Metric Learning with Deep Neural Networks

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Tuesday 5th March, 2024 11:20

Preamble

[To TOC]

Metric learning is about representing your data objects, such as images or text or whatever, with numerical vectors such that the data objects that are similar acquire vector representations that are close together. By the same token, we would want the dissimilar data objects to acquire representations that are far apart.

In keeping with the use of the word "embedding" in the earlier lectures, we can refer to such vectors as "embeddings" or "embedding vectors" because you are embedding the data objects in a vector space.

Since pulling together the embeddings for similar objects and pushing apart those for dissimilar objects is obviously fundamental to metric learning, it requires a distance metric in the underlying vector space. The most commonly used metric for this purpose is the Euclidean metric — the L_2 norm.

At this point, you might ask: What exactly is meant by "similar" and "dissimilar" objects?

Preamble

In the domain of face images — a problem domain that has proved fertile for the development of metric learning algorithms — we would want to consider all the face images of the same human subject to be similar *regardless of the pose of the head vis-a-vis the camera*.

You may think of the above-mentioned sense of similarity as one that is based on the identity label associated with a face. Such a definition of similarity is obviously appropriate in the domain of, say, face verification since we would want to cluster together the face images for the same human subject and pull as far as apart as possible the face images of two different subjects regardless of the head poses involved.

In practice, the images of a given face from all possible viewpoints do not form a single cluster, but multiple clusters that reside on a manifold as explained in our following publication:

https://engineering.purdue.edu/RVL/Publications/FaceRecognitionUnconstrainedPurdueRVL.pdf

That fact does not make irrelevant the importance of metric learning for solving problems like face verification — you'd want to reduce the distances between the embeddings for the face images that fall in each single single cluster of a multi-modal representation on a manifold.

Preamble

Extending the notion of identify based similarity to other areas where metric learning has proved useful, consider the domain of retail products. This domain is relevant for applications such as automated check-out systems for supermarkets and department stores. With a metric learning based framework you would want the embeddings for a bag of potato chips for its views from different angles to cluster together as tightly as possible. For that reason, you would consider all those views to be similar. This is another example of identity based similarity.

But if our similarity and dissimilarity distinctions are driven by object identities, that means we are obviously talking about supervised clustering of the learned embeddings. In addition, if the learning by the neural network is going to be supervised, **why not just use a classifier?**

As for why what you get with metric learning goes beyond the capabilities of a classifier, consider the case of what has come to be known as *extreme classification* where you have a very large number of classes — of the order of tens of thousands — with highly unbalanced training data.

Preamble

For example, the Stanford Online Products dataset (scraped from eBay) contains 120,000 images for 23,000 product classes. If you designed a neural-network classifier along the lines discussed in, say, my Week 4 lecture, your output layer will have 23,000 nodes and the network weights would need to be learned from a tiny number (on the average just 4) of images per class. It's highly unlikely that you will get a satisfactory result.

In a solution based on metric learning for the problem mentioned above, your focus will shift to learning the embeddings for the images so that the embedding vectors are as close together as they can be for the images in each class and, for each class, as far as they can be from the embeddings for the other classes.

Success of metric-learning based frameworks when used with difficult datasets speaks to the fact that learning the mappings from the images to the embedding vectors for the purpose of clustering is a lot more forgiving than directly learning how to classify the images in one go.

After you have trained a network for clustering the embedding vectors, for the purpose of classifying a previously unseen images, you can use the computationally efficient nearest-neighbor classifier or even the nearest cluster-center classifier.

Preamble

My main goal in this lecture is to introduce you to some of the basic ideas in metric learning — especially those that revolve around the Pairwise Contrastive Loss and the Triplet Loss.

The Pairwise Contrastive Loss for metric learning was first promulgated in a 2006 publication "*Dimensionality Reduction by Learning an Invariant Mapping*" by Hadsell et al.:

https://www.academia.edu/download/56222713/cvpr06.pdf

For Pairwise Contrastive (PC) Loss, you first extract positive and negative pairs of training samples from a batch. Positive pairs are those for which both the items in a pair carry the same class label. For negative pairs, the two items in the pair carry different labels. Extracting the Positive and Negative pairs from a batch is referred to as *mining*.

The Pairwise Contrastive Loss is a sum of the values calculated separately from the positive pairs and the negative pairs. For the positive pairs, it is simply the average distance between the two items in a pair — since our goal is to make that distance as small as possible.

Preamble

The component of the Pairwise Contrastive Loss for the negative pairs is, however, a bit more complicated and it requires specifying a quantity known as the margin. To understand the notion of a margin, remember that our goal is to make as large as possible the distances between the two items in the negative pairs. However, the intuition would suggest that we should exclude the negative pairs for which the distance between the two items is already large. That's what's accomplished by using a margin. Should the distance between the two items of a negative pair exceed the margin, we want such a negative pair to contribute zero to the overall loss. For distances less than the margin, we want the loss to be $m - d$ where $m > 0$ is the margin and d the distance between the two items. A minimization of $m - d$ would obviously cause the distance d to at least approach the value of m.

That takes me to making some introductory remarks about the Triplet Loss for metric learning that was first formulated in a 2015 publication "*FaceNet: A Unified Embedding for Face Recognition and Clustering*" by Schroff et al.:

`https://arxiv.org/pdf/1503.03832.pdf`

Preamble

Formulation of the Triplet Loss starts with every pair of images in a batch that have the same class label; one of the two is considered to be the Anchor and the other as Positive. Every $(Anchor, Pos)$ pair constructed in this manner forms a Triplet with a Negative, that is, an image with a different class label, provided the Neg satisfies certain conditions on the $(Anchor, Pos)$ distance vis-a-vis the $(Anchor, Neg)$ distance.

Creating $(Anchor, Pos, Neg)$ triplets from a batch in the manner described above is also referred to as *mining*, but now we can talk about negative-hard mining, negative semi-hard mining, etc., with the different mining strategies possessing different computational properties related to convergence and the avoidance of getting trapped in local minima. These different approaches to triplet mining depend on the distance between the anchor and the negative vis-a-vis the distance between the anchor and the positive.

In the material that follows, I'll start with an explanation of the fundamental notions mentioned so far in this preamble. Subsequently, I'll go over some interesting coding issues related to computationally efficient approaches to the mining of pairs and triplets from a batch.

Preamble

Even after you have trained a network to map the input data to the embedding vectors that minimize the distances between the vectors that are supposed to be similar and maximize the distances between those that are meant to be dissimilar, you are still faced with the challenge of how to search in a given set of such embeddings.

To elaborate, let's say you have an image — I'll refer to it as the query image — and you want the network mentioned above to tell you what its class label should be. You feed the image into the network and it outputs an embedding vector for the image. Now you want to search through a database of embedding vectors (with known associated class labels) that were previously produced by the network. By searching through this database, you want to return for the query image the same class label that belongs to its closest neighbor in the database.

When the sort of databases mentioned above are large, searching for the nearest neighbor for a query embedding poses its own challenges — on account of the rather high-dimensionality of the embeddings. I will address these challenges in this lecture and talk about the modern search methods that are based on the notion of product quantization.

Preamble

Finally, I'll describe DLStudio's `MetricLearning` class that has my implementations for both pairwise contrastive learning and triplet learning.

Your best entry points for becoming familiar with the code in the MetricLearning class are the following two scripts in the top-level directory `ExamplesMetricLearning` in the DLStudio platform:

1. example_for_pairwise_contrastive_loss.py

2. example_for_triplet_loss.py

As the names imply, the first is for learning about the Pairwise Contrastive Loss for metric learning and the second for the Triplet Loss for doing the same.

Both these scripts use (through the invocations of the functions programmed into DLStudio's MetricLearning co-class) the FAISS library for the nearest-neighbor search for evaluating the performance of the embeddings learned.

Both these scripts also produce visualizations of the clusters by using the tSNE algorithm.

Preamble

This Preamble would not be complete if I did not share with you the news that the CVPR 2024 conference has just accepted a Metric Learning publication from us:

"Dual Pose-invariant Embeddings: Learning Category and Object-specific Discriminative Representations for Recognition and Retrieval"

This work is based on Rohan Sarkar's Ph.D dissertation work in RVL.

In case you did not know, CVPR is one of a small number of conferences where the latest developments in Deep Learning are first presented. The other conferences of the same caliber are ICCV, NeurIPS, and ICLR. It is difficult to get a paper accepted at any of these conferences.

Outline

Outline

Representing Images with Embeddings

- The basic issue is representing each input image in a vector space of a user-specified dimensions. It is not that different from the more traditional idea of extracting a certain number of features from an image (which could, for example, be color, texture, and other features), and then representing an input image by a single point in the vector space spanned by all the features.

- In the jargon of deep learning, we refer to such a vector representation as an *embedding* since we are embedding the image in the underlying vector space.

- The big difference between the old-style vector representations and the modern embeddings is that one had to manually specify the features for the former, whereas it is done automatically under the hood for the latter.

Representing Images with Embeddings (contd.)

- With regard the automatic specification of the features to be used for the calculation of the embeddings, as you know, each convolutional layer outputs feature maps for a particular sized convolutional operator whose parameters are all learnable weights. Depending on the number of output channels, you end up with that many feature maps at the output. If the final layer of the network is a Linear layer, the number of nodes in the output of that layer will be the embedding produced for the input image. This embedding would obviously be the end-product of the feature maps discovered along the way.

- As I stated earlier in the Preamble, the goal of Metric Learning is to learn the embeddings so that the vectors are pulled together for similar images (meaning images of the same class) and pushed further apart for dissimilar images (meaning images of the different classes).

- The notions of pulling-together or pushing-apart of the vectors requires that we first agree upon how to measure the distance between two vectors in the underlying space.

Representing Images with Embeddings (contd.)

- I'll represent the embedding for an image $\mathbf{x}$ by $f(\mathbf{x})$. You can think of the neural network that generates the embeddings as creating a mapping function $f()$ from the set of all images to their embeddings.

- Given two images $\mathbf{x}$ and $\mathbf{y}$, the most commonly used distance metric for measuring how close or distant their embeddings are is through the Euclidean metric (also known as the L_2-norm):

$$d(f(\mathbf{x}), f(\mathbf{y})) \quad = \quad ||f(\mathbf{x}) - f(\mathbf{y})||_2$$

  where the subscript 2 on the vector norm means that we are talking about the L_2-norm (which is the same thing as the Euclidean norm).

- And when the distance between the two vectors is used as a loss for training a neural network, we are likely to use the square of the distance because it is differentiable everywhere:

$$\mathcal{L} \quad = \quad d^2(f(\mathbf{x}), f(\mathbf{y})) \quad = \quad ||f(\mathbf{x}) - f(\mathbf{y})||_2^2$$

Representing Images with Embeddings (contd.)

- This sets us up to talk about the Pairwise Contrastive Loss and the Triplet Loss that can be used to learn the embeddings for the input images such that the embeddings for the images that are meant to be similar are pulled together and those for the images that belong to different classes are pushed apart.

Outline

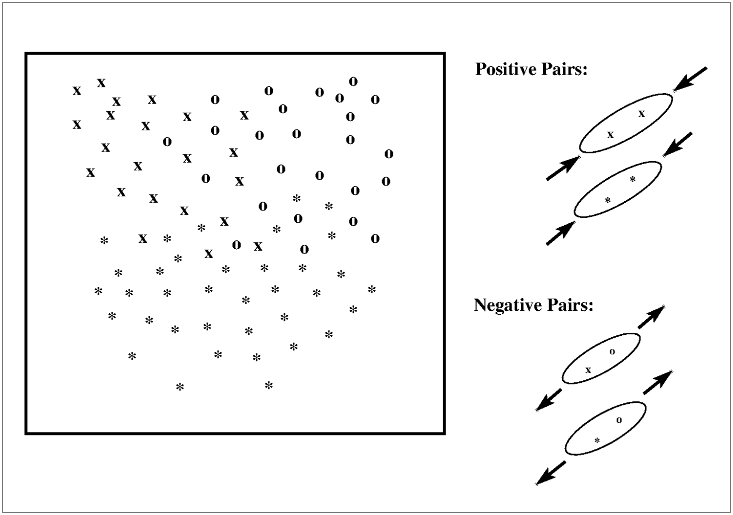
Pairwise Contrastive Loss

- As its name implies, Pairwise Contrastive Loss requires that a batch be reorganized in the form of pairs of training samples. There are two types of pairs to be constructed: Positive Pairs and Negative Pairs. Both items in a Positive Pair carry the same class label and the two items in a Negative Pair must carry different class labels. The piece of code that converts a batch into such pairs is called a *miner*.

- Shown on the next slide is a depiction in which the training data belongs to three classes. The depiction is in some 2D feature space for the purpose of this explanation. We want to map these training samples to embeddings so that the embedding vectors for any two samples that belong to the same class are pulled together and the embeddings for any two samples that belong to two different classes are pulled apart.

- Let $f(\mathbf{x})$ represent the mapping of a training sample $\mathbf{x}$ to an embedding vector of user-specified dimensionality. And let $d(\mathbf{a}, \mathbf{b})$ be the L_2 distance between the embeddings vectors $\mathbf{a}$ and $\mathbf{b}$.

Mining Positive and Negative Pairs



Formulation of Pairwise Contrastive Loss

- Given a positive pair of images, $(\mathbf{x}_1, \mathbf{x}_2)$, we would want its contribution to the loss to be directly proportional to the L_2 distance between the embedding vectors $f(\mathbf{x}_1)$ and $f(\mathbf{x}_2)$. We could, for example, set a positive pair's contribution to the loss to $d^2(f(\mathbf{x}_1^i), f(\mathbf{x}_2^i))$ where i indexes all the positive pairs extracted from the batch:

$$L_p \quad = \quad \sum_i \left[d\Big(f(\mathbf{x}_1^i), f(\mathbf{x}_2^i) \Big) \right]^2$$

  where the summation is over all positive pairs pulled from the batch. This contribution to loss makes eminent sense since the whole point of metric learning is to make such pairwise distances as small possible.

- That brings us to the more difficult case: The contribution to the loss made by a negative pair. The point here is that we want this distance to be as large as possible. Note that a specification that says "*to be as large as possible*" is NOT as precise as the specification "*to be as close to zero as possible*" for the case of positive pairs.

Formulation of Pairwise Contrastive Loss (contd.)

- Making the distance between the dissimilar items in a negative pair beyond a certain point would amount to wasting the learning effort. One can even go as far as saying that if the two samples in a negative pair are already well separated, they should not even participate in the learning process.

- These aspects of learning from negative pairs can be taken into account through the concept of a *margin* that is commonly represented by the symbol m.

- We can say that that if the two samples in a negative pair are separated by a distance larger than m, then the loss that such a negative pair contributes to the overall loss for a batch should be zero.

- In general, we can use the following expression to measure how much the j^{th}-negative pair $(\mathbf{x}_1^j, \mathbf{x}_2^j)$ should contribute to the overall loss:

$$\max\left\{0,\ m - d\Big(f(\mathbf{x}_1^j), f(\mathbf{x}_2^j)\Big)\right\}$$

Formulation of Pairwise Contrastive Loss (contd.)

- The expression shown at the bottom of the previous slide would yield a zero when the distance between the two embeddings in the pair exceeds the margin m. And, at the same time, by making the loss proportional to the value returned by the expression and realizing that the goal of the learning process is to alter the embedding vectors so that the loss is as small as possible, that would cause the distance between the embeddings to become larger — in the direction of approaching the value of m.

- If we use L_n to denote the partial loss that should be attributed to the negative pairs in a batch, we can set it to:

$$L_n \quad = \quad \sum_j \left[\max \left\{ 0,\ m - d\Big(f(\mathbf{x}_1^j), f(\mathbf{x}_2^j) \Big) \right\} \right]^2$$

  where j is the index over all the negative pairs pulled from the batch.

- The next slide shows how we can write a single composite expression for the overall loss for all the pairs in a batch by combining L_p and L_n.

Formulation of Pairwise Contrastive Loss (contd.)

- In general, from the standpoint of how the pairwise contrastive loss is coded up, it is more efficient to introduce a binary variable $y \in \{0, 1\}$ whose value is 0 for a positive pair and 1 for a negative pair and to then define the following composite expression for the overall loss $\mathcal{L}$ associated with a batch:

$$\mathcal{L} \;=\; \sum_i \;\; (1 - y_i) \cdot \left[d\Big(f(\mathbf{x}_1^i), f(\mathbf{x}_2^i) \Big) \right]^2 \;+\; y_i \cdot \left[\max\Big\{ 0,\; m - d\Big(f(\mathbf{x}_1^i), f(\mathbf{x}_2^i) \Big) \Big\} \right]^2$$

  where the index i goes over all the pairs mined from a batch.

Outline

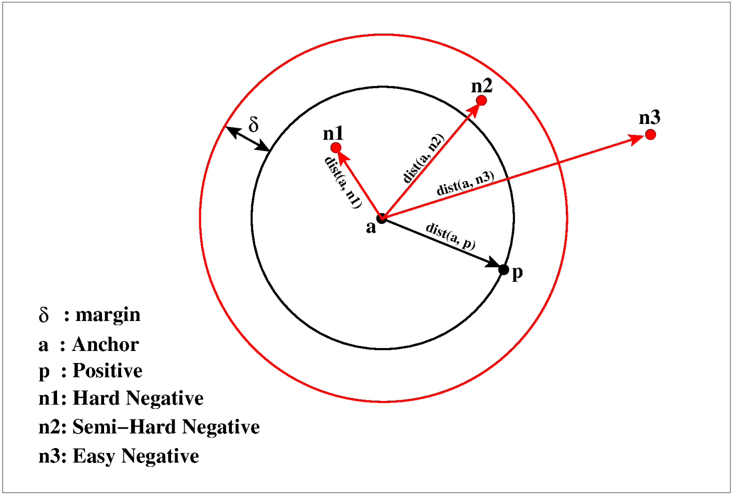
Triplet Loss

- For calculating the Triplet Loss, a batch is mined for the triplets that are formed using the rule described below for every pair of training samples that have the same class label.
- Given a pair of training samples with the same class label, one of the two is chosen as the Anchor and the other as the Positive, thus forming an $(Anchor, Positive)$ pair.
- For each $(Anchor, Positive)$ pair thus formed, the miner collects all the samples in the batch whose class labels are different from those for the $(Anchor, Positive)$ pair. These would constitute the set of negative samples for a given $(Anchor, Positive)$ pair. Let's denote this set by $\mathcal{N}$.
- For a given $(Anchor, Positive)$ pair, I'll denote the distance between the Anchor and the Positive by $d(a, p)$ and, for any choice of the negative sample $n \in \mathcal{N}$, the distance between the Anchor and the Negative by $d(a, n)$.

Triplet Loss (contd.)

- What's interesting is that not all the negative samples $n \in \mathcal{N}$ carry the same level of significance with regard to how informative they are for the learning that is required.
- Consider the three negative samples, n_1, n_2 and n_3, for the $(Anchor, Pos)$ pair shown in the figure on the next slide.
- Remember, our goal is for two embeddings in the pair $(Anchor, Pos)$ to come as close together as possible and for the embeddings in the pair $(Anchor, Neg)$ to be as far apart as possible. In relation to the distance between the Anchor and the Pos, the negative embedding n_3 is already too far out. Our formulation of the loss has to be such that a negative embedding such as n_3 plays an insignificant role, if any at all, in the learning process.
- Now consider the opposite case that is represented by the negative embedding n_1 — it is closer to the Anchor than the Positive in the Triplet. Any learning must push the embedding vectors for such negatives further out.

Triplet Loss (contd.)



Triplet Loss (contd.)

- Experience has shown if only the negatives that constitute Hard Negative Mining are selected for the loss function, that causes the network to converge to a local minimum at best, or to collapse to a state in which all the embedding vectors are zero.

- So here we are: On the one hand, the negatives that are too far out from the Anchor are ineffective for the learning that is required and the negatives are too close can cause the learning to become unstable.

- The dilemma described above is resolved by formulating the notion of a user defined *margin* that is illustrated in the figure on the previous slide. The margin is supposed to capture our bet that the negatives in that band are sufficiently close to the Anchor for some effective learning, but without the values for the learnable parameters getting stuck in a local minimum.

- In this lecture I denote the width of the margin by δ. A typical value for δ is 0.2. The next slide explains why this number makes sense.

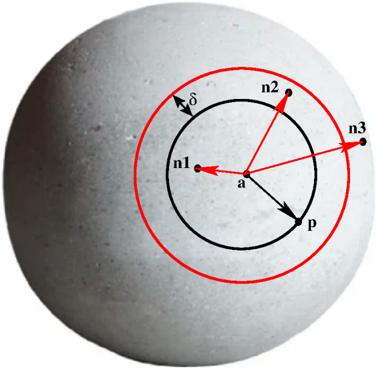
Triplet Loss (contd.)

- To make sense of any size specified for the margin, note that, during training, all embeddings are normalized to be of size unity. So what's shown in the figure on Slide 28 is best visualized on the surface of a unit sphere. That is, all of points labeled 'a' for Anchor, 'p' for Positive, 'n1' for one of the three Negatives, etc., reside on the surface of a unit sphere. And you can think of the margin band is an annulus on the surface of the sphere as shown on the next slide.

- δ is obviously a hyperparameter of Metric Learning with Triplet loss. For a given value of δ, a miner can divide the set $\mathcal{N}$ of negatives into hard-negatives, semi-hard negatives, and easy-negatives on the basis of their distance from the anchor as shown in the figure on Slide 28.

- Slide 32 presents a summary of the definitions of what's meant by the different negative mining strategies.

Triplet Loss (contd.)



All five points shown, including the anchor point 'a', are on the surface of the sphere.

Triplet Loss (contd.)

- For a given $(Anchor, Positive)$ pair in a batch, shown below are the criteria for dividing the set of negatives $\mathcal{N}$ in the batch into three separate groupings:

Hard-Negative Mining: This strategy involves the negatives that are closer to the Anchor than the Positive in the $(Anchor, Positive)$ pair.

$$dist(a, n) \quad < \quad dist(a, p)$$

With hard-negative mining, a given pair $(Anchor, Positive)$ is likely to be extended to a single triplet involving the negative that is closest to the Anchor.

Semi-Hard Negative Mining: These negatives fall in the margin δ shown in the figure on Slide 28:

$$dist(a, p) \quad < \quad dist(a, n) \quad < \quad dist(a, p) + \delta$$

With semi-hard-mining, a given pair $(Anchor, Positive)$ is likely to be extended to as many triplets as the number of negatives within the margin band.

Easy Negatives: Finally, these are the negatives that lie beyond the margin:

$$dist(a, p) + \delta \quad < \quad dist(a, n)$$

As you will see later, these are typically discarded in the calculation of the loss.

Triplet Loss (contd.)

- Let's use $\mathcal{T}$ to denote the set of triplets constructed from a batch and let N be its cardinality.
- Let's denote the three images in the i^{th} triplet by $(\mathbf{x}_i^a, \mathbf{x}_i^p, \mathbf{x}_i^n)$. Based on the concepts in the previous section, we can write the following expression for Triplet Loss:

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

- Note that, for a given pair $(Anchor, Pos)$, the loss defined above will not accept any contributions from those negatives that are outside the margin shown in the figure on Slide 28. For any negatives outside the margin, the value of $||f(\mathbf{x}_i^a) - f(\mathbf{x}_i^n)||_2^2 \;-\; \delta$ will exceed that of $||f(\mathbf{x}_i^a) - f(\mathbf{x}_i^p)||_2^2$. And, when that happens, the max will return 0 for those triplets.

Outline

You Need Pairwise Distances Between Embedding Vectors

- Based on the discussion so far in this lecture, fundamental to the estimation of the Pairwise Contrastive Loss or the Triplet Loss is the calculation of the distances *between* the embedding vectors in a batch. If you are not careful with how this step is programmed, you are highly unlikely to end up with a practically useful framework for similarity learning.

- Let's say the batch tensor produced by a network is of shape (B, M) where B is the batch size and M the size of the embeddings learned so far for the input images.

- When you input a batch of images into the embedding generator, the dataloader would also have made available the classification labels associated with the images. Therefore, at the output of the embedding generator, you will have two things: A tensor of the embeddings, the shape of the tensor being (B, M), and another $(B, 1)$ shaped tensor of the integer labels that go with each embedding vector in the batch.

Forming the Pairs and the Triplets

- Next, in order to estimate either of the metric learning losses, you must form positive and negative pairs from the embedding vectors in the batch, or form the triplets. Subsequently, you must calculate the distances between the vectors.

- Creating positive and negative pairs, or the triplets, is *seemingly trivial* since it only requires the labels associated with the batch instances. I'll illustrate that with a simple example on Slide 39 that is based on the batch size $B = 6$ and the embedding size $M = 3$.

- As to my choice of the phrase *seemingly trivial* in the previous bullet, what that usually means is that you can solve a problem easily by iterating through the data using, say, a "`for`"-loop. **But, unfortunately, "`for`"-loops and GPU based processing were never meant to be natural friends.** GPU based processing is fundamentally about parallel execution of matrix-vector multiplications — especially when the same matrix needs to multiply different vectors.

Forming the Pairs and the Triplets (contd.)

- To continue with the thought in the last bullet on the previous slide, in addition to the fact that iterative processing with loops is an anathema to what GPU processing is all about, **what exacerbates the incompatibility between the two is how loops are handled in Python.**

  [**Python being a high-level language, its loops are costly and generate a lot of not-really-needed book-keeping code when its loops are translated into C. What I mean is that the same loop implemented directly in C would need far fewer memory allocations and need much shorter execution time.**]

- In the context of similarity learning with neural networks, the cost of iterative processing is simply much too great. [**As I will argue in this section, depending on the batch size, you may be able to achieve a thousand-fold speedup when you eliminate the loops that you would otherwise need for estimating the Pairwise Contrastive or Triplet loss functions.**]

- In the remainder of this section, I'll start with the simple iterative implementations — just to get the idea across that fundamentally what you need to achieve is trivial. **And then I will switch over to the purely tensorial implementations.** The discussion in the examples I will use for the latter are based on the article "Triplet loss and quadruplet loss via tensor masking" by Tomek Korbak:

`https://tomekkorbak.com/2020/11/15/triplet-loss-quadruplet-loss/`

Forming the Pairs and the Triplets (contd.)

- But first, as promised, we start with a trivial implementation as shown on the next slide.
- The integer labels associated with the 6 embedding vectors in Line (A) are shown in Line (B). Lines (C), (E), and (G) show the straightforward syntax using list comprehensions for forming the positive and negative pairs and the triplets.
- The integer labels that you see in the results produced by the print statements in Lines (D), (F), and (H) are NOT to be confused with the integer labels defined in Line (B).
- The integers in the pair and triplet results are the index values associated with the embeddings in Line (A). The index associated with the first embedding vector 0, with the second 1, with the third 2, and so on.

Forming Pairs and Triplets (contd.)

- Shown below is the code example mentioned on the previous slide for illustrating how trivial it is to form positive and negative pairs and the triplets using just the labels associated with the embedding vectors.

```
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)
##          [((0, 2), 1), ((0, 2), 3), ((0, 2), 4), ((0, 2), 5),
##                              ((3, 5), 0), ((3, 5), 1), ((3, 5), 2), ((3, 5), 4)]
```

Forming Positive and Negative Pairs Without the `for` Loops

- As it turns out, given a vector of labels for the embedding vectors in a batch, it is not too difficult to figure out the pairs and triplets without the "`for`"-loops shown on the previous slide. [**All you need do is to carry out a test for boolean equality between the $B \times B$ array constructed from the column-vector representation of the label vector with the same array constructed from the row-vector representation as shown below.**]

```
>>> 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]])
```

[**This works because, in general, when a binary operator in PyTorch (and also in numpy) is supplied with two argument arrays whose shapes do not agree with respect to one of the axes, the argument arrays "sweep" over the conforming axis to bring the two shapes into correspondence.**]

- Another way to accomplish the same thing as shown above would be:

```
>>> labels = torch.tensor([0, 1, 0, 3, 4, 3])
>>> labels_equal = labels[None,:] == labels[:,None]          # See the next slide for the use of "None" here
>>> 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]])
```

Forming Positive and Negative Pairs Without the `for` Loops (contd.)

- To elaborate on the use of `None` at the different index positions in the array access syntax on the previous slide, this ploy endows the tensor with a new axis of dimensionality 1. [**With `None` placed at the leftmost index position, you are basically embedding what was previously a 1-D array into a $1 \times B$ array. And with `None` placed at the rightmost index position, the array data in `labels` gets mapped into Axis 0 of dimensionality 6, whereas the new Axis 1 has dimensionality 1.**]

- Think of `None` in this context as "no data" along the new axis being created. This use of `None` is intimately tied with the use of the slice operator ":" for mapping the available data to the other axes of the target array.

- In the result shown on the previous slide, the tautologically true annoying entries on the diagonal can be gotten rid by taking a logical "&" of the result shown above with the negation of a boolean identity array as shown below:

```
>>> labels_equal  &  ~  torch.eye(B, dtype=bool)
tensor([[False, False,  True, False, False, False],
        [False, False, False, False, False, False],
        [ True, False, False, False, False, False],
        [False, False, False, False, False,  True],
        [False, False, False, False, False, False],
        [False, False, False,  True, False, False]])
```

Probabilities of Positive and Negative Pairs in a Batch

- The example on the last couple of slides shows a lot more negative pairs than the number of positive pairs.

  [**As to why, let n_c be the number of classes in dataset. The probability that any one batch image chosen randomly would belong any one given class is $1/n_c$. Therefore, the probability that a randomly chosen image from a batch will not correspond to a given class is $1 - \frac{1}{n_c}$. And the probability that none of the images in the batch under consideration would carry a specific label is $(1 - \frac{1}{n_c})^B$ where B is the batch size. Therefore, the probability that at least one image in the batch will carry that label is $1 - (1 - \frac{1}{n_c})^B$. Note that the commonly used approximation for such expressions, which would be $\frac{B}{n_c}$ in our case, that is based on $(1 + x)^n \approx 1 + nx$ as $x \to 0$, cannot be used in our case because the ratio $\frac{B}{n_c}$ is likely to be far from satisfying the $x \to 0$ condition.**]

- Extending the above reasoning to the probabilities of Positive and Negative Pairs in a batch, the probability of Positive Pair would be $\frac{1}{n_c} \cdot (1 - (1 - \frac{1}{n_c})^B)$ and the probability of a Negative Pair would be given by $\frac{1}{n_c} \cdot \frac{n_c - 1}{n_c}$.

  [**The expression for the Positive Pair is the product of $1/n_c$ for the probability finding the first image of the pair corresponding to one of the n_c classes and the probability derived above for the class label for image to be a specific label — the label for the first image in the pair. For the Negative Pair, the first component of the product remains the same as before, but the second component becomes $1 - \frac{1}{n_c}$ for the $n_c - 1$ classes allowed for the second image in the pair. Very approximately, these probabilities boil down to $\frac{1}{{n_c}^2}$ for the Positive Pair verses $\frac{n_c - 1}{{n_c}^2}$ for the Negative Pair. That should give you a sense of why you are likely to have many more Negative Pairs than Positive Pairs.**]

Calculating the Distance Matrix

- The Distance Matrix is the Euclidean distance between every pair of embedding vectors in the batch. After you have calculated such a Distance Matrix, it can be masked in various ways to yield the losses.

- For my explanation of how best to compute the Distance Matrix, I'll use the same example as before for the embeddings:

```
import torch
device = torch.device('cuda')
X = torch.FloatTensor(                                     ## Each row is an embedding vector
                       [ [0.0, 0.1, 0.0],
                         [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] ],
                     ).to(device=device)
```

- A most naive way to compute the Distance Matrix for these 6 embedding vectors would be to perform the calculations in a double "`for`"-loop, but, as explained on the next slide, that's guaranteed to be much too inefficient in a practical scenario.

Calculating the Distance Matrix (contd.)

- To continue with the last bullet on the previous slide, as the outer loop variable indexes over each of the embedding vectors, the inner loop variable will also index over all of the embedding vectors while, at the same time, calculating the Euclidean Distance with respect to the vector pointed to by the outer loop variable.
- The naive approach outlined above will always be much too slow for the batch sizes likely to be used for metric learning. With a batch size of 128, the system would need to make 8,192 fetches from the GPU memory — as opposed to a single fetch with the procedure outlined below. In that sense, the inherently tensor based solution described in what follows will give you a speedup of over 8,000 over the naive approach.
- The alternative is a purely tensor based approach with **no** "`for`"-loops as outlined on the next few slides.

Calculating the Distance Matrix (contd.)

- The without "`for`"-loops and purely tensor-based method for calculating the Distance Matrix starts with the realization that the following *matrix multiplication* yields a dot product of every pair of embedding vectors in X

$$dot\_products \quad = \quad X \; @ \; X.T \tag{1}$$

[**If B is the batch size, the shape of X will be (B, M) where M is the size of the embedding vectors. For our example, the shape of X is $(6, 3)$. A matrix multiplication of $(6, 3)$ tensor with a $(3, 6)$ tensor will yield a tensor of shape $(6, 6)$. The $(i, j)^{th}$ element the resulting tensor will be the dot product of the i^{th} embedding vector with the j^{th} embedding vector. The 6 elements on the diagonal of dot_products tensor will be the squared norms of each of the six embedding vectors. The computational "trick" of packing all the vectors into an array and calculating all the pairwise dot products between the vectors simultaneously through matrix multiplication owes its origins to early work in creating efficient implementations for the PCA algorithm. For more info on that work, just google my "Optimal Subspaces" tutorial.**]

- Let's now see how the information in the `dot_products` tensor shown above can be marshaled for calculating the Distance Matrix. But first let's stare at the following formula for the Euclidean distance between two embedding vectors, x and y, in the tensor X:

$$||\vec{\mathbf{x}} - \vec{\mathbf{y}}||^2 \quad = \quad (\vec{\mathbf{x}} - \vec{\mathbf{y}})(\vec{\mathbf{x}} - \vec{\mathbf{y}})^T \quad = \quad ||\vec{\mathbf{x}}||^2 - 2\vec{\mathbf{x}} \cdot \vec{\mathbf{y}}^T + ||\vec{\mathbf{y}}||^2 \tag{2}$$

Calculating the Distance Matrix (contd.)

- What the RHS of the formula shown at the bottom of the previous slide says is that to calculate the Euclidean distance between any two embedding vectors, we need the square of the norm of each of the vectors. And, we need the value of the dot product between the two vectors. What's most interesting about the tensor dot product shown in Eq. (1) on the previous slide is that it contains both of these quantities for *every* pair of embedding vectors in X.

- For example, the individual vector norms for the embedding vectors shown in Eq. (2) are on the diagonal of the tensor *dot_product* in Eq. (1). And the dot products between pairs of embedding vectors needed in Eq. (2) are in the off-diagonal elements of *dot_product* tensor. Recall again, when B is the batch-size, the shape of the *dot_product* tensor is (B, B).

- Our *distance_matrix* is also of shape (B, B) and its $(i, j)^{th}$-element is the distance between the i^{th} and the j^{th} embedding vectors.

Calculating the Distance Matrix (contd.)

- Therefore, at every $(i,j)^{th}$-element of *distance_matrix*, we must make available the squared norm of the i^{th} embedding vector, and also the squared norm of the j^{th} embedding vector. The question then becomes as to the best way to "pull" them out the diagonal of the *dot_product* tensor and present them where they are needed.

- Let's first extract the squared norms off the diagonal of the *dot_product* tensor. This we can do by:

$$squared\_norms\_embedding\_vecs \quad = \quad torch.diagonal(dot\_products) \tag{3}$$

  For the 6-embedding vectors example on Slide 43, the answer returned by the right-hand side will be a one-axis tensor of 6 values, one for each of the 6 embedding vectors. If we printed out this result for that example, we would get

```
print(squared\_norms\_embedding\_vecs)
##           tensor([0.0100, 0.0600, 0.2600, 0.1600, 0.0900, 0.5000], device='cuda:0')
```

Calculating the Distance Matrix (contd.)

- The first of the six squared norms shown at the bottom of the previous slide needs to be made available at all the elements of the first column *distance_matrix* on account of the $j = 0$ index values and also at all the elements of the first row of the same matrix on account the $i = 0$ index values. Recall, we use the (i, j) index for the elements of *distance_matrix*.

- Along the same lines, the second of the six values shown at the bottom of the previous slide needs to be made available at both the second column and the second row of *distance_matrix*. And so on.

- In addition to the embedding-vector squared norms, we must also make available at each (i, j) element of *distance_matrix* the dot product of the i^{th} and the j^{th} embedding vectors.

$$\begin{aligned} distance\_matrix \quad = \quad & squared\_norms\_embedding\_vecs.view(1,6) \\ & - \; 2.0 * dot\_products \\ & + \; squared\_norms\_embedding\_vecs.view(6,1) \end{aligned} \tag{4}$$

Calculating the Distance Matrix (contd.)

- In case you are wondering how it is possible to add (or subtract) together three tensors of very different shapes — $(1,6),(6,6),(6,1)$ — it is because several of the operators are overloaded for the tensors in a manner inherited from numpy.

- Shown below is a short interactive Python script that adds to a 2×2 array a 1×2 array in Line (A) and a 2×1 array in Line (B). [**For the examples shown in this lecture, as long as one of the two argument arrays to an operator agree with respect to all but one of the axes of the arrays involved, the smaller array can be thought as "sweeping" through the non-conforming axis of the larger array as the operation in question is being carried out.**]
[**This is referred to as the smaller array (or tensor) "broadcasting" across the larger array (or tensor). The goal of broadcasting is to vectorize array operations so that looping occurs much more efficiently in the underlying C code as opposed to in Python itself. Google "numpy broadcasting" and "pytorch tensor broadcasting" for more info.**]

```
>>> X = torch.tensor( [[3,4], [5,6]] )
>>> X
tensor([[3, 4],
        [5, 6]])
>>> Y = torch.tensor( [ [100, 200] ] )
>>> Y
tensor([[100, 200]])
>>> X + Y                                                    ## (A)
tensor([[103, 204],
        [105, 206]])
>>> X + Y.T                                                  ## (B)
tensor([[103, 104],
        [205, 206]])
```

Calculating the Distance Matrix (contd.)

- Slides 44 through 48 presented a tensor-based solution for calculating the pairwise distances between all the embedding vectors in a batch. [**The final calculation in Eq. (4) on Slide 48 is the answer for the Distance Matrix. That solution required that you first calculate the dot-product of the batch tensor with its transpose, pull out the values that are on the diagonal of the dot-product tensor, and then use Eq. (4) to merge the vectors norms with the vector dot products for the answer.**]

- What's interesting is that if you wanted to use `None` in the manner explained previously on Slide 41, you could carry out all of the calculations mentioned above in a single statement as shown below:

```
>>> X
tensor([[0.0000, 0.1000, 0.0000],
        [0.1000, 0.1000, 0.2000],
        [0.4000, 0.3000, 0.1000],
        [0.0000, 0.0000, 0.4000],
        [0.3000, 0.0000, 0.0000],
        [0.1000, 0.0000, 0.7000]])
>>>
>>> distance_matrix =  torch.sum( ( X[None,:]  -   X[:,None] )**2, 2 )  ##  This distance_matrix is the same as yielded by
>>>                                                                     ##    Eq. (4) on Slide 48.  The last arg of value 2
>>>                                                                     ##    means that sum is to carried out along Axis 2.
>>>                                                                     ##  See Slide 41 for what is accomplihsed by None
>>>                                                                     ##    in the array element access syntax.
>>>
>>> distance_matrix
tensor([[0.0000, 0.0500, 0.2100, 0.1700, 0.1000, 0.5100],
        [0.0500, 0.0000, 0.1400, 0.0600, 0.0900, 0.2600],
        [0.2100, 0.1400, 0.0000, 0.3400, 0.1100, 0.5400],
        [0.1700, 0.0600, 0.3400, 0.0000, 0.2500, 0.1000],
        [0.1000, 0.0900, 0.1100, 0.2500, 0.0000, 0.5300],
        [0.5100, 0.2600, 0.5400, 0.1000, 0.5300, 0.0000]])
```

Constructing the Triplet Mask

- After you have calculated the $B \times B$ Distance Matrix whose $(i, j)^{th}$-element is the Euclidean distance between the i^{th} and j^{th} embedding vectors in a batch of size B, you must add the distances for some of those elements for calculating the loss.

- The elements of the Distance Matrix that needed to be added together are identified with a $B \times B$ boolean mask.

- For Pairwise Contrastive Loss, this mask is directly given by the logic explained on Slides 39 and 40. However, for the Triplet Loss, the mask is a little bit more complicated on account of the fact that each element of the summation on Slide 33 involves three embedding vectors: anchor, positive, and negative. On Slide 33, the embedding vectors in the i^{th} triplet were identified as (x_i^a, x_i^p, x_i^n).

- In order to simplify the discussion to follow, I'll assume that each triplet es denoted by the index values (i, j, k) where i is the index in the batch for the anchor embedding vector, j for the positive, and k for the negative.

Constructing the Triplet Mask (contd.)

- Using the triples (i, j, k) for indexing the triplets pulled from a batch, we can use the logic outlined in this slide to construct a Triplet Mask. First we define three boolean arrays as follows:
  - Define a boolean array of shape $(B, B, 1)$ that has True at all elements for which i is **not** equal to j.
  - Define a boolean array of shape $(B, 1, B)$ that has True at all elements where i is **not** equal to k.
  - Define a boolean array of shape $(1, B, B)$ that has True at all elements for which j is **not** equal to k.
- What's interesting is that we can take advantage of the broadcast property of the numpy arrays and PyTorch tensors (see Slide 49) so that we can essentially ignore the axis whose dimensionality is 1 for each of the three boolean arrays defined above.

  [**When we take a logical `and` of all three boolean arrays, the resulting boolean array will have values True only when all three indexes i, j, and k are unequal.**]

Constructing the Triplet Mask (contd.)

The code snippet shown below shows that the starting point for the defining the three boolean arrays mentioned in the previous slide is a $B \times B$ boolean array that stores True wherever the row index is *not* equal to the column index. Recall, B is the size of the batch of the embedding vectors. In the code shown below, the name of this array is *not_equal_ij*:

```
>>> labels = torch.tensor( [0, 1, 0, 3, 4, 3])          ## Means that the first embedding vector in the batch has label 0,
>>>                                                     ## the second the label 1, the third again the label 0, and so on.
>>>                                                     ## The first part of what's shown below only depends on B.
>>> B = labels.shape[0]
>>> B                                                   ## The batch size is 6
6
>>> not_equal_ij = ~ torch.eye(B,dtype=bool)
>>> not_equal_ij
tensor([[False,  True,  True,  True,  True,  True],
        [ True, False,  True,  True,  True,  True],
        [ True,  True, False,  True,  True,  True],
        [ True,  True,  True, False,  True,  True],
        [ True,  True,  True,  True, False,  True],
        [ True,  True,  True,  True,  True, False]])
>>>
>>> i_not_equal_j[:,:,0]
>>> i_not_equal_j = not_equal_ij.view( B, B, 1 )
>>> i_not_equal_j[:,:,0]
tensor([[False,  True,  True,  True,  True,  True],
        [ True, False,  True,  True,  True,  True],
        [ True,  True, False,  True,  True,  True],
        [ True,  True,  True, False,  True,  True],
        [ True,  True,  True,  True, False,  True],
        [ True,  True,  True,  True,  True, False]])
```

(Continued on the next slide .....)

Constructing the Triplet Mask (contd.)

(...... continued from the previous slide)

```
>>> i_not_equal_k = not_equal_ij.view(B,1,B)
>>> i_not_equal_k
tensor([[[False,  True,  True,  True,  True,  True]],

        [[ True, False,  True,  True,  True,  True]],

        [[ True,  True, False,  True,  True,  True]],

        [[ True,  True,  True, False,  True,  True]],

        [[ True,  True,  True,  True, False,  True]],

        [[ True,  True,  True,  True,  True, False]]])
>>>
>>>
>>>
>>> j_not_equal_k = not_equal_ij.view(1,B,B)
>>> j_not_equal_k
tensor([[[False,  True,  True,  True,  True,  True],
         [ True, False,  True,  True,  True,  True],
         [ True,  True, False,  True,  True,  True],
         [ True,  True,  True, False,  True,  True],
         [ True,  True,  True,  True, False,  True],
         [ True,  True,  True,  True,  True, False]]])
>>> distinct_indices = i_not_equal_j & i_not_equal_k &  j_not_equal_k
>>> distinct_indices
tensor([[[False, False, False, False, False, False],
         [False, False,  True,  True,  True,  True],
         [False,  True, False,  True,  True,  True],
         [False,  True,  True, False,  True,  True],
         [False,  True,  True,  True, False,  True],
         [False,  True,  True,  True,  True, False]],

        [[False, False,  True,  True,  True,  True],
         [False, False, False, False, False, False],
         [ True, False, False,  True,  True,  True],
         [ True, False,  True, False,  True,  True],
         [ True, False,  True,  True, False,  True],
         [ True, False,  True,  True,  True, False]],
```

(Continued on the next slide .....)

Constructing the Triplet Mask (contd.)

(...... continued from the previous slide)

```
[[False,  True, False,  True,  True,  True],
 [ True, False, False,  True,  True,  True],
 [False, False, False, False, False, False],
 [ True,  True, False, False,  True,  True],
 [ True,  True, False,  True, False,  True],
 [ True,  True, False,  True,  True, False]],

[[False,  True,  True, False,  True,  True],
 [ True, False,  True, False,  True,  True],
 [ True,  True, False, False,  True,  True],
 [False, False, False, False, False, False],
 [ True,  True,  True, False, False,  True],
 [ True,  True,  True, False,  True, False]],

[[False,  True,  True,  True, False,  True],
 [ True, False,  True,  True, False,  True],
 [ True,  True, False,  True, False,  True],
 [ True,  True,  True, False, False,  True],
 [False, False, False, False, False, False],
 [ True,  True,  True,  True, False, False]],

[[False,  True,  True,  True,  True, False],
 [ True, False,  True,  True,  True, False],
 [ True,  True, False,  True,  True, False],
 [ True,  True,  True, False,  True, False],
 [ True,  True,  True,  True, False, False],
 [False, False, False, False, False, False]]])
```

- So far all we have done is to create a $B \times B \times B$ boolean array whose (i, j, k) indexed elements are True only where all three index values are different. For creating this boolean array, all we needed to know was the batch size B.

- Next, of all the True (i, j, k) triples, for the Triplet Mask we must choose only those that agree with the Triplet "discipline". Those triplets will be referred to as the valid triplets. For (i, j, k) to be a valid triplet, the pair (i, j), interpreted as $(anchor, pos)$, must form a Positive Pair and the the pair (i, k), interpreted as $(anchor, neg)$, must form a Negative Pair.

Constructing the Triplet Mask (contd.)

- For identifying the Positive and Negative Pairs in a given (i, j, k) in accordance with what was said in the last bullet in the previous slide, we are going to need some of the same logic that you saw earlier in Slide 40.
- In what follows, I'll use the logic of Slide 40 to make sure that in a valid (i, j, k), we have $labels[i] == labels[j]$ and $labels[i] \neq labels[k]$.

```
>>> labels
tensor([0, 1, 0, 3, 4, 3])                                    ##  Means that the first embedding vector in the batch has label 0,
>>>                                                           ##  the second the label 1, the third again the label 0, and so on.
>>>                                                           ##  The first part of what's shown below only depends on B.
>>> B
6                                                             ##  The batch size is 6.
>>> labels_equal_ij = labels.view(1, B) == labels.view(B,1)          ##  The goal is to identify the batch items indexed i
>>>                                                                  ##    and j that carry the same label
>>> labels_equal_ij
tensor([[ True, False,  True, False, False, False],           ##  If the location (i,j) is True, then the batch items i and j
        [False,  True, False, False, False, False],           ##    carry the same label
        [ True, False,  True, False, False, False],
        [False, False, False,  True, False,  True],
        [False, False, False, False,  True, False],
        [False, False, False,  True, False,  True]])
>>>
>>> labels_i_equal_j = labels_equal_ij.view(B,B,1)            ##  The boolean array created by the previous step is reshaped
>>>                                                           ##    for the logic needed for Triplet Mask generation
>>> labels_i_equal_j[:, :, 0]
tensor([[ True, False,  True, False, False, False],           ##  This is the same as what you saw above.   Again, this boolean
        [False,  True, False, False, False, False],           ##    array has True for all pairs (i,j) that carry the same label
        [ True, False,  True, False, False, False],           ##    in the batch
        [False, False, False,  True, False,  True],
        [False, False, False, False,  True, False],
        [False, False, False,  True, False,  True]])
```

(Continued on the next slide .....)

Constructing the Triplet Mask (contd.)

(...... continued from the previous slide)

```
>>> labels_i_equal_k = labels_equal_ij.view(B,1,B)        ##  Here comes a similar boolean arrary whose elements (i,k) are
>>>                                                        ##    True if the labels for i and k are identical in the batch
>>>                                                        ##    Note that we obtain this boolean array by simply reshaping
>>> labels_i_equal_k
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]]])
>>>
>>>                                                       ##  For a triple (i,j,k) to be valid from the standpoint of the
>>>                                                       ##    labels involved, the labels at i and j must agree, but the
>>>                                                       ##    labels at i and k must NOT be the same.
>>> valid_labels = labels_i_equal_j  &  ~labels_i_equal_k
>>>
>>> valid_labels
tensor([[[False,  True, False,  True,  True,  True],
         [False, False, False, False, False, False],
         [False,  True, False,  True,  True,  True],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False]],

        [[False, False, False, False, False, False],
         [ True, False,  True,  True,  True,  True],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False]],

        [[False,  True, False,  True,  True,  True],
         [False, False, False, False, False, False],
         [False,  True, False,  True,  True,  True],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False]],
```

(Continued on the next slide .....)

Constructing the Triplet Mask (contd.)

(...... continued from the previous slide)

```
        [[False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [ True,  True,  True, False,  True, False],
         [False, False, False, False, False, False],
         [ True,  True,  True, False,  True, False]],

        [[False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [ True,  True,  True,  True, False,  True],
         [False, False, False, False, False, False]],

        [[False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [ True,  True,  True, False,  True, False],
         [False, False, False, False, False, False],
         [ True,  True,  True, False,  True, False]]])
>>>                                                     ##  In what follows, distinct_indices boolean array is from Slide 54
>>>
>>> valid_labels_at_valid_indices = distinct_indices & valid_labels
>>>
>>> valid_labels_at_valid_indices

tensor([[[False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [False,  True, False,  True,  True,  True],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False]],

        [[False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False]],

        [[False,  True, False,  True,  True,  True],
         [False, False, False, False, False, False],
         [False, False, False, False, False, False],
```

(Continued on the next slide .....)

Constructing the Triplet Mask (contd.)

(...... continued from the previous slide)

```
          [False, False, False, False, False, False],
          [False, False, False, False, False, False],
          [False, False, False, False, False, False]],

         [[False, False, False, False, False, False],
          [False, False, False, False, False, False],
          [False, False, False, False, False, False],
          [False, False, False, False, False, False],
          [False, False, False, False, False, False],
          [ True,  True,  True, False,  True, False]],

         [[False, False, False, False, False, False],
          [False, False, False, False, False, False],
          [False, False, False, False, False, False],
          [False, False, False, False, False, False],
          [False, False, False, False, False, False],
          [False, False, False, False, False, False]],

         [[False, False, False, False, False, False],
          [False, False, False, False, False, False],
          [False, False, False, False, False, False],
          [ True,  True,  True, False,  True, False],
          [False, False, False, False, False, False],
          [False, False, False, False, False, False]]])
>>>
>>> how_many_triplets = torch.count_nonzero(valid_labels_at_valid_indices)
>>> how_many_triplets
tensor(16)                                                      ## The labels shown earlier allow for 16 Triplets as given
>>> torch.nonzero( valid_labels_at_valid_indices )
tensor([[0, 2, 1],
        [0, 2, 3],
        [0, 2, 4],
        [0, 2, 5],
        [2, 0, 1],
        [2, 0, 3],
        [2, 0, 4],
        [2, 0, 5],
        [3, 5, 0],
        [3, 5, 1],
        [3, 5, 2],
        [3, 5, 4],
        [5, 3, 0],
        [5, 3, 1],
        [5, 3, 2],
        [5, 3, 4]])
```

Outline

Similarity Search

- Metric learning algorithms, in and of themselves, do not constitute a complete solution to a practical problem.

- Consider, for example, the problem of retrieving from a database of images a photo that is most similar to the one you are looking at. Let's say you have trained a metric-learning network on the database using either contrastive loss or triplet loss and that it does a decent job of mapping the images to their corresponding embeddings. [As for the application scenario here, imagine that you have trained a metric-learning network on a large dataset of the face images of a large majority of the hard-core criminals in the United States. The police are tracking a particular individual and they want to compare a photo captured by a traffic camera with the images in the database.]

- I'll refer to the photo you are looking at as the *query image*. You can obviously feed the query image also into your trained metric learning network and obtain its embedding. We can refer to that as the *query embedding* or, more succinctly as just the *query*.

Similarity Search (contd.)

- Subsequently, you are faced with the question of how to compare the query embedding with the embeddings for the images in the dataset. What you want to do is to solve the "Nearest Neighbor" (NN) problem. That is, for the query embedding vector Q, you want to find the embedding vector in the dataset that is closest to Q.

- Conceptually speaking, the easiest solution to the problem is to carry out a brute-force calculation of the distance between Q and the embedding vectors for all the images in the dataset. The time complexity of the brute-force solution is $O(N)$ where N is the size of the dataset. The brute-force search is not scalable. It essentially amounts to a telephone operator having to scan serially through all the names (in the worst case) in a telephone directory when asked for the telephone number of a specific individual.

- If the dimensionality D of the embedding vectors were taken into account, the time-complexity formula show above would become $O(ND)$.

Similarity Search (contd.)

- For more efficient search for the nearest neighbor, we must resort to either using a popular data structure like the KD-Tree or using one of the ANN (Approximate Nearest Neighbor) algorithms.
- Since KD-Tree is a popular data structure for multi-dimensional data, in the next slide I argue that it's not going to work for what we have in mind — comparing the embeddings vectors for the images.
- Fortunately, we can resort to one of the ANN algorithms. Unlike a deterministic algorithm based on KD-tree, the nearest neighbor returned by an ANN algorithm is not guaranteed to be the true NN, but, with a probability approaching 1 depending on your choice of the algorithm parameters, a sufficiently close neighbor for most practical purposes.
- Although there now exist several variants of the ANN algorithms, the two leading candidates are based on Locality Sensitive Hashing (LSH) and Product Quantization (PQ).

Similarity Search (contd.)

- The last bullet on the previous slide mentioned two possibilities for ANN algorithms. Here is a link to my "tutorial implementation" of one of them, the LSH algorithm:

  `https://engineering.purdue.edu/kak/distLSH/LocalitySensitiveHashing-1.0.1.html`

- Although LSH is still a strong contender, for the sort of applications this lecture is all about, you are more likely to use an ANN algorithm based on Product Quantization (PQ). Probably the best introduction to PQ is in the paper in which it was first proposed:

  `https://lear.inrialpes.fr/pubs/2011/JDS11/jegou_searching_with_quantization.pdf`

- For its starting point, PQ uses what's now a very old idea — vector quantization of multi-dimensional data — and then extends it in such a way that allows NN search to be carried out efficiently in high dimensional vector spaces even when a dataset has billions of vectors.

- Before introducing you to PQ, on the next slide I want to get back to KD-Trees to quickly go over why we cannot use them in our case.

Similarity Search (contd.)

- Getting back to the case of KD-trees for NN search, it works well only when the dimensionality D of the data is small.

  [**Obviously, when $D = 1$, you have a simple binary tree for scalar data and the tree is theoretically guaranteed to return the nearest neighbor in time $O(log_2 N)$. In general, you are likely (but not guaranteed) to get the same sort of efficiency for NN search when D is small, but as the data dimensionality D becomes larger, you could end up exhaustively searching through the entire dataset of N vectors in the worst case as explained below.**]

- As to the main reason for why a KD-tree may not work well when D is large, it is that node splitting in the tree amounts to partitioning each axis of the vector space with an orthogonal hyperplane.

  [**As a result, all the data at the leaf nodes of the tree can be thought as residing in "boxes" whose sides are parallel to the coordinate hyperplanes of the vector space. With that kind of space splitting, as you descend down the tree with a query vector Q, you could end up at a leaf node where the data element is NOT the closest Euclidean neighbor of Q.**]

- Therefore, with KD-tree based NN search for vector data, after you have reached a leaf node, you must always do some backtracking to explore the other paths at the higher level nodes to see if the leaf nodes they lead to give you closer neighbors for Q. It is theoretically possible that, in the worst case, you may end up visiting all the nodes during the backtracking step.

Outline

Vector Quantization

- In this section, I'll be getting back to Product Quantization (PQ) mentioned in the previous section. Since PQ is an extension of the very old Vector Quantization (VQ) idea, it'd be good to first review what that means.

- VQ means to partition a vector space into Voronoi cells with respect to a set of points. These points, typically called the centroids of the cells in which they reside, are meant to represent all the vectors in the corresponding cells.
  [**That is, once you create such a partition, you could say that you are quantizing all of the vectors in a cell to the centroid for the cell. Unless you are already familiar with the idea, I suppose that begs the question: What is a Voronoi partition?**]

- To present the foundational idea of a Voronoi partition, more commonly known as the Voronoi diagram, consider a 2D plane and a set $S = \{p_1, p_2, \cdots, p_n\}$ of points in the plane. A Voronoi diagram vis-a-vis the set S of points is a partition of the plane into cells $C = \{c_1, c_2, \cdots, c_n\}$ such that all the points in cell c_i are closer to the point p_i than to any other point in set S.

Vector Quantization (contd.)

- If you really think about it, the definition presented on the previous slide can be translated into a quick procedure for constructing a Voronoi diagram in a plane: Let $h_{p,q}$ be the half-plane obtained by drawing a bisector of the line joining the points p and q from the set S. Since the bisector will create two half-planes, one on each side of the bisector line, you might ask: Which of the two half-planes is represented by the notation $h_{p,q}$? The notation represents the half-plane that contains the point p.

- For a given point p, the intersection of all such half-planes for all different possible values for q would constitute the Voronoi cell for the point p. This is illustrated in the figure for the next for the four points $S = \{p_1, p_2, p_3, p_4\}$ shown there.

- It follows from the construction of a Voronoi partition as presented above that a Voronoi cell will always be a convex region.

Voronoi Partitioning of a Plane

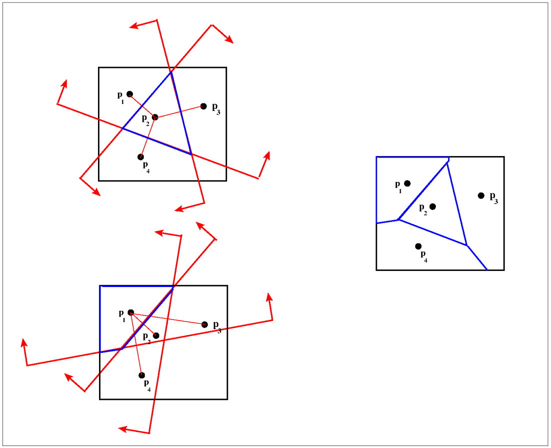


Figure: An example of the Voronoi diagram for the case of four points $\{p_1, p_2, p_3, p_4\}$ in a plane. Shown at upper left is the construction of the Voronoi cell for the point p_2. For the cell corresponding to p_2, you find the intersection of the half planes formed by the bisectors of the line segments that you get by joining p_2 with each of the other three points. You take the half-planes that contain the point p_2. The half-planes are depicted with long red lines and the part kept indicated with the arrows. The lower left shows us constructing the cell for the point p_1. You can do the same for the other two points, p_3 and p_4. The final Voronoi diagram is shown at right.

Vector Quantization (contd.)

- Now that you understand how you can go about constructing a Vornoi partition of the underlying vector space, we are still faced with the question as to where the points in the set S are supposed to come from. Typically, these points are the K centroids generated by applying the K-Means algorithm to the vector data.
- If you are not already familiar with the K-Means algorithm, you might want to read at least the Description section of the doc page of my Perl implementation of K-Means: `https://metacpan.org/pod/Algorithm::KMeans`
- So let's say you have applied K-Means to your data for a specified value for the integer K. The algorithm will attempt to partition the data into K clusters under the assumption that each cluster is a Gaussian with isotropic covariance.
- You feed the K points returned K-Means into Voronoi partitioning algorithm and, lo and behold, you now have a vector quantizer.

Vector Quantization for Dimensionality Reduction

- With a vector quantizer (VQ) constructed as described on the previous slide, you will be quantizing any vector value in the underlying vector space to one of the K cluster centers. The vectors that will get quantized to a specific cluster center returned by K-Means will be those that fall in the Voronoi cell corresponding to that cluster center.
- Given such a VQ, you could, in principle, significantly reduce the dimensionality of your multi-dimensional data, as explained below.
- Let's say that your police department has the 1024×1024 face images of hardcore criminals in your area. Let's also say that you represent each image with a 128-element embedding vector in the manner described earlier in this section.
- Let's say we vector quantize the space spanned by the real-valued 128-element embedding vectors to 2^B Voronoi centroids.

VQ for Dimensionality Reduction (contd.)

- To continue where I left off on the previous slide, subsequently, we will be able to represent all of the embedding vectors collected with a B-bit binary code. Obviously, all of the embedding vectors that fall in the same Voronoi cell will be mapped to the same code.
- Basically, this would amount to representing each 1024×1024 real-valued face image with a B-bit code. Depending on the value chosen for B, that could amount to a huge reduction in the end-point dimensionality of the image. The set of 2^B binary code words is referred to as the **Index Set**, denoted $\mathcal{I}$, for the database. You would obviously need to store a lookup table that would show the mappings from each codeword in $\mathcal{I}$ to the corresponding Voronoi centroid in the original vector space. The set of centroids is denoted $\mathcal{C}$.
- Subsequently, you'll be able to index the dataset of images by creating an inverse mapping from a given B-bit codeword to all the image pathnames that map to the same codeword. Such a lookup table is referred to as the Inverted Index for a dataset.

VQ for NN Retrieval Using the Inverted Index

- The NN search problem can now be solved more efficiently by first computing the B-bit mapping for the Query image, reaching into the Inverted Index to access all the dataset images that map to the same code, actually measuring the Euclidean distance between the embedding vector of the query Q and each of the embedding vectors pointed to by the Inverted Index, and, finally, returning the image whose embedding vector is closest to that of the Query image Q.

- The main issue then is what value one should choose for the size B of the codewords for the Voronoi centroids. For the sizes of datasets that call for metric learning, it is believed you need at least 64-bit codewords. That is, you would want to set B to 64.

- Setting B to 64 implies having to reliably calculate 2^{64} Voronoi centroids with the K-Means algorithm. In general, on account of the challenges created by what's loosely referred to as "curse of dimensionality", applying K-Means directly at such a scale is not feasible — as argued on the next slide.

Impracticality of Using VQ Directly for NN Retrieval

- As mentioned on the previous slide, using $B = 64$ translates into 2^{64} centroids in the vector space spanned by the embedding vectors what would need to be learned by a K-Means algorithm. That would be way beyond the ability of any K-Means that has ever been developed. Just imagine how many training images you would need. Even if you could get away with the assumption that, on the average, you will need only 10 images per Vornoi cell, you are still talking about an astronomical amount of data and possibly impractical training time.

- This is where Product Quantization comes to our rescue — an idea that was first proposed in the following publication:

  https://lear.inrialpes.fr/pubs/2011/JDS11/jegou_searching_with_quantization.pdf

- The main idea in Product Quantization is to chop up the embedding vectors into smaller segments and to then subject each segment separately to vector quantization.

Product Quantization for NN Retrieval

- To elaborate, let's say the embedding vectors are 128-elements long. That is, each embedding vector consists of 128 real-valued numbers. For Product Quantization (PQ), we divide each such vector into *m* segments:

```
          ----------------------------------------------------------------------------------
         |0|1| ... |7|8| ... |15|16| ... |23|24| ... |31|32|      ..............      [126|127 |
          ----------------------------------------------------------------------------------

         <----------><---------><----------><----------><----------->     ...     <----------->

m:           0          1          2          3          4                            15
```

  That is, we will consider each embedding vector to be composed of 16 smaller segments, each involving only 8 real numbers.

- The integers shown in the topmost row of such values are the element index values in a single embedding vector that has 128 real valued elements in it. In the depiction, we have $m = 16$ for the 16 segments, and each segment consists of k elements. We have $k = 8$.

Product Quantization for NN Retrieval (contd.)

- Since, as depicted on the previous slide, each segment of the embedding vector spans only 8 real values, it is reasonable to talk about a 4-bit code for the Voronoi centroids in the 8-dimensional space of reals for each segment.

- So overall, we will still end up with a 64-bit binary code word quantization of each original 128-element embedding vector.

- In general, with PQ, we can talk about separate Index Sets, denoted $\mathcal{I}_i$ for the i^{th} segment of the embedding vectors. And we can talk about the codewords in $\mathcal{I}_i$ mapping to the set $\mathcal{C}_i$ of Voronoi centroids in the D/m-dimensional real space where D is the dimensionality of the original embedding vectors and m the number of segmentations of the embedding vectors.

- The overall representation achieved in this manner for the original embeddings vectors can be considered to reside in two Cartesian Product spaces, $\mathcal{I}$ and $\mathcal{C}$, as explained on the next slide.

PQ for NN Retrieval (contd.)

- The two Cartesian Product spaces are defined by $\mathcal{I} = \mathcal{I}_0 \times \mathcal{I}_1 \ldots \mathcal{I}_{m-1}$ for the indexes and $\mathcal{C} = \mathcal{C}_0 \times \mathcal{C}_1 \ldots \mathcal{C}_{m-1}$ for the Voronoi centroids.

- To create PQ based representation for an embedding vector, we first map the each i^{th} segment, $i = 0, 1, \ldots, m-1$, of the D-dimensional real numbers to its binary code in the Index Set $\mathcal{I}_i$. We subsequently concatenate all the D/m-bit code words together to form the overall code word representation for in the input embedding vector.

- We refer to the vector quantizers for each of the m segments of the embedding vectors as *subquantizers*. We assume that binary codewords produced by each subquantizer are based on k^* bits. That is, each subquantizer will map its D/m-dimensional real valued input to one of 2^{k^*} Voronoi centroids.

Outline

The FAISS Library for Similarity Search

- The acronym FAISS stands for “Facebook AI Similarity Search” and it was presented to the world in the following 2017 publication:

  `https://arxiv.org/pdf/1702.08734.pdf`

- Most folks in machine learning would say that Faiss is the best implementation of the Product Quantization approach (discussed earlier in this lecture) to similarity search. Faiss is implemented in C++ and comes with Python bindings.

- The heart of Faiss is its “indexer”. If you can assume that your vectors are going to be more or less uniformly distributed in the underlying vector space, you are likely to construct the indexer with the following sort of a call:

  ```
  indexer  =  faiss.IndexFlatL2( embedDim )
  ```

  where `embedDim` is the dimensionality of the embedding vectors.

Similarity Search with Faiss (contd.)

- The "L2" in the name of the indexer shown on the previous slide means that it will use the L_2 distance metric for indexing. Given the overall dimensionality of the vector space in which the embeddings reside, it is indexer's job to decide what value to use for m on Slide 75. Recall, m is the number of "sub-vectors" created from each embedding vectors and then vector quantization carried out separately in the subspaces formed by the sub-vectors.

- Here is a useful page by one of the authors of Faiss. It's a highly readable tutorial intro to the material:

  `https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/`

- Here is a link to the main documentation page:

  `https://faiss.ai/index.html`

  As you will see, there is a lot more indexing than what I have mentioned in my very brief intro to Faiss.

Similarity Search with Faiss (contd.)

- Shown below is a toy example that should make you a bit more familiar with the syntax to use when calling on Faiss to return the nearest neighbors of a given query vector:

```
import faiss
import numpy as np
np.random.seed(0)

embedDim = 10           ## dimensionality of the vectors to be compared for similarity
dataset_size = 100
how_many_query_vecs = 1
how_many_nearest_neighbors = 3

##  Create a dataset vectors of the specified embedding dimension:
X = np.random.rand(dataset_size, embedDim).astype('float32')
## Create the indexer:
indexer = faiss.IndexFlatL2(embedDim)
##  Populate the vector space of the indexer with the data:
indexer.add(X)
##  Let's now specify a single query, that is, a single vector of floats
##  whose dimensionality is also embedDim:
queries = np.random.rand(how_many_query_vecs, 10).astype('float32')
##  Let's ask for the specified number of nearest neighbors for each
##  query vector:
D, I = indexer.search(queries, k=how_many_nearest_neighbors)
print("\n\nInteger indexes of the nearest neighbors:", I)
print("\n\nDistances to the nearest neighbors:", D)

##  Answer returned:
##      Integer indexes of the nearest neighbors: [[35 10 50]]
##      Distances to the nearest neighbors: [[0.42326236 0.6387429  0.67744243]]
```

Outline

Visualizing the Clusters Created in a High-Dimensional Space

- Metric learning imposes a similarity structure on the data, in the sense that it will attempt to pull together the data samples with the same class labels and push apart the samples with different class labels. [**To be more precise, metric learning groups together similar training samples even if that means that all the data corresponding to the same class be represented by multiple clusters.**]

- After you have trained a network to carry out metric learning using a training dataset, is there an easy way to tell how well the network is doing its job? Yes, you can run your testing dataset through the network and use one of the modern data visualization algorithms like t-SNE and UMAP to see the result.

- The magic of algorithms like t-SNE and UMAP is they can take a data distribution in a high dimensional space and then, for the purpose of visualization, create a 2-dimensional (or 3-dimensional) version of the original data distribution in which the inter-cluster relationships (in terms of the distances between the clusters) are substantially preserved. [**Given the huge reduction in the dimensionality when you go from the original data to what is visualized, the algorithms do not carry a theoretical guarantee that the final result will faithfully mimic all the intra- and inter-cluster relationships of the original data space, but the experiments bear out that if the data clusters in the original space are well separated, they will also be well separated in the visualization space.**]

Visualizing the Clusters (contd.)

- The two algorithms that are currently the most popular for visualizing similarity structures in high dimensional spaces are t-SNE and UMAP.
- t-SNE is an extension of the original SNE algorithm and the acronym SNE stands for "Stochastic Neighbor Embedding".

  [**SNE means creating low-dimensional vector representations (embeddings) for the original data that preserve the distance relationships between the neighbors in a probabilistic sense. The prefix "t-" in the name t-SNE is for replacing the Gaussian model for the clusters in the visualization space with the Student-t distribution.**]
- Here is the link to the paper "Visualizing Data using t-SNE" by van der Maaten and Hinton that presents the t-SNE algorithm (and also has a great description of the SNE algorithm that came out earlier):

  `https://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf`
- Here is a link to a paper by McInnes, Healy, and Melville that describes the more recently released UMAP algorithm for data visualization. This algorithm is presumably more scalable to very large datasets and has a faster runtime performance:

  `https://arxiv.org/pdf/1802.03426.pdf`

SNE's Conditional Probabilities for Modeling Neighborhoods

- Fundamental to the SNE algorithm is the idea that associated with each data point in the underlying vector space is its "sphere of influence". Given a knowledge of this sphere of influence at the i^{th} point $\mathbf{x}_i$, we can estimate as to what extent another point $\mathbf{x}_j$ can be considered to be a neighbor of $\mathbf{x}_i$.
- We will use the scalar parameter σ_i to denote the sphere of influence at the point $\mathbf{x}_i$.
- And, we will use *conditional probabilities* to capture what is conveyed by the concept of sphere of influence.
- The conditional probability $p_{j|i}$ measures the extent to which a given point $\mathbf{x}_i$ considers another point denoted $\mathbf{x}_j$ to be its neighbor. This conditional probability is given by:

$$p_{j|i} \quad = \quad \frac{e^{-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{2\sigma_i^2}}}{\sum_{k \neq i} e^{-\frac{||\mathbf{x}_i - \mathbf{x}_k||^2}{2\sigma_i^2}}} \tag{5}$$

SNE's Conditional Probabilities (Contd.)

- The denominator in the expression shown on the last slide is just for normalization so that the value returned by the expression shown has the correct probability semantics. We assume that $p_{i|i} = 0$ since the focus is on comparing *different* data points.

- We are obviously modeling the conditional probability $p_{j|i}$ as a Gaussian distribution over the Euclidean distance between the given point $\mathbf{x}_i$ and the other point $\mathbf{x}_j$ that is under consideration for belonging to the same cluster as the given point.

- The key to estimating $p_{j|i}$ at any given point $\mathbf{x}_i$ is getting hold of the corresponding variance σ_i. This variance is a measure of the tightness of the distribution of the data points in the neighborhood of $\mathbf{x}_i$. [**The neighborhood property I am talking about is supposed to be class agnostic. That is, we are NOT talking about how each object class in our dataset is distributed in the space of embedding vectors — but only about how all the points are distributed in the space. Some authors refer to how all the points are distributed vis-a-vis one another as the structure of the overall data. When we map the data to, say, a 2-dimensional space for visualization, we want to preserve this structure.**]

- Estimating σ_i is a story unto itself, as explained on the next slide.

Estimating σ_i for a Given $\mathbf{x}_i$

- An intuitive way to estimate σ_i at $\mathbf{x}_i$ would be fan out from the point and count the number of the other same-category points at different distances. Since this sort of a calculation can be expensive, the authors of SNE introduced a user-supplied parameter called *Perplexity* that significantly expedites the estimation of σ_i.

- Perplexity is meant to be *proportional* to the number of the neighbors of $\mathbf{x}_i$ that are within its σ_i distance. What that implies is that as you go outbound from $\mathbf{x}_i$, you keep track of how many of the neighbors you encounter at different distances from $\mathbf{x}_i$. You stop when you have reached the count signified by the value of Perplexity. The distance you had to go outbound to reach the Perplexity count is proportional the value of σ_i.

- The actual logic that is implemented in SNE for estimating σ_i at each data point $\mathbf{x}_i$ involves binary search as explained on the next slide.

SNE: Binary Search for σ_i

- The best value for σ_i is one that yields a value for the entropy H_i associated with the conditional distribution $p_{j|i}$ (considered as a probability distro over the index j for a given index i) so that 2^{H_i} equals the user supplied perplexity. As you would guess, the entropy H_i has the usual definition:

$$H_i \quad = \quad -\sum_j p_{j|i} \cdot \log_2 p_{j|i} \tag{6}$$

- That is, given the user supplied value for the perplexity and with the entropy H_i calculated with our estimate for σ_i, we want the following equation to be satisfied:

$$user\_supplied\_perplexity \quad = \quad 2^{H_i} \tag{7}$$

- These observations translate into the following binary search algorithm for the correct value for σ_i: **[Binary search starts with two user-specified values, one for the lower-bound and the other for the upper-bound that prescribe the search range for σ. The same bounds are used for all values for the index i. In the original implementation provided by the authors of SNE, the initial values for the two bounds were set to $1e-20$ and 10000. You take the midpoint of this range as your first guess for σ. If the user supplied value for perplexity is less than this midpoint, you change the upper-bound to the midpoint while keeping the lower-bound unchanged. On the other hand, should the user-supplied value for perplexity be greater than the midpoint, you change the lower-bound to the midpoint, while keeping the upper-bound unchanged. You iterate in this manner until you are close enough to the user supplied value for the perplexity.]**

The Distance Matrix (Again)

- The pairwise Euclidean distances that you need in Eq. (5) for the conditional probabilities $p_{j|i}$ for all values of the indexes i and j constitutes the same distance matrix you saw earlier on Slide 50.

- Let's say you have used your labeled training dataset to train a metric-learning network and now you would like to get a sense of how well the network is working. [**This you could do by feeding the embedding vectors produced for the images in the test dataset into a data visualization algorithm like SNE. Let's say that you have 1000 test images and that the network puts out a 128-dimensional embedding vector for each image.**]

- For the SNE algo to do its job, the very first thing it would need to do is to compute the distance matrix required by the conditional probs in Eq. (5) on Slide 83. Assuming that you are using numpy for the visualization code, as previously explained on Slide 50, all you would need to do is to execute the following command in which X is a numpy array of shape $(1000, 128)$ made up of the embedding vectors produced by the network:

```
   distance_matrix   =   numpy.sum(  (X[None, :]  -  X[:, None])**2,  2 )
```

Mapping the Original Data Points to Their Low-Dimensional Counterparts

- So far I have only talked about how to model the neighborhoods in the space of the embedding vectors. Given any point indexed i in the space, we use Eq. (5) on Slide 83 for the conditional probability that another point j in the space is i's neighbor.
- That brings us to the key question in SNE: How to map the points in their native high-dimensional space to a low-dimensional visualization space so that the structure of the data is preserved?
- Let $\mathbf{y}_i$ represent the low-dimensional counterpart of the original data point $\mathbf{x}_i$. As with the original data, let the conditional probability $q_{j|i}$ measure the extent to which a given point $\mathbf{y}_i$ considers another point $\mathbf{y}_j$ to be its neighbor. We model this conditional probability by:

$$q_{j|i} \quad = \quad \frac{e^{-||\mathbf{y}_i - \mathbf{y}_j||^2}}{\sum_{k \neq i} e^{-||\mathbf{y}_i - \mathbf{y}_k||^2}} \tag{8}$$

Mapping to Low-Dimensional Counterparts (contd.)

- There is one big difference between the conditional probs in Eq. (5) on Slide 83 and those in Eq. (8) on the previous slide: The assumption that the neighborhood variances at each of the mapped points have been assumed to be the same. That is, if we were to use ${\sigma'}_i$ to represent the mapped data's counterpart of the original data's σ_i, we are assuming that all ${\sigma'}_i$'s are the same and equal to $\frac{1}{\sqrt{2}}$.

- To see the merit of the above assumption, consider the case when the object class distributions in the original high-dimensional space are well separated. Now focus on just a single cluster in the original space and consider it to be distributed as a Gaussian: [**The points in the vicinity of the center of the cluster will be closer together than the points near the periphery. So if σ_i is inversely proportional to the number of neighbors within, say, a unit sphere around the point $\mathbf{x}_i$, the value of σ_i will increase as you go from the cluster center to its periphery.**]

- This assumption about the mapped points implies that the cluster spread will be more or less the same for the mapped points. That implies that SNE is more concerned about maintaining the structure between the clusters rather than within the individual clusters.

Mapping to Low-Dimensional Counterparts (contd.)

- The problem of mapping the high-dimensional points, $\mathbf{x}_i$, to their low-dimensional counterparts $\mathbf{y}_i$ can now be stated as follows: Find the coordinate values for the mapped $\mathbf{y}_i$ points so that the conditional probs $\{p_{i|j}, i, j = 0, 1, 2, ..., N\}$ are as close as possible to the conditional probs $\{q_{i|j}, i, j = 0, 1, 2, ..., N\}$.

- In general, given two probability distributions $P = \{p_i, i = 1, 2, \ldots, N\}$ and $Q = \{q_i, i = 1, 2, \ldots, N\}$, we can use KL-Divergence to estimate how close the two are to each other. [See the explanation of KL-Divergence in Slides 17 through 22 of my Week 11 lecture on "Generative Data Modeling with Networks Based on Adversarial Learning and Denoising Diffusion'.] In general, we assume that p_i's are the true distribution and q_i's the estimated approximations thereof. The KL-Divergence between the two is given by:

$$d_{KL}(P, Q) \quad = \quad \sum_{i=1}^{N} \mathbf{p}_i \log_2 \frac{\mathbf{p}_i}{\mathbf{q}_i} \tag{9}$$

Mapping to Low-Dimensional Counterparts (contd.)

- The formula shown at the bottom of the previous slide can also be used to find the divergence between the two conditional distributions as given by $p_{j|i}$'s and $q_{j|i}$'s for a given point $\mathbf{x}_i$:

$$d_{KL}(\{\mathbf{p}_{j|i}\}, \{\mathbf{q}_{j|i}\}) \quad = \quad \sum_{j=1}^{N} \mathbf{p}_{j|i} \log_2 \frac{\mathbf{p}_{j|i}}{\mathbf{q}_{j|i}} \tag{10}$$

  where the notation $\{\mathbf{p}_{j|i}\}$ means the conditional prob distribution formed by the conditional probs $\mathbf{p}_{j|i}$, etc.

- What's shown above is for a single chosen point indexed i in the original data space. For constructing an overall cost function, designated $\mathcal{C}$, for all the data points in the original space, we must add the KL divergences at all of them:

$$\mathcal{C} \quad = \quad \sum_{i=1}^{N} \sum_{j=1}^{N} \mathbf{p}_{j|i} \log_2 \frac{\mathbf{p}_{j|i}}{\mathbf{q}_{j|i}} \tag{11}$$

- Our goal is to find those coordinate values for the mapped points $\mathbf{y}_i$ for $i = 1, 2, \ldots, N$ that minimize the cost $\mathcal{C}$.

Mapping to Low-Dimensional Counterparts (contd.)

- For our purposes, we express Eq. (11) in the following form:

$$\begin{aligned} \mathcal{C} &= \sum_{i=1}^{N}\sum_{j=1}^{N} \mathbf{p}_{j|i}[\log_2 \mathbf{p}_{j|i} - \log_2 \mathbf{q}_{j|i}] \\ &= \sum_{i=1}^{N}\sum_{j=1}^{N} \mathbf{p}_{j|i} \log_2 \mathbf{p}_{j|i} - \sum_{i=1}^{N}\sum_{j=1}^{N} \mathbf{p}_{j|i} \log_2 \mathbf{q}_{j|i} \end{aligned} \tag{12}$$

- Note the minimization of $\mathcal{C}$ reduces to the minimization of the second term at right in Eq. (12), since the **y** coordinates only show up in that term through the conditional probs $q_{j|i}$. The second term on the right in Eq. (12) are the cross-entropies of the approximating distributions as represented by $\{q_{j|i}\}$ vis-a-vis the entropies associated with the original data that is represented by the first term in Eq. (12).

- By substituting Eq. (8) in Eq. (12) and taking the partial of the result with respect to the variables $\mathbf{y}_i$, we can write the following for a gradient descent based solution for the $\mathbf{y}_i$'s:

$$\frac{\partial \mathcal{C}}{\partial \mathbf{y}_i} = 2\sum_{j=1}^{N} \left(\mathbf{p}_{j|i} - \mathbf{q}_{j|i} + \mathbf{p}_{i|j} - \mathbf{q}_{i|j} \right)(y_i - y_j) \tag{13}$$

Extending to t-SNE

- The basic idea of t-SNE is the same as that for its predecessor algorithm SNE: [**You want to model the similarities between the data points in the high-dimensional space by probabilities that depend on the Euclidean distance between them on a pairwise basis. And, when you map these points to their 2-dimensional counterparts, you want to model the mapped points similarly. To estimate the coordinates of the mapped points in the 2D visualization space, you want to minimize a cost function that measures the divergence between the two probabilistic models.**]

- The main differences between the SNE and t-SNE are with regard to the probability model used for the points in the high-dimensional space and to the one used in the visualization space.

- Instead of using conditional probabilities, t-SNE started out by experimenting with the following probabilities on a pairwise basis.

$$p_{ij} = \frac{e^{-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{2\sigma^2}}}{\sum_{k \neq l} e^{-\frac{||\mathbf{x}_k - \mathbf{x}_l||^2}{2\sigma^2}}} \tag{14}$$

$$q_{ij} = \frac{e^{-||\mathbf{y}_i - \mathbf{y}_j||^2}}{\sum_{k \neq l} e^{-||\mathbf{y}_i - \mathbf{y}_l||^2}} \tag{15}$$

Extending to t-SNE (contd.)

- The joint probability $\mathbf{p}_{ij}$ measures the extent to which a pair of data points $\mathbf{x}_i$ and $\mathbf{x}_j$ belong together on account of their similarities as measured by the Euclidean distance between them. The joint probability $\mathbf{q}_{ij}$ has the same interpretation in the visualization space.

- Superficially, the joint probabilities shown on the previous slide look similar to the conditional probabilities in Eqs. (5) and (8) for the SNE algorithm — since the numerators are identical in the two cases. Note, however, the denominators are radically different. [**For SNE, the denominators provide normalization for each point indexed i separately. On the other hand, for t-SNE, the normalizations are all the same for every pair of points chosen. In that sense, it is less expensive to compute the joint probabilities as opposed to the conditional probabilities.**]

- The formulas in Eqs. (14) and (15) are just the starting points for thinking about modeling the similarity relationships between the data points in their respective spaces. Here is the problem with the formula in Eq. (14): [**According to the authors of t-SNE, when the high-dimensional data contains outliers, the pairwise distances between such points and the rest of the data may be so large that the resulting pairwise joint probabilities are much too low. When that happens, the corresponding mapped points contribute very little to the overall cost function. As a result, the minimization process ceases to be effective for the placement of such mapped points.**]

Extending to t-SNE (contd.)

- The following alternative formulation of the joint pairwise probabilities for the high-dimensional data takes care of the problem on the previous slide:

$$p_{ij} = \frac{\mathbf{p}_{i|j} + \mathbf{p}_{i|j}}{N} \tag{16}$$

- When you use the joint distributions as described above, the formula for the gradient-descent solution for $\mathbf{y}_j$'s becomes:

$$\frac{\partial \mathcal{C}}{\partial \mathbf{y}_i} = 4\sum_{j=1}^{N}(\mathbf{p}_{ij} - \mathbf{q}_{ij})(\mathbf{y}_i - \mathbf{y}_j) \tag{17}$$

- About the formula in Eq. (15) for the mapped points, it can lead to what's referred to as the "crowding problem", that is, the propensity of the mapped points in the 2D visualization space to crowd up close to the centers of the clusters. The main cause for this is that, in general, a Gaussian exerts a pull towards the mean on the modeled data. [**Again, in general, in the high-dimensional space, the data is likely to reside on a manifold of significantly lower dimensionality, but one that is much larger than that of the 2D visualization space. It would not be too difficult to imagine configuration of points on the manifold that cannot be mapped into the 2D space without seriously violating the pairwise relationships between the points. Under such conditions, a Gaussian is likely to pull the otherwise unmappable points toward the mean.**]

Extending to t-SNE (contd.)

- The t-SNE algorithm solves the crowding problem by using the t-distribution (more commonly known as Student's t-distribution) in place of the Gaussian distribution shown in Eq. (14). The t-distribution has a particularly simple form for the univariate case:

$$f(t) \quad = \quad \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}\Gamma(\frac{\nu}{2})}\left(1+\frac{t^2}{\nu}\right)^{-(\nu+1)/2} \tag{18}$$

  where Γ is the Gamma function (a generalization of the factorial to real numbers), and where the integer-valued parameter ν, known as the "Degrees of Freedom", controls the width of the distribution and also how heavy-tailed the distribution is.

  [**See the Wikipedia page on "Student's t-distribution" for how ν changes the shape of the distribution. As you will see there, the smaller the value of ν, the heavier the tails of what superficially looks like a Gaussian distribution. You get the heaviest tails with $\nu = 1$ and the t-distribution approaches the Gaussian as $\nu \to \infty$. Note that the heavier the tails, the greater the ability of the distribution to accommodate what would otherwise be considered as outliers. As you would expect, estimation based on Gaussian distributions can give very wrong answers in the presence of such outliers, especially when the outliers are not really outliers but simply the more rarely occurring instances of the phenomenon being modeled. So if the physical phenomenon being modeled produces data that looks like it came from a Gaussian most of the time, but every once in a long while it produces a data value very far from the mean of the data generally produced but a value that is still legitimate and representative of the same phenomenon, you have no choice but to use the t-distribution. There are some computational challenges associated with using a distribution that my lab has addressed in the following publication:**

  `https://engineering.purdue.edu/RVL/Publications/Aeschliman2010ANovel.pdf`

  ]

Extending to t-SNE (contd.)

- When we assume $\nu = 1$, the form shown on the previous slide becomes to a Cauchy distribution. This is the form used in the t-SNE algorithm:

$$f(t) \quad = \quad \frac{1}{1 \; + \; t^2} \tag{19}$$

- When the above form is used for modeling the pairwise joint probabilities in the visualization space, we get:

$$q_{ij} \quad = \quad \frac{\left(1 \; + \; ||\mathbf{y}_i - \mathbf{y}_j||^2\right)^{-1}}{\sum_{k \neq l} \left(1 \; + \; ||\mathbf{y}_k - \mathbf{y}_l||^2\right)^{-1}} \tag{20}$$

- With Eq. (17) for the joint pairwise probabilities for the data in the high-dimensional space and the above equation for the same in the visualization space, the formula for the gradient descent solution for $\mathbf{y}_j$'s becomes:

$$\frac{\partial \mathcal{C}}{\partial \mathbf{y}_i} \quad = \quad 4 \sum_{j=1}^{N} \frac{(\mathbf{p}_{ij} \; - \; \mathbf{q}_{ij})(\mathbf{y}_i \; - \; \mathbf{y}_j)}{1 \; + \; ||\mathbf{y}_i - \mathbf{y}_j||^2} \tag{21}$$

Outline

The MetricLearning Co-Class in DLStudio

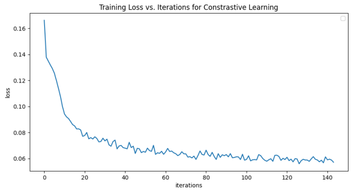
- Starting with Version 2.3.2, the DLStudio platform includes a new module called `MetricLearning`. You can scan through its code in your browser just by clicking on one of the links near top of the webpage for DLStudio.

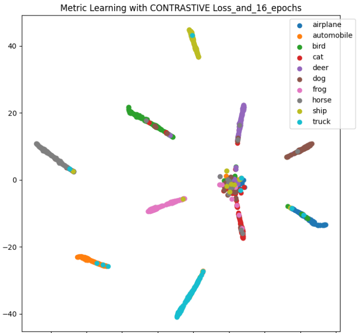
- The `ExamplesMetricLearning` contains the following two scripts that demonstrate the results you can get with my code on the CIFAR-10 dataset:
  1. example_for_pairwise_contrastive_loss.py
  2. example_for_triplet_loss.py

- As the names imply, the first script demonstrates using the Pairwise Contrastive Loss for metric learning and the second script using the Triplet Loss for doing the same. Both scripts can work with either the pre-trained ResNet-50 trunk model or the homebrewed network supplied with the MetricLearning module.

Results with Contrastive Learning on CIFAR-10

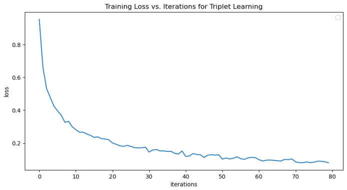
- Shown below are the training loss and the clustering achieved on a collection of images from the test dataset after the network was trained on CIFAR-10 dataset with Contrastive Loss over 16 epochs.
- With Pairwise Contrastive learning (and with no hyperparameter tuning of any sort), you get an accuracy of around 74% for `Precision@1`. Also, I used the default value for the margin in the loss function, which is probably the worst thing to do in metric learning.

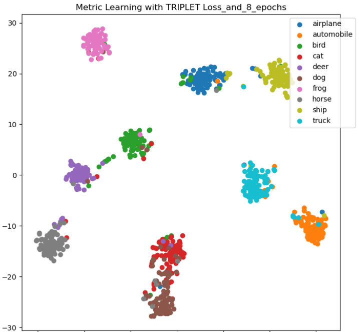




Results with Triplet Learning on CIFAR-10

- Shown below are the training loss and the clustering achieved on a collection of images from the test dataset after the network was trained on the CIFAR-10 dataset with Triplet Loss over 8 epochs.
- With Triplet learning (and with no hyperparameter tuning of any sort), you get an accuracy of around 84% for `Precision@1`. Again, I used the default value for the margin in the loss function, which is probably the worst thing to do in metric learning.


Training Loss vs. Iterations for Triplet Learning
loss
iterations


Metric Learning with TRIPLET Loss_and_8_epochs
airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck