centroids (~cells) are typically the K centroids generated by applying the K-means algorithm to the (sample of) database of vectors
      - Any vector can be quantized in the underlying vector space to one of the K cluster centers
      - Vectors that fall in the same cluster will be mapped to the same codeword
      - Clusters can be differently populated based on data distribution
    - Each codeword is represented by an integer value → *B*-bit representation
  - Managing a lookup table with 2^B codewords
    - Transforming each database image vector into a codeword
    - Each codeword in the lookup table is associated with a list of the database images (or paths to these images) of the same codeword
  - Dimensionality of data can be dramatically reduced – example scenario:
    - Each image represented by a 512-D embedding vector of floats → $512 \cdot 4 = 2{,}048$ bytes
    - $B = 16$ → 512-D vectors quantized to $2^{16} = 65{,}536$ codewords
    - Each image then represented by the 16-bit code (2 bytes) → 1,024x compression

VQ

- Search phase:
  - Before the query is transformed to codeword, the ranking of pivots is created
    - The distance between the query vector and each pivot vector must be calculated
  - The query gets the codeword corresponding to the nearest (most-ranked) pivot
    - Other “query-relevant” codewords can also be considered, e.g., as 2nd or 3rd most-ranked
  - The database vectors associated with the same codeword as the query codeword (or any of “query-relevant” codewords) become candidates
  - The distance between the query vector and each candidate is evaluated
    - Each candidate vector must be loaded, e.g., from secondary storage
  - The *k*-most similar candidates (with the smallest distance) are returned as the query result

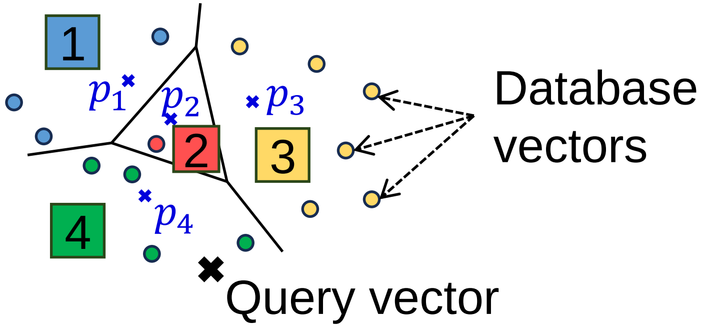


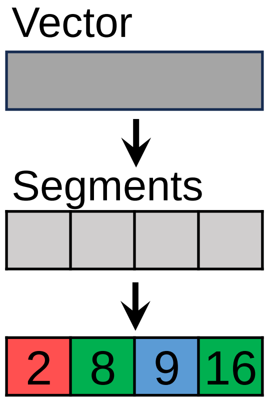
Illustration of query evaluation:

- Query vector → 4
- Candidates are database vectors ● associated with codeword 4

VQ

- Limitations of VQ:
  - If the codebook is too large (e.g., $B = 64$ → 2^{64} clusters are needed to be found), it is impossible for K-means to detect such a huge number of clusters
  - Returning the *k*-most query relevant database vectors requires to load a large set of candidate vectors and calculate their distance with respect to the query

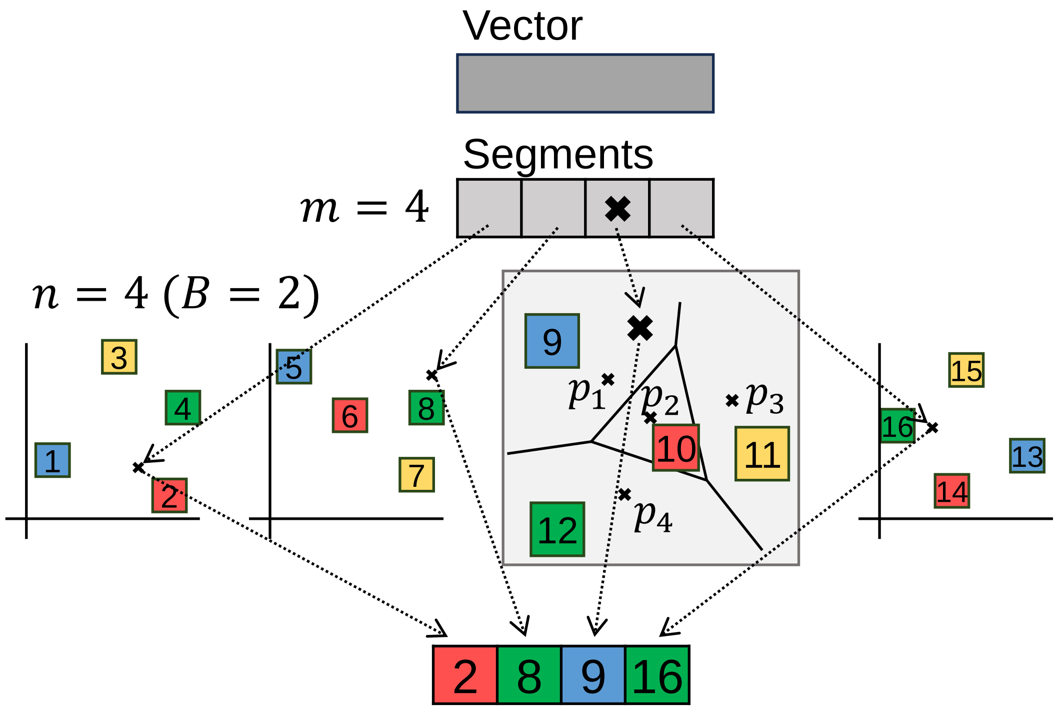
PQ



- Product quantization (PQ) idea:
  - Original database vector split into several segments (sub-vectors)
    - Each sub-vector follows the vector quantization independently
    - Quantized vector – concatenation of codewords of individual segments
  - Pre-processing phase:
    - Create a sub-quantizer – a codebook for each of the segments – separately (i.e., #codebooks = #segments)
      - Clustering operation applied to each set of sub-vectors
    - Create a mapping from all codewords of each codebook to the original image vectors within a database
  - Search phase:
    - Original database vectors need not be loaded (e.g., from secondary storage)
    - Distance between the query and a database vector is efficiently approximated
      - Based on pre-computed distances between the query segments and centroids in each codebook

PQ

- Pre-processing phase:
  - Creating m segments → m codebooks, each of $n = 2^B$ codewords
    - Each codeword is again represented by an integer value → B-bit representation
  - Codewords of all segments are concatenated →$(m \cdot B)$-bit representation
  - Managing a lookup table with 2^B codewords for each segment (m lookup tables)

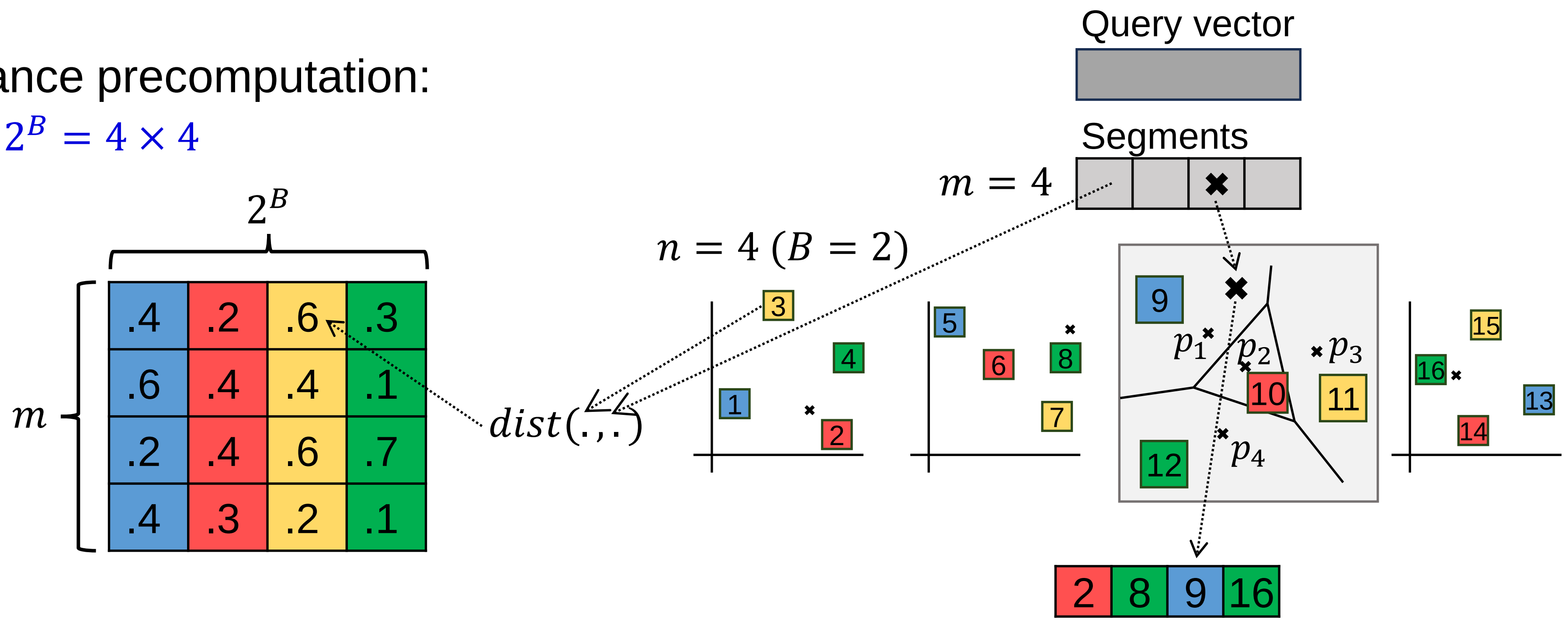


Example scenario:

- Image as a 512-D vector of floats → $512 \cdot 4 = 2{,}048$ bytes
- $m = 32$ → 32 segments (codebooks)
- $B = 8$ → 16-D sub-vectors quantized to $2^8 = 256$ codewords
- Each image then as $(32 \cdot 8)$-bit code (32 bytes) → 64x compression

PQ

- Search phase:
  - 1) Precompute the distances between each query sub-vector and each centroid in the corresponding codebook → $m \cdot 2^B$ distances in total
    - Distances kept within a query distance matrix QDM of size $m \times 2^B$
    - E.g., $32 \cdot 256 = 8{,}192$ distances in the previous example scenario, which is cheap to compute
    - Illustration of distance precomputation:
      - QDM of size $m \times 2^B = 4 \times 4$
        - $m = 4, B = 2$



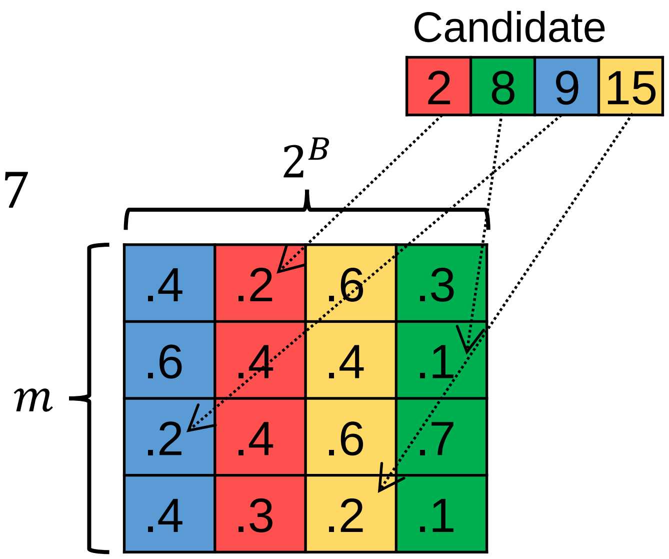
.4	.2	.6	.3
.6	.4	.4	.1
.2	.4	.6	.7
.4	.3	.2	.1

PQ

- Search phase:
  - 2) Identify the nearest cluster(s) in the same way as in the VQ approach
  - 3) Approximate the distance of each candidate vector in the nearest cluster(s):
    - For i-th segment and associated codeword c_i of the candidate vector, approximate the sub-distance between i-th query segment and i-th candidate segment by the precomputed sub-distance $QDM[i, c_i]$
    - Sum the sub-distances for all the segments: $\sum_{i=0}^{m-1} QDM[i, c_i]$

Illustration of approximating the distance (for query 2 8 9 16):

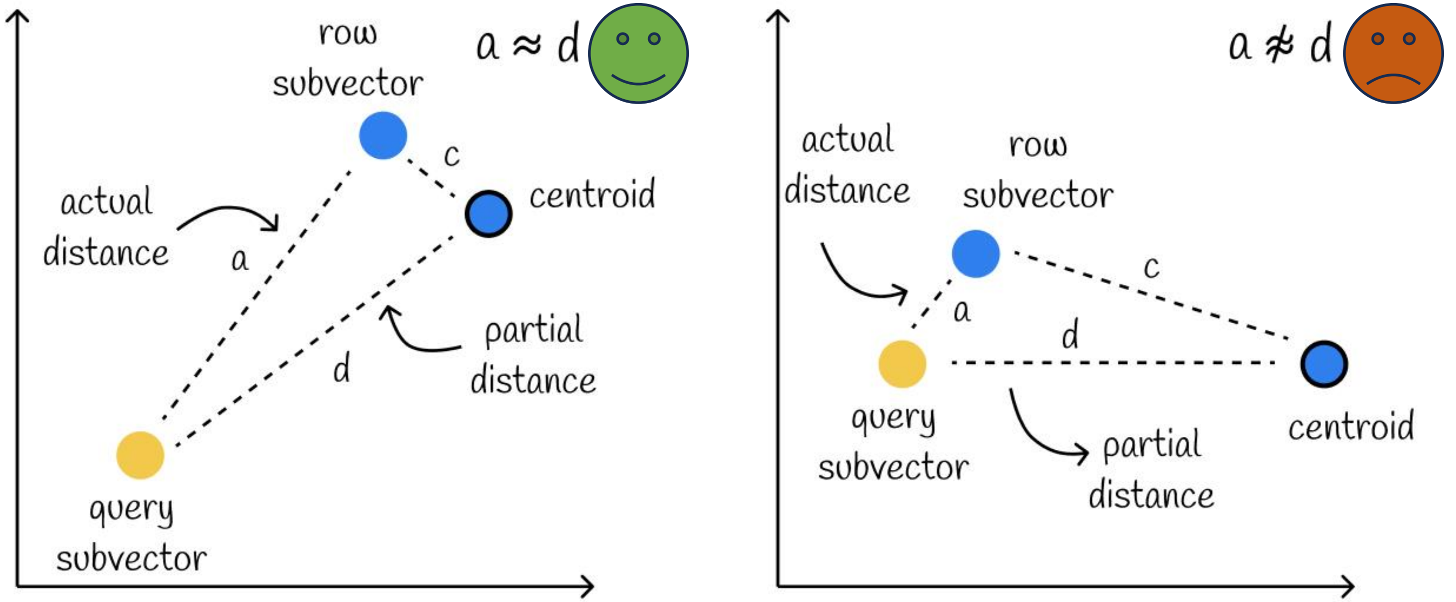
- Candidate 2 8 9 15 : $\sum_{i=0}^{3} QDM[i, c_i] = 0.2 + 0.1 + 0.2 + 0.2 = 0.7$
- Candidate 2 8 12 15 : $0.2 + 0.1 + 0.7 + 0.2 = 1.2$
- Candidate 1 8 9 16 : $0.4 + 0.1 + 0.2 + 0.1 = 0.8$



.4	.2	.6	.3
.6	.4	.4	.1
.2	.4	.6	.7
.4	.3	.2	.1

PQ

- Summary:
  - For distance approximations, the candidate vectors are not accessed
    - Accessing candidate vectors can be bottleneck in VQ, especially when vectors are stored within secondary storage
    - ANN search carried out efficiently in high dimensional vector spaces even when a database has billions of vectors
  - Approximated distances need not be perfect



Similarity search using FAISS

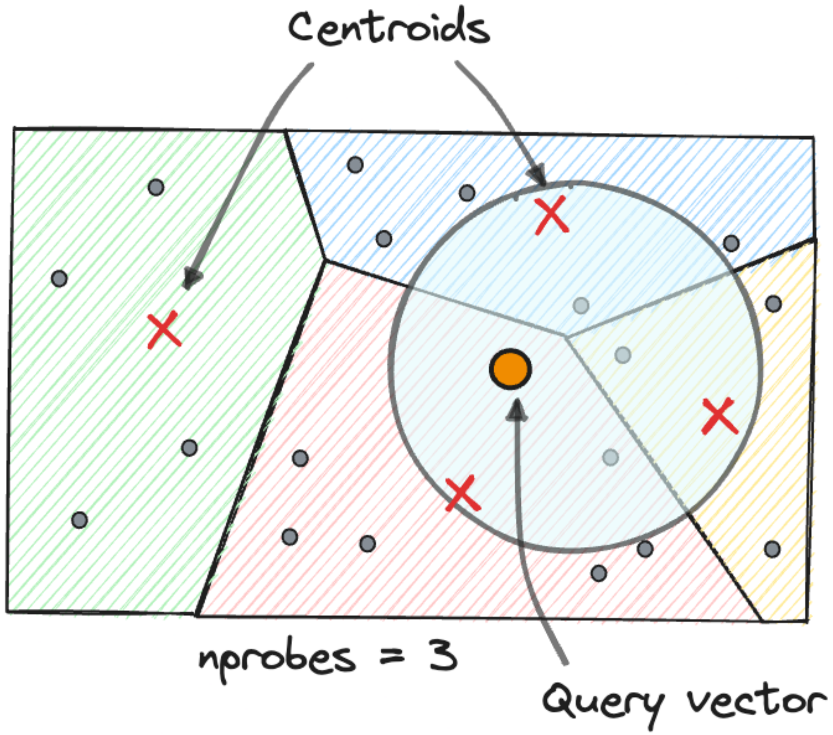
- FAISS – Facebook AI Similarity Search
- Library developed by Facebook AI Research for efficient similarity search and clustering of dense vectors
- Useful for large-scale similarity search problems, which are common in various machine learning and information retrieval tasks
- Designed to work on either the GPU or CPU and provides significant performance improvements compared to other nearest neighbor search algorithms
- One of the best implementation of the Product Quantization approach to similarity search
- Implemented in C++ with Python bindings

FAISS

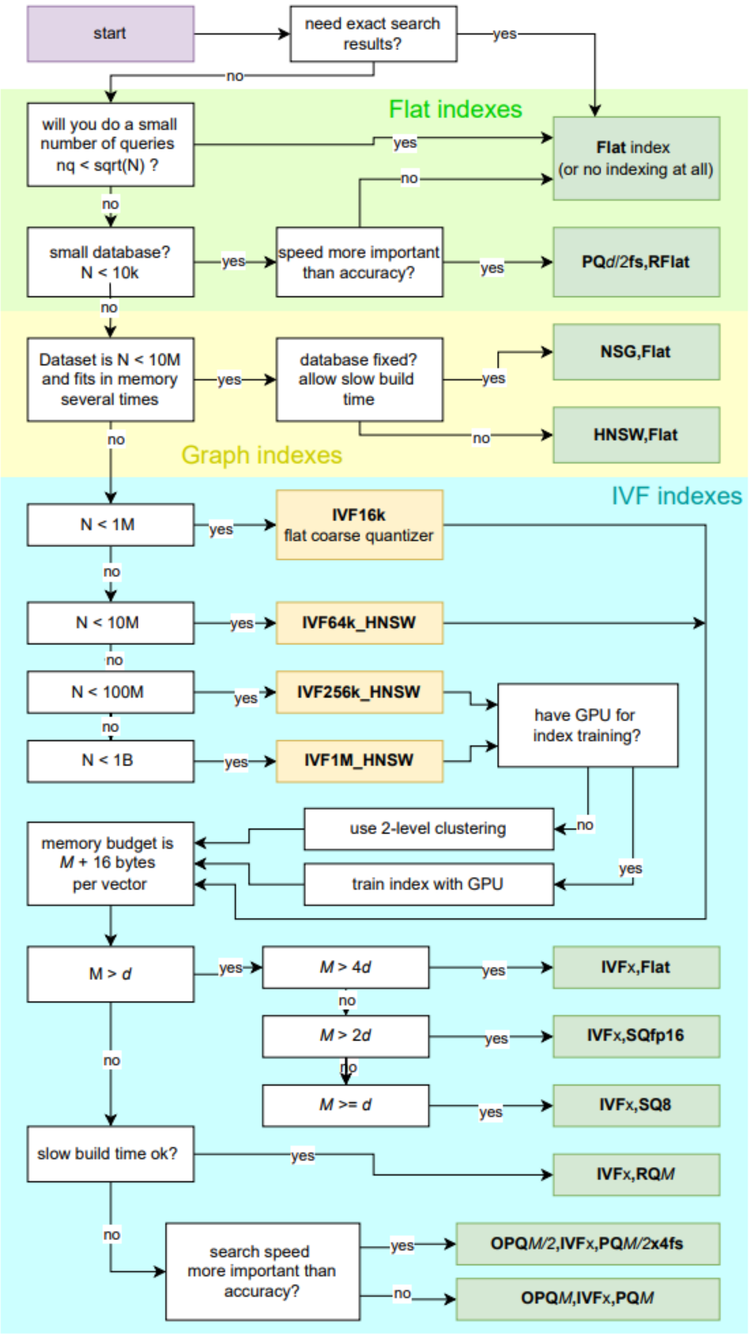
- Several techniques to achieve efficient similarity search:
  - Quantization – compresses the embeddings which significantly reduces memory usage and accelerates distance computations
    - Supports Product Quantization (PQ)
  - Indexing – FAISS provides multiple index types for different use cases and trade-offs between search speed and search quality
    - Flat index – brute-force index that computes exact distances between query vectors and indexed vectors
    - IVF (Inverted File) index – partitioned index that divides the vector space into Voronoi cells
    - HNSW (Hierarchical Navigable Small World) – graph-based index that builds a hierarchical graph structure, enabling efficient nearest neighbor search with logarithmic complexity

FAISS

- Example of IVF (Inverted File) index:
  - *nprobes* parameter specifies the number of nearest cluster(s) to be visited
  - Clusters ranked by the distance between the cluster centroid and query vector



Index selection guidelines



FAISS

- Useful references
  - Tutorials:
    - https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/
    - https://github.com/facebookresearch/faiss/wiki/
  - Research papers:
    - Johnson et al.: Billion-scale similarity search with GPUs, 2017: https://arxiv.org/abs/1702.08734
    - Douze et al.: The FAISS library, 2024: https://arxiv.org/abs/2401.08281

Sources

- Avi Kak and Charles Bouman: Metric Learning with Deep Neural Networks. Purdue University, 2024
- https://www.pinecone.io/learn/series/faiss/faiss-tutorial/