Metric Learning with Deep Neural Networks

Avinash Kak

Purdue University

Lecture Notes on Deep Learning by Avi Kak and Charles Bouman

Monday 1st December, 2025 22:46

Preamble

To TOC To HowTo

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.

Such learned vector representations are commonly referred to as "embeddings" or "embedding vectors" because you are embedding the data objects in a new vector space with the expectation that these alternative representations of the data objects would exhibit the properties mentioned above.

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*.

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 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 cluster of a multi-modal representation of the face images for any single individual.**

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. That begs the question: 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 — and **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 6 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 a computationally efficient nearest-neighbor classifier or even a nearest-cluster-center classifier**.

Preamble

As you can see in the TOC in Slide 14, this lecture consists of **four parts**. The main goal in **Parts 1 and 2** 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**. And the main goal in **Parts 3 and 4** is to introduce you to Representation Learning (which is mostly unsupervised) that draws heavily on Metric Learning concepts.

The Pairwise Contrastive Loss was introduced 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** 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 two loss values that are calculated *separately* for the positive pairs and for the negative pairs. For the positive pairs, the loss 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 loss calculation for the negative pairs in Pairwise Contrastive Loss is made a bit complicated by the fact that 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 in a negative pair. 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

For formulating the **Triplet Loss**, you again first extract all the Positive Pairs in a batch in the same manner as for the Pairwise Contrastive Loss. However, now, in each Positive Pair, you refer to one sample as the Anchor, denoted $Anchor$, while denoting the other sample as Pos because it has the same label as the Anchor.

Next, for every $(Anchor, Pos)$ pair, you collect all the samples in the batch for which the classification labels are different from the label in the $(Anchor, Pos)$ pair. These would constitute the Negative Set vis-a-vis the pair in question. Using the Negative Set, you form the triplets $(Anchor, Pos, Neg)$ provided $dist(Anchor, Neg)$ satisfies certain conditions vis-a-vis $dist(Anchor, Pos)$ where $dist()$ is the distance between the embedding vectors for the samples.

Creating $(Anchor, Pos, Neg)$ triplets from a batch 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.

From the set of triplets constructed, you formulate the Triplet Loss using the difference of the distance between the Anchor and the Positive, on the one hand, and the distance between the Anchor and the Negative, on the other, as I'll explain in this lecture.

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

To end the discussion in Parts 1 and 2, 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

In **Parts 3 and 4**, I'll shift my focus to unsupervised Representation Learning that, one could say, sits on top of the basic ideas of Metric Learning.

The ultimate goal of Representation Learning is to endow a neural network to learn a vocabulary for the evidence that sits in its various layers as the data flows from the input layer to the final output where the decisions are made. What's critical here is that a neural network be allowed to suggest entities that may not look familiar to humans — since human cognition and neural-network are not congruent.

Most current approaches to Representation Learning are based on unsupervised discovery of data entities through the maximization of Mutual Information between candidate entities and the data elements.

The loss functions that are commonly used for this purpose are those that are inspired by Noise Contrastive Estimation, the two most popular loss functions being InfoNCE and PatchNCE.

I'll end Parts 3 and 4 by introducing you to CLIP that carries out multi-modal metric learning.

Preamble

Finally, here is a link to our own publication on metric learning at CVPR 2024:

`https://arxiv.org/abs/2403.00272`

And here is a photo of a happy Rohan Sarkar on whose PhD dissertation this nominated-for-award paper was based:

`https://engineering.purdue.edu/kak/RohanCVPR.html`

The paper demonstrates that you can significantly improve the performance of a classifier based on metric learning if both the category-based and the object-identity-based embeddings are learned simultaneously during training. In hindsight, that sounds intuitive because learning about the categories is more fundamental than learning about the individual objects that correspond to those categories.

Preamble – How to Learn from These Slides

Since it is a large slide deck, you may need some help with how to digest all the information that is presented here. To that end, at your first reading, please focus on just the slides I have highlighted below.

- A good understanding of the Pairwise Contrastive Loss and Triplet Loss is fundamental to understanding this lecture. **At your first reading, just focus on the Slides 19 through 34 for these two concepts.**
- About the material in Slides 35 through 60, at your first reading, you only need to know **why** you must write purely tensor-based code for batch mining (as opposed to, say, using `for`-loops). **That is, initially, it would be sufficient if you go over the material on just the Slides 35 through 40.**.
- The material on Slides 61 through 82 deals with what's known as Similarity Search — that is, how to find the learned embedding vector that is closest to a query embedding vector. **For your first reading, it would be sufficient if you just go over the material in Slides 60 through 66.**

That makes for a total of 26 slides you need to focus on at your first reading of this lecture. Obviously, after you have become comfortable with the fundamental concepts of metric learning, I would want you to go over the rest of the material — especially the material that addresses the coding challenges involved and the subject of Product-Quantization based similarity search

Outline

Outline

Representing Images with Embeddings

- Fundamental to metric learning is representing each input image by a learnable vector of user-specified dimensionality. For example, we may wish to represent each image in a dataset of face images by a 256-dimensional vector (regardless of the size of the images).

- We refer to such vectors as *embeddings* or *embedding vectors* since we are "embedding" the image in an underlying vector space.

- It is interesting to keep in mind the fact that the more traditional machine learning would also represent an image as a point in a vector space. Those vector spaces are, however, defined by a set of user-specified features that are believed to provide the needed discriminations between the images. The embedding vectors, on the other hand, are vectors whose elements are learned by minimizing a loss and with stochastic gradient descent.

Representing Images with Embeddings (contd.)

- 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.
- 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):

$$d(f(\mathbf{x}), f(\mathbf{y})) \quad = \quad ||f(\mathbf{x}) - f(\mathbf{y})||_2 \tag{1}$$

Representing Images with Embeddings (contd.)

- The subscript 2 on the vector norm shown in the formula at the bottom of the previous slide 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:

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

- 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

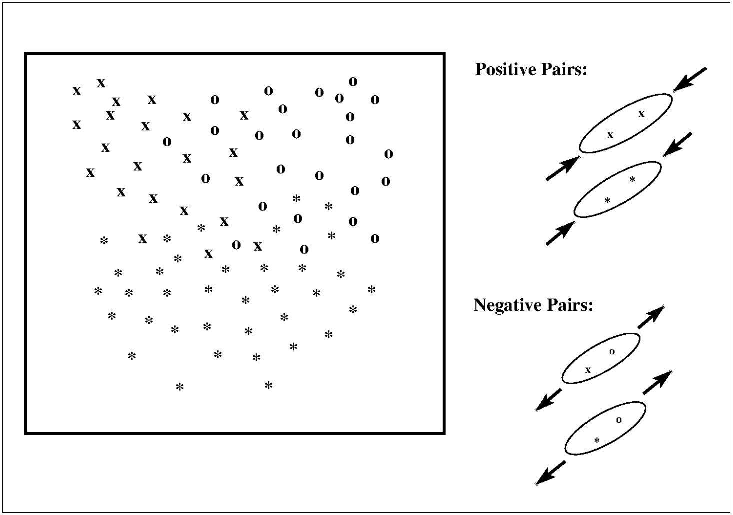
Pairwise Contrastive Loss

- 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:

$$L_p \quad = \quad \sum_i \left[d\Big(f(\mathbf{x}_1^i), f(\mathbf{x}_2^i) \Big) \right]^2 \qquad (3)$$

  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:

$$\max\left\{0,\ m - d\Big(f(\mathbf{x}_1^j), f(\mathbf{x}_2^j)\Big)\right\} \tag{4}$$

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:

$$L_n \;=\; \sum_j \left[\max \left\{ 0,\; m - d\Big(f(\mathbf{x}_1^j), f(\mathbf{x}_2^j) \Big) \right\} \right]^2 \tag{5}$$

  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:

$$\mathcal{L} \;=\; \sum_i \;\; (1 - y_i) \cdot \left[d\left(f(\mathbf{x}_1^i), f(\mathbf{x}_2^i) \right) \right]^2 \;+\; y_i \cdot \left[\max\left\{ 0,\; m - d\left(f(\mathbf{x}_1^i), f(\mathbf{x}_2^i) \right) \right\} \right]^2 \quad (6)$$

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.)

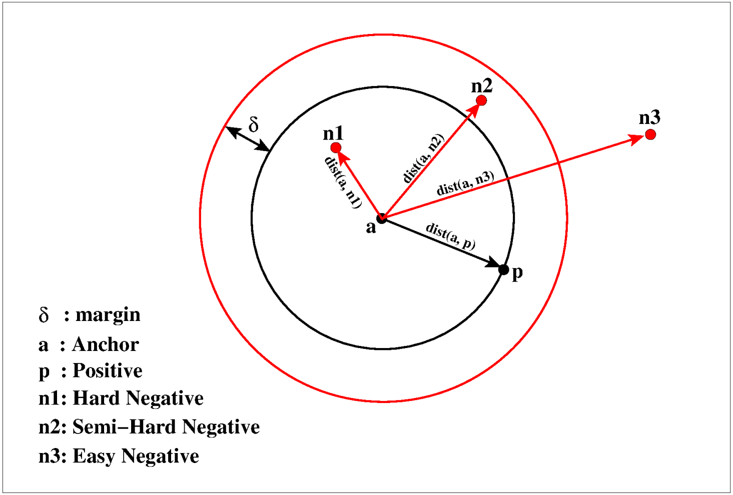
- What's interesting is that not all the negative samples $n \in \mathcal{N}$ carry the same level of significance with regard to how informative they are for the learning that is required.

- 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.

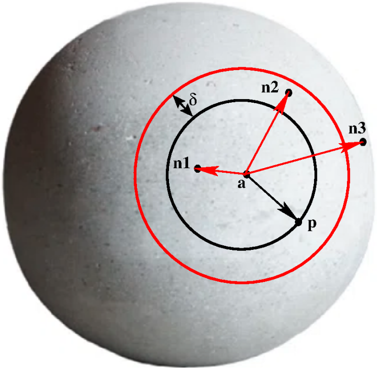
Triplet Loss (contd.)

- 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 29 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 29.

- Slide 33 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.

$$dist(a,n) \quad < \quad dist(a,p) \tag{7}$$

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 29:

$$dist(a,p) \quad < \quad dist(a,n) \quad < \quad dist(a,p) + \delta \tag{8}$$

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:

$$dist(a,p) + \delta \quad < \quad dist(a,n) \tag{9}$$

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:

$$\mathcal{L} \;=\; \sum_{i=1}^{N} \max\left\{ ||f(\mathbf{x}_i^a) - f(\mathbf{x}_i^p)||_2^2 - ||f(\mathbf{x}_i^a) - f(\mathbf{x}_i^n)||_2^2 + \delta, \;\; 0 \right\} \quad (10)$$

- Note that, for a given pair $(Anchor, Pos)$, the loss defined above will not accept any contributions from those triplets whose negatives are outside the margin shown in the figure on Slide 29. For such triplets, since the negative will be 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

Why is Mining a Difficult Problem?

- Writing network code for mapping an input image into an embedding vector of a specified size is relatively trivial. You can take any image classification network and replace its last layer (the one meant for predicting the classification label) with a Linear layer that produces the desired embedding vectors.

  [**DLStudio's** `MetricLearning` **gives you two choices,** `EmbeddingGenerator1` **and** `EmbeddingGenerator2`**, for mapping input images into embedding vectors. The former is based on the same network that I have used elsewhere in DLStudio for classifying the CFAR10 dataset images. And the latter uses a pretrained ResNet-50 as the backbone (meaning as a feature extractor) whose output is again fed into a Linear year that generates the embeddings.**]

- From a coding standpoint, the challenging part of metric learning is *mining* the batches for positive and negative pairs for the Pairwise Contrastive Loss and for triplets for the Triplet Loss.

- Your first reaction to writing the code for mining is likely to be: **Why is it such a difficult problem?**

  [**YOU ARE LIKELY TO CONTINUE: Since you have the embedding vectors and their associated class labels in a batch, why can't we write a couple of** `for`**-loops in which we form Positive Pairs based on the class labels being the same for a pair of embedding vectors, and employ similar loop-based logic for the Negative Pairs and for the Triplets.**]

Why is Mining a Difficult Problem? (Contd.)

- Regarding the small-font note in red at the bottom of the previous slide, the problem is that iterative processing using `for`-loops and GPU based processing are not 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.

- **So if you tried to solve the problem of batch mining for metric learning with a couple of `for`-loops, while your code will run, your per-batch time during training would be so long as to render your code worthless.**

- In addition to the fact that iterative processing with loops is an anathema to GPU-based processing of data, **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.**]

Why is Mining a Difficult Problem? (Contd.)

- As you will see in this section, with purely tensor-based methods for mining a batch, you may be able to achieve a thousand-fold speedup when calculating Pairwise Contrastive or Triplet Losses. (As you how much speedup you actually achieve in any particular case would depend on the batch size and other factors.)

- In the remainder of this section, I'll start with the simple `for`-loop based iterative implementations — just to get the idea across that fundamentally what you need to achieve is trivial. **And then I will switch over to purely tensor-based 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 with Iterative Logic

- 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

- Although highly unintuitive (but not so in hindsight), given a vector of B labels for the embedding vectors in a batch, you can figure out the Positive and Negative Pairs by testing for Boolean equality between the row and the column representations of the same label vector:

```
>>> 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)             ## Comparing the row and col representations
>>> labels_equal                                                    ##      of the same vector
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, one argument array "sweeps" over the non-conforming axis of the other array in order 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 explain the use of `None` at the different axis positions in the array access syntax on the previous slide, this ploy when using the slice operator endows a 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` 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" on the diagonal will
        [False, False, False, False, False, False],          ##   prevent entries like (n,n)
        [ True, False, False, False, False, False],          ##   showing up as Posive Pairs
        [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 $x \rightarrow 0$, cannot be used in our case because the ratio $\frac{B}{n_c}$ is likely to be far from satisfying the $x \rightarrow 0$ 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

- After you have enumerated the Positive and Negative Pairs in a batch, your next job is to calculate 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:

```
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 on the next slide. 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

$$dot\_products \quad = \quad X \; @ \; X.T \tag{11}$$

[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 (think row vectors), x and y, in the tensor X:

$$||\vec{\mathbf{x}} - \vec{\mathbf{y}}||^2 \quad = \quad (\vec{\mathbf{x}} - \vec{\mathbf{y}})(\vec{\mathbf{x}} - \vec{\mathbf{y}})^T \quad = \quad ||\vec{\mathbf{x}}||^2 - 2\vec{\mathbf{x}} \cdot \vec{\mathbf{y}}^T + ||\vec{\mathbf{y}}||^2 \tag{12}$$

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:

$$squared\_norms\_embedding\_vecs \quad = \quad torch.diagonal(dot\_products) \tag{13}$$

  For the 6-embedding vectors example on Slide 44, 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.

$$\begin{aligned} distance\_matrix \quad &= \quad squared\_norms\_embedding\_vecs.view(1,6) \\ &\quad - \; 2.0 * dot\_products \\ &\quad + \; squared\_norms\_embedding\_vecs.view(6,1) \end{aligned} \tag{14}$$

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 49 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 42, 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 49.  The last arg of value 2
>>>                                                                     ##    means that sum is to carried out along Axis 2.
>>>                                                                     ##  See Slide 42 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 40 and 41. 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 34 involves three embedding vectors: anchor, positive, and negative. On Slide 34, 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 50) 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 41.
- In what follows, I'll use the logic of Slide 41 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 the Pairwise Contrastive Loss or the 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 (NN), 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 NN returned by an ANN algo 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.

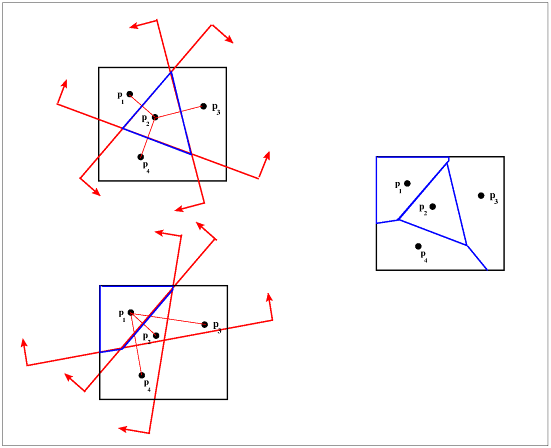
Vector Quantization (contd.)

- 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



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 that 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, for $m = 16$, we will consider each embedding vector to be composed of 16 smaller segments, each involving only 8 real numbers.

- The vertical bars between the integers are the separators (for visual convenience) between the adjacent position index values in a 128-element long embedding vector. As you know, a 128-element embedding vector consists of 128 real valued numbers, each a floating-point value between -1.0 and +1.0.

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.

  [**The main argument here is that we do not expect to see more than $2^4 = 16$ K-Means clusters to form in the space spanned by just 8 real-valued numbers.**]

- 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.

PQ for NN Retrieval (contd.)

- 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 below.
- 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 (presented in the previous section) 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 76. 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 to indexing than what I have described in the previous section.

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:

$$p_{j|i} \quad = \quad \frac{e^{-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{2\sigma_i^2}}}{\sum_{k \neq i} e^{-\frac{||\mathbf{x}_i - \mathbf{x}_k||^2}{2\sigma_i^2}}} \tag{15}$$

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:

$$H_i = -\sum_j p_{j|i} \cdot \log_2 p_{j|i} \tag{16}$$

- 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:

$$user\_supplied\_perplexity = 2^{H_i} \tag{17}$$

- 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 52.

- 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 84. Assuming that you are using numpy for the visualization code, as previously explained on Slide 51, 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 84 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:

$$q_{j|i} \quad = \quad \frac{e^{-||\mathbf{y}_i - \mathbf{y}_j||^2}}{\sum_{k \neq i} e^{-||\mathbf{y}_i - \mathbf{y}_k||^2}} \tag{18}$$

Mapping to Low-Dimensional Counterparts (contd.)

- There is one big difference between the conditional probs in Eq. (5) on Slide 84 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:

$$d_{KL}(P,Q) \quad = \quad \sum_{i=1}^{N} \mathbf{p}_i \log_2 \frac{\mathbf{p}_i}{\mathbf{q}_i} \tag{19}$$

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 $\mathbf{x}_i$:

$$d_{KL}(\{\mathbf{p}_{j|i}\}, \{\mathbf{q}_{j|i}\}) \quad = \quad \sum_{j=1}^{N} \mathbf{p}_{j|i} \log_2 \frac{\mathbf{p}_{j|i}}{\mathbf{q}_{j|i}} \tag{20}$$

  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:

$$\mathcal{C} \quad = \quad \sum_{i=1}^{N} \sum_{j=1}^{N} \mathbf{p}_{j|i} \log_2 \frac{\mathbf{p}_{j|i}}{\mathbf{q}_{j|i}} \tag{21}$$

- 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:

$$\begin{aligned}\mathcal{C} &= \sum_{i=1}^{N}\sum_{j=1}^{N} \mathbf{p}_{j|i}[\log_2 \mathbf{p}_{j|i} - \log_2 \mathbf{q}_{j|i}] \\ &= \sum_{i=1}^{N}\sum_{j=1}^{N} \mathbf{p}_{j|i}\log_2 \mathbf{p}_{j|i} - \sum_{i=1}^{N}\sum_{j=1}^{N} \mathbf{p}_{j|i}\log_2 \mathbf{q}_{j|i} \end{aligned} \tag{22}$$

- 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:

$$\frac{\partial \mathcal{C}}{\partial \mathbf{y}_i} = 2\sum_{j=1}^{N}\left(\mathbf{p}_{j|i} - \mathbf{q}_{j|i} + \mathbf{p}_{i|j} - \mathbf{q}_{i|j}\right)(y_i - y_j) \tag{23}$$

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.

$$p_{ij} = \frac{e^{-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{2\sigma^2}}}{\sum_{k \neq l} e^{-\frac{||\mathbf{x}_k - \mathbf{x}_l||^2}{2\sigma^2}}} \tag{24}$$

$$q_{ij} = \frac{e^{-||\mathbf{y}_i - \mathbf{y}_j||^2}}{\sum_{k \neq l} e^{-||\mathbf{y}_i - \mathbf{y}_l||^2}} \tag{25}$$

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:

$$p_{ij} = \frac{\mathbf{p}_{i|j} + \mathbf{p}_{i|j}}{N} \tag{26}$$

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

$$\frac{\partial \mathcal{C}}{\partial \mathbf{y}_i} = 4 \sum_{j=1}^{N} (\mathbf{p}_{ij} - \mathbf{q}_{ij})(\mathbf{y}_i - \mathbf{y}_j) \tag{27}$$

- 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:

$$f(t) \quad = \quad \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}\Gamma(\frac{\nu}{2})}\left(1+\frac{t^2}{\nu}\right)^{-(\nu+1)/2} \tag{28}$$

  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 $\nu \rightarrow \infty$. 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:

$$f(t) \quad = \quad \frac{1}{1 \ + \ t^2} \tag{29}$$

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

$$q_{ij} \quad = \quad \frac{\left(1 \ + \ ||\mathbf{y}_i - \mathbf{y}_j||^2\right)^{-1}}{\sum_{k \neq l}\left(1 \ + \ ||\mathbf{y}_k - \mathbf{y}_l||^2\right)^{-1}} \tag{30}$$

- 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:

$$\frac{\partial \mathcal{C}}{\partial \mathbf{y}_i} \quad = \quad 4\sum_{j=1}^{N} \frac{(\mathbf{p}_{ij} \ - \ \mathbf{q}_{ij})(\mathbf{y}_i \ - \ \mathbf{y}_j)}{1 \ + \ ||\mathbf{y}_i - \mathbf{y}_j||^2} \tag{31}$$

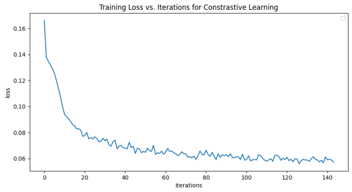
Outline

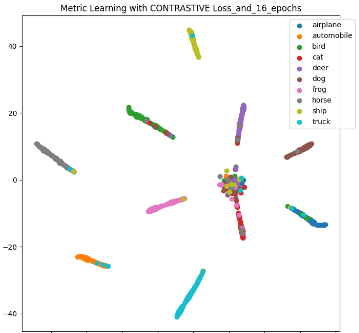
The MetricLearning Module in DLStudio

- 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 home-brewed network supplied with the MetricLearning module.

Results with Contrastive Learning on CIFAR-10

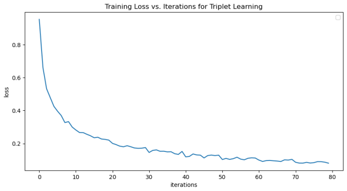
- Shown below are the training loss and the clustering achieved on a collection of images from the test dataset after the network was trained on CIFAR-10 dataset with Contrastive Loss over 16 epochs.
- With Pairwise Contrastive learning (and with no hyperparameter tuning of any sort), you get an accuracy 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.

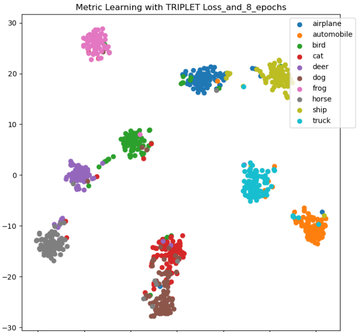




Results with Triplet Learning on CIFAR-10

- 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.





The MetricLearning Co-Class in DLStudio

-

Outline

What is Representation Learning?

- In old-style computer vision, you associated a set of features with the objects to be recognized and, at inference time, your main job was to extract the features from the images and to then declare the presence/absence of the objects in a scene based on the values of the features. **You could say that those features formed the representational vocabulary for the objects of interest.**
- And if, after training, the system made a decision that you thought was unacceptable, you could try to understand that decision based on what was happening in the data through the extracted features.
- Now compare that to how a neural network makes its decisions. There are no human-delineated features that the neural network has to be made aware of. The neurons in a network extract from the pixels whatever it takes to minimize the loss function being used. What's important here is that such a network can significantly outperform the older approaches in many cognitive tasks, such as classification, segmentation, tracking, etc.

What is Representation Learning? (contd.)

- While the superior performance of neural networks is wonderful, it comes with a price — the great difficulty for humans to figure out as to why a particular decision was made by a neural network. **This is referred to as the explainability problem of neural networks.**

- Another way of saying the same thing is that, at the moment, typical neural networks are unable to explain their decisions. [**This is unlike what is possible with, say, a Decision Tree based classifier. Through introspection abilities, it can tell you what elements of the training dataset directly influenced the output decision. See my Decision Trees tutorial.**]

- Any attempt at endowing neural networks with the ability to explain their own decisions **must begin with what's known as Representation Learning**. It is the ability of a neural network *to learn a vocabulary for the evidence that sits in its various layers as the data flows from the input layer to the final output where the decisions are made.*

What is Representation Learning? (contd.)

- Representation Learning may be supervised or unsupervised. **However, one thing is important: its goal should NOT be limited to learning what humans think are recognizable features in the input data.**

  [**When authors of research papers speak excitedly about their network having identified a neuron that was triggered by, say, the automobile wheels in the images of cars, that does NOT go to the heart of what Representation Learning is all about. Obviously, if a network can discriminate between, say, cars and airplanes, and if the wheels are prominent features in the images of the cars used for training, we can certainly expect that the process of gradient descent would result in convolutional operators extracting the wheels at some point in the network. But Representation Learning must go beyond that.**]

- Representation Learning means for a network to learn **on its own** the features in the data flowing through the network — regardless of their interpretability by humans — that eventually result in the network making the required discriminations. **In other words, Representation Learning would allow a network to create its own vocabulary of features that it could subsequently use to explain its decisions**.

- Representation Learning, in my mind, is best exemplified by how the pulmonary radiologists learn to distinguish between more than 200 different diseases on the basis of the HRCT (High Resolution Computed Tomography) images of the lung. See the next slide for elaboration.

What is Representation Learning? (contd.)

- With regard to the example of Representation Learning in the previous slide, the disease-specific visual patterns mentioned there are distinctive and yet non-descriptive which makes them much too challenging to be committed to one's memory.
- So when a radiologist is looking at the image of a new patient, they have to carry out a visual comparison between the patterns in the new patient's image and those in a published atlas of such images to assign a specific disease label to the new patient.
- A Representation Learning question here would be: Could a neural network be designed to learn automatically the visual patterns that you can find in a published atlas such as those at the following two links:

`http://maladies-pulmonaires-rares.fr/ckfinder/userfiles/files/documents-telecharger/par_maladie/IMAGING%20ATLAS%20OF%20INTERSTITIAL%20LUNG%20DISEASES.pdf`

`https://www.atsjournals.org/doi/full/10.1164/rccm.202303-0534OC`

Outline

NCE (Noise Contrastive Estimation) as a Stepping Stone to Representation Learning

- Noise Contrastive Estimation (NCE), first proposed by Gutmann and Hyvärinen in 2010, allows you to estimate a probability distribution more efficiently through *contrastive learning* that involves, on the one hand, the true samples drawn from the distribution in question and, on the other, artificially injected noise that is irrelevant to the distribution being learned.

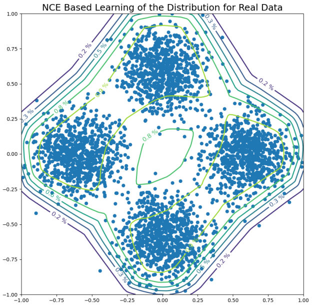
- The left-side of the visualization on Slide 114 shows the samples in a 2D plane drawn from a multi-Gaussian distribution. The equi-value contours that you also see on the left are an NCE estimate for the probability distribution in question. [**Yes, I know that I have not yet told the reader about how the NCE algorithm works. You'll see it shortly in these slides!**]

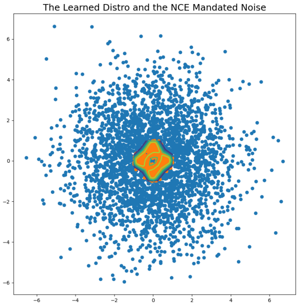
- The NCE algorithm that produced the probability density result (in the form of equi-value contours) shown at left on Slide 114 used only one sample of the artificially generated noise (I call it NCE-mandated noise).

NCE as a Stepping Stone (contd.)

- Now compare the result shown at left on Slide 114 with the result shown in Slide 115 where the NCE algorithms is asked to use 5 “negative” samples from the NCE-mandated noise vis-a-vis only one on the next slide.
- Obviously, using 5-samples of the NCE-mandated noise for each sample of the real-data distribution gives us a more accurate estimate of the real-data distribution.
- What is shown at right in Slides 114 and 115 are *all* the points used for estimating the distribution of the real data. These include the points drawn from the real-data distribution and the points drawn from the NCE-mandated noise. **Pay attention to the scales used for the plots on the left and on the right**. What you see on the left in both slides corresponds to the small rad-orange central part in the plots on the right. What the right-hand-side plots tell us is that the NCE-mandated points represent a “background” noise with a much wider variance than the real-data points in the xy-plane.

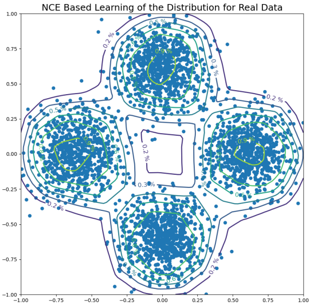
NCE as a Stepping Stone (contd.)

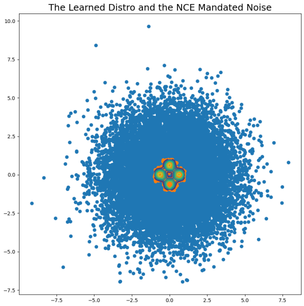




What you see on the left are the points from a "real data" multi-Gaussian distribution. The contours depict the probability distribution estimated by the NCE algorithm. The visualization at right shows **all** the points used by the algorithm, that is, both the real-data points and the NCE-mandated background noise. Pay attention to the horizontal scale in both visualizations. What you see in the small reddish area in the middle at right is all of the visualization at left.

NCE as a Stepping Stone (contd.)





What you see on the left are the points from a "real data" multi-Gaussian distribution. The contours depict the probability distribution estimated by the NCE algorithm. The visualization at right shows **all** the points used by the algorithm, that is, both the real-data points and the NCE-mandated background noise. Pay attention to the horizontal scale in both visualizations. What you see in the small reddish area in the middle at right is all of the visualization at left.

NCE as a Stepping Stone (contd.)

- That brings me to explaining the working of the NCE algorithm that produced the results on the previous two slides. There are two critical parts to the working of this algorithm: **(i)** The Data Generator that spits out the data points for the real data and, for each real data point, that spits out a designated number of NCE-mandated noise points in the xy-plane. And **(ii)** The network that learns to discriminate between the real-data points and the NCE-mandated noise.

- Slide 119 presents the code for the Data Generator. Note the two parameters defined for the function in Line (A): `batch_size` and `N_for_neg_samples_for_each_pos` .

- Regarding the second parameter named above: it tells the data generator how many negative samples — meaning samples from the NCE-mandated noise to generate for each so-called positive sample. [**The positive samples correspond to the multi-Gaussian defined by the four centers in Line (E) and the** `real_data_spread` **in Line (C). On the other hand, the negative samples from the NCE-mandated noise are generated by the value of** `noise_noise_spread` **in Line (D) and in Lines (Q) through (T).**]

NCE as a Stepping Stone (contd.)

- The starting point for generating both the positive and the negative samples is a call to the generic random number generator `np.random.randn(2)`, as shown in Line (K) for the positive samples and in Line (R) for the case of negative samples.

  [**Such a call returns a pair of floating-point values that can be construed as being drawn from a bivariate standard normal distribution, meaning a 2D Gaussian distribution with zero-mean and unit standard deviation.**]

- You'd find it interesting to note that for any given value for the parameter `batch_size` in Line (A), the actual size of the batch would also depend on the value for the parameter `N_for_neg_samples_for_each_pos` . Here is the reason:

  [**Let's say `batch_size` is 32 and the value of the variable `N_for_neg_samples_for_each_pos` is 1. In this case, for every positive sample, you have one negative sample. In this case, the batch returned will have alternating positive and negative sample. That is, your batch of 32 samples will consists of 16 positive samples 16 negative samples. However, when `N_for_neg_samples_for_each_pos` is 5, each positive sample will be followed by 5 negative samples. So, with batch-size set to 32, the batch will contain 16 positive samples, but 80 negative samples. So the actual batch-size in this case would be 96.**]

- Also note the role played by the variable `scale` in Line (B). You need to set a value for this variable to that most of the real-data distribution will lie inside a square that spans $(-1.0, 1.0)$ width-wise and $(-1.0, 1.0)$ height-wise.

NCE as a Stepping Stone (contd.)

- On the other hand, the spread of the NCE-mandated noise is determined exclusively the variable `noise_noise_spread` in Line (D). So you can see the reason for showing separately the plots for the real data and for the NCE-mandated noise in the visualizations on Slides 114 and 115.

- Finally, note that the function shown on the next slide is a Python Generator because it supplies its output through the `yield()` function in Line (V) . Theoretically at least, it is capable of returning an unbounded number of batches, each different from the others. As you know, from a user's standpoint, a Python generator object is an iterator for which you do not have to define the `__iter__()` and `__next__()` that would otherwise be required by the iterator protocol.

NCE as a Stepping Stone (contd.)

```
##  The DATA GENERATOR:
def multi_gaussian_source_with_nce_mandated_noise(batch_size, N_for_neg_samples_for_each_pos):                ## (A)
    scale = 0.6                                                                                               ## (B)
    real_data_spread = 0.15                                                                                   ## (C)
    nce_noise_spread = 2                                                                                      ## (D)
    centers = [                                                                                               ## (E)
        (1.0,0),
        (-1.0,0),
        (0, 1.0),
        (0, -1.0),
    ]
    centers = [(scale * x, scale * y) for x, y in centers]                                                    ## (F)
    while True:                                                                                               ## (G)
        databatch = []                                                                                        ## (H)
        labels  = []                                                                                          ## (I)
        for i in range(batch_size//2):                                                                        ## (J)
            #  the positive sample
            point = np.random.randn(2) * real_data_spread         ##  for the real-data sample                ## (K)
            center = random.choice(centers)                                                                   ## (L)
            point[0] += center[0]                                                                             ## (M)
            point[1] += center[1]                                                                             ## (N)
            databatch.append((point))                                                                         ## (O)
            labels.append(1)                                      ##  1 is the label for the real data       ## (P)
            ##  for the NCE-mandated negative samples:
            for _ in range(N_for_neg_samples_for_each_pos):                                                   ## (Q)
                point = np.random.randn(2) * nce_noise_spread                                                 ## (R)
                databatch.append((point))                                                                     ## (S)
                labels.append(-1)                         ##  -1 is the label for the NCE mandated noise      ## (T)
        databatch = np.array(databatch, dtype='float32')                                                      ## (U)
        yield torch.tensor(databatch, dtype=torch.float32),  torch.tensor(labels)                             ## (V)
```

NCE as a Stepping Stone (contd.)

- The presentation on the last half-dozen slides addressed the first of the two critical NCE-related things mentioned in the first bullet on Slide 110. That brings me to the second of those things: A Discriminator that should learn to discriminate between the real data and the NCE-mandated noise.

- Shown below is the Discriminator used for the results shown earlier. It is a simple binary classifier that takes a batch of input data, shoves it through a couple of `nn.Linear` layers, with the final layer producing a scalar value that is subject to `nn.Sigmoid` activation that, as you will see later, will be subject to a Binary Cross Entropy loss.

```
DIM = 32
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        main = nn.Sequential(
            nn.Linear(2, DIM),
            nn.ReLU(),
            nn.Linear(DIM, DIM),
            nn.ReLU(),
            nn.Linear(DIM, 1),
            nn.Sigmoid()
        )
        self.main = main

    def forward(self, x):
        x = self.main(x)
        return x
```

NCE as a Stepping Stone (contd.)

- Shown below is the entire code file that you can copy and paste for carrying out your own experiments in the NCE-based approach to the estimation of the probability models for point data.
- Note that the script as written takes one command line argument as shown below:

```
    python3  NCE_for_learning_point_distro.py  N          ## if you want N negative samples for each po
```

```
##  NCE_for_learning_point_distro.py
##  Avi Kak
##  Nov 17, 2025

import random
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import sys

## np.set_printoptions(precision=3)
## np.set_printoptions(threshold=sys.maxsize)
## torch.set_printoptions(edgeitems=10_000)

np.random.seed(123)

##  Set the value of how many NCE-mandated noise samples you want (these are the negative samples) for each positive sample:
if len(sys.argv) != 2:
    sys.stderr.write("Usage: %s  <how many negatives for each real>\n" % sys.argv[0])
    sys.exit(1)
```

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

NCE as a Stepping Stone (contd.)

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

```
N_for_neg_samples_for_each_pos = int(sys.argv[1])

##  The DATA GENERATOR:
def multi_gaussian_source_with_nce_mandated_noise(BATCH_SIZE):
    scale = 0.6
    real_data_spread = 0.15
    nce_noise_spread = 2
    centers = [
        (1.0,0),
        (-1.0,0),
        (0, 1.0),
        (0, -1.0),
#        (1.0/np.sqrt(2), 1.0/np.sqrt(2)),             ## For making the real-data distro more complex, uncomment
#        (1.0/np.sqrt(2), -1.0/np.sqrt(2)),            ## these four lines
#        (-1.0/np.sqrt(2), 1.0/np.sqrt(2)),
#        (-1.0/np.sqrt(2), -1.0/np.sqrt(2))
    ]
    centers = [(scale * x, scale * y) for x, y in centers]
    while True:
        databatch = []
        labels  = []
        for i in range(BATCH_SIZE//2):
            #  the positive sample
            point = np.random.randn(2) * real_data_spread         ##  for the real-data sample
            center = random.choice(centers)
            point[0] += center[0]
            point[1] += center[1]
            databatch.append((point))
            labels.append(1)                                      ##  1 is the label for the real data

            ##  for the NCE-mandated negative samples:
            #  The value for N_for_neg_samples_for_each_pos was supplied as a command-line argument to the scrip6
            for _ in range(N_for_neg_samples_for_each_pos):
                point = np.random.randn(2) * nce_noise_spread
                databatch.append((point))
                labels.append(-1)                                 ##  -1 is the label for the NCE mandated noise
        databatch = np.array(databatch, dtype='float32')
        yield torch.tensor(databatch, dtype=torch.float32),  torch.tensor(labels)
```

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

NCE as a Stepping Stone (contd.)

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

```
##  The MODEL:
DIM = 32
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        main = nn.Sequential(
            nn.Linear(2, DIM),
            nn.ReLU(),
            nn.Linear(DIM, DIM),
            nn.ReLU(),
#           nn.Linear(DIM, DIM),
#           nn.ReLU(),
            nn.Linear(DIM, 1),
            nn.Sigmoid()
        )
        self.main = main

    def forward(self, x):
        x = self.main(x)
        return x

def weights_init(m):
    """
    This function is used to initialize the learnable weights in the Discriminator network
    """
    classname = m.__class__.__name__
    if classname.find('Linear') != -1:
        m.weight.data.normal_(0.0, 0.02)
        m.bias.data.fill_(0)

## INITIALIZE THE MODEL:
BATCH_SIZE = 32
model = Discriminator()
weights_init(model)
loss_fn =  nn.BCELoss()
optimizer = optim.Adam(model.parameters())

## TRAINING:
iter_index = 0
```

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

NCE as a Stepping Stone (contd.)

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

```
running_loss = 0.0
for datapoints, labels in multi_gaussian_source_with_nce_mandated_noise(BATCH_SIZE):
    iter_index += 1
    output = model(datapoints)
    loss = 0.0
    for i,label in enumerate(labels):
        if label == -1: # noise
            loss += loss_fn(output[i], torch.tensor([0]).float())
        else: # real
            loss += loss_fn(output[i], torch.tensor([1]).float())
    running_loss += loss
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()  # Update model parameters
    if iter_index % 1000 == 999:
        print("\niteration: %d    loss: %f" % (iter_index+1, running_loss/1000))
        running_loss = 0.0
#    if iter_index == 2000: break           ## for debugging visualization code
    if iter_index == 10000: break


##  EVALUATION:
data_trove_eval = None                    ##  only the real data samples
labels_eval = []
all_data_used = None                      ##  includes both the real data and the nce mandated noise
all_labels_used = None
correct = 0
total = 0
with torch.no_grad():
    eval_iter = 0
    for datapoints, labels in multi_gaussian_source_with_nce_mandated_noise(BATCH_SIZE):
        if all_data_used is None:
            all_data_used = datapoints
            all_labels_used = labels.tolist()
        else:
            all_data_used =  np.concatenate( (all_data_used, datapoints), axis=0)
            all_labels_used.append(labels)
        output = model(datapoints)
        predicted = output > 0.5            # Predict label based on output score
        for i,label in enumerate(labels):
            total += 1
```

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

NCE as a Stepping Stone (contd.)

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

```
            if label == -1:
                if predicted[i] == False:
                    correct += 1
            else:
                if predicted[i] == True:
                    correct += 1
                pt_coords = datapoints[i].numpy()
                if  abs( pt_coords[0] ) < 1.0  and  abs( pt_coords[1] ) < 1.0:
                    if data_trove_eval is None:
                        data_trove_eval = np.expand_dims( pt_coords, axis=0 )
                    else:
                        pt_coords = np.expand_dims( pt_coords, axis=0 )
                        data_trove_eval =  np.concatenate( (data_trove_eval, pt_coords), axis=0)
        eval_iter += 1
        if eval_iter == 200: break
print(f"\n\n\nACCURACY: {correct / total * 100:.2f}%")

##  VISUALIZATION:

##  Visualization consists of two parts:  The first part shows you the learned distro for the real data.
##                                       And the second part also shows all of the data used, including the
##                                       NCE mandated noise.

#  This custom formatter from Matplotlib doc pages removes trailing zeros, e.g. "1.0" becomes "1", and
#  then adds a percent sign.
def fmt(x):
    s = f"{x:.1f}"
    if s.endswith("0"):
        s = f"{x:.0f}"
    return rf"{s} \%" if plt.rcParams["text.usetex"] else f"{s} %"

plt.figure(figsize=(10, 10))
#plt.scatter(data_trove_eval[:,0], data_trove_eval[:,1], c=labels_eval)
##  To understand the indexing below, note that data_trove_eval is a numpy array of shape (N,2) where N is, say, 1000.  The first
##  element in the args below returns the x-coord of all the (x,y) points and the second element returns all the y-coords:
plt.scatter( data_trove_eval[:,0], data_trove_eval[:,1] )
plt.title("NCE Based Learning of the Distribution for Real Data", fontsize=18)
x_vals = np.linspace(-1.0, 1.0, 100)
y_vals = np.linspace(-1.0, 1.0, 100)
xv, yv = np.meshgrid(x_vals, y_vals)
```

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

NCE as a Stepping Stone (contd.)

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

```
## Calling "xv.ravel()" yields a list of all the x-coordinates for the 100x100  array of points
##   as defined by x_vals and y_vals values above. Note that we have a total of 10,000 points.
##   And, "xv.ravel()" is a list of 10,000 x-coordinates in a row-major scan of all the points
##   in the xy-plane.  Similarly, "yv.ravel()" is a list of the corresponding 10,000 y-coordinates.
grid_input = torch.tensor(np.stack([xv.ravel(), yv.ravel()], axis=1), dtype=torch.float32)
##                                   \_________/  \________/
##                                10000 x_coords  10000 y_coords
##   np.stack pairs up the individual x-coords and y-coords in the two lists returned by  xv.ravel()
##   and yv.ravel().  What is returned by  np.stack()  is just a list of 10,000 (x,y)-coordinates in
##   (x,y)-plane
grid_output = model(grid_input).view(xv.shape).detach().numpy()
min_data, max_data = np.amin(grid_output), np.amax(grid_output)
print("\n\nmin val in grid output: %f     max val in grid output: %f" % (min_data, max_data) )
#cst = plt.contour(xv, yv, grid_output.reshape((len(xv), len(yv))), linewidths=[2], colors=['red'])
cst = plt.contour(xv, yv, grid_output.reshape((len(xv), len(yv))), linewidths=[2])
plt.clabel(cst, inline=1, fmt=fmt, fontsize=12)
plt.savefig("learned_distro_for_real_data_" + str(N_for_neg_samples_for_each_pos) + ".png")
plt.show()

##  Visualization including the nce mandated noise:
plt.figure(figsize=(10, 10))
plt.scatter(all_data_used[:,0], all_data_used[:,1])
plt.scatter( data_trove_eval[:,0], data_trove_eval[:,1] )
plt.title("The Learned Distro and the NCE Mandated Noise", fontsize=18)
x_vals = np.linspace(-1.0, 1.0, 100)
y_vals = np.linspace(-1.0, 1.0, 100)
xv, yv = np.meshgrid(x_vals, y_vals)
grid_input = torch.tensor(np.stack([xv.ravel(), yv.ravel()], axis=1), dtype=torch.float32)
grid_output = model(grid_input).view(xv.shape).detach().numpy()
min_data, max_data = np.amin(grid_output), np.amax(grid_output)
print("\n\nmin val in grid output: %f     max val in grid output: %f" % (min_data, max_data) )
plt.contour(xv, yv, grid_output.reshape((len(xv), len(yv))), linewidths=[2])
plt.savefig("learned_distro_and_nce_noise" + str(N_for_neg_samples_for_each_pos) +  ".png")
plt.show()
```

Outline

Mutual Information and the InfoNCE Loss

- Perhaps the best application of the NCE idea is in the InfoNCE loss function (and, as I'll mention later, in the PatchNCE loss function that followed InfoNCE) that was used by van den Oord et el. in their 2019 publication "*Representation Learning with Contrastive Predictive Coding*" to show that a system could learn **on its own the semantically significant "concepts" in a data stream**.

- To understand InfoNCE, let's say that the data at our disposal can be explained through a set of concepts $\mathcal{C} = \{c_1, c_2, \cdots, c_K\}$. If we use the notation $\mathcal{X} = \{x_1, x_2, \cdots\}$ to represent the data instances, which, for your imagination, could be the individual images in your dataset, the question then becomes what's best way to learn the concepts as we examine the data instances (in no particular order).

- According to the principles enunciated by van den Oord et el., our goal should be to maximize the **Mutual Information (MI)** between the concepts and the individual data observations. That begs the question: **What is Mutual Information?**

MI and InfoNCE Loss (contd.)

- Given two random variables x and y, the MI between them is given by

$$MI(X;Y) = \sum_{x,y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)} = \sum_{x,y} p(x,y) \log \frac{p(x|y)}{p(x)} \qquad (32)$$

where $p(x,y)$ is the joint probability distribution for the two random variables. To understand the formula for MI intuitively (through, say, a Venn diagram), the greater the "overlap" between the marginals for x and y, the larger the MI between the two variables.

[**At this point, you may wish to quickly review the entropy related material in Slides 14 through 25 of my Week 7 lecture. The RHS shown above bears some resemblance to the definition of joint entropy on Slide 25 of that lecture, but, in terms of what they stand for, the two formulas are as different as different can be. In the middle formula in the equation above, the appearance of the product of two marginals in the denominator of the argument to the logarithm completely alters the character of the formula. While the formula for joint entropy on Slide 25 of my Week 7 lecture measures the uncertainty associated with the joint distribution** $p(x,y)$ **(the more uniform the distro, the larger the uncertainty, etc.), the formula shown above measures the "relatedness" of the two variables** X **and** Y**. Nevertheless, it is trivial to show by writing** $\log \frac{A}{B} = \log A - \log B$ **that the formula for MI can be expressed as** $MI(X,Y) = H(X,Y) - H(X|Y) - H(Y|X)$.]

- Using the notation $\mathcal{C}$ for the contexts and $\mathcal{X}$ for the observations, the formula shown above for the relatedness between the two can be expressed as:

$$MI(\mathcal{X};\mathcal{C}) = \sum_{x,c} p(x,c) \cdot \log \frac{p(x|c)}{p(x)} \qquad (33)$$

MI and InfoNCE Loss (contd.)

- From now on, I'll use $\mathcal{X}$ to denote the N data samples in a batch.
- The formula shown at the bottom of the previous slide tells us how to sum up the individual MI-related contributions from given (x_i, c_k) pairs. **What it says is that if the data sample in a batch is x_i and you are examining its relevance to the concept c_k, that relevance is given by the ratio $\frac{p(x_i)|c_k}{p(x_i)}$.** Subsequently, you carry out a weighted sum of the logs of these ratios for an overall measure of the relatedness between all the batch instances and all available concepts.
- So with regard to the MI based relevance of an individual batch sample x_i to a given concept c_k, let's define:

$$MI_{rel}(x_i; c_k) \quad \propto \quad \frac{p(x|c)}{p(x)} \tag{34}$$

MI_{rel} is just the ratio of the probability with which we are likely to see x_i given that we know that it was generated by the phenomenon described by the concept c_k and the probability it is likely to be encountered at random (without consideration of what generated it).

MI and InfoNCE Loss (contd.)

- Now that we have a criterion that can tell us how related a given sample x_i in a batch is to a concept c_k, the next question that needs to be addressed is how to compare the relatedness of all available concepts to all the data samples in a batch.

- **That is, we want to optimally map all the data samples in a batch to all the available concepts.**

- I'll use $p(d = i \mid \mathcal{X}, c_t)$ to denote the probability that, of all the N samples in batch $\mathcal{X}$, the sample x_i is the positive sample with regard to the concept c_t, and that **all other samples are negative with respect to the same concept**. Positive sample means that its MI with the concept in question is high. So we can write:

$$p(d = i \mid \mathcal{X}, c_t) \quad = \quad Z \cdot p(x_i|c_t) \cdot \prod_{l \neq i} p(x_l) \tag{35}$$

  where Z is the normalizer so that the numbers returned by the RHS can be treated as valid probabilities.

MI and InfoNCE Loss (contd.)

- For obvious reasons, the normalization mentioned on the previous slide would need to be with respect to all possible values for the index i on the RHS for each value of the same index on the LHS.
- To make the normalization explicit, we need to divide the RHS above by the sum of those terms over all possible i:

$$\begin{aligned}
p(d=i|\mathcal{X},c_t) &= \frac{p(x_i|c_t)\cdot\prod_{k\neq i}p(x_k)}{\sum_j p(x_j|c_t)\cdot\prod_{k\neq j}p(x_k)} \\
&= \frac{\frac{p(x_i|c_t)}{p(x_i)}\cdot p(x_i)\cdot\prod_{k\neq i}p(x_k)}{\sum_j \frac{p(x_j|c_t)}{p(x_j)}\cdot p(x_j)\cdot\prod_{k\neq j}p(x_k)} \\
&= \frac{\frac{p(x_i|c_t)}{p(x_i)}\cdot\prod_k p(x_k)}{\sum_j \frac{p(x_j|c_t)}{p(x_j)}\cdot\prod_k p(x_k)} \\
&= \frac{\frac{p(x_i|c_t)}{p(x_i)}}{\sum_j \frac{p(x_j|c_t)}{p(x_j)}}
\end{aligned} \tag{36}$$

MI and InfoNCE Loss (contd.)

- Using the MI_{rel} formula shown on Slide 123 that tells how to measure the relatedness of a given data sample x_i to a given concept c_k, we can now write:

$$p(d = i \mid \mathcal{X}, c_t) \quad = \quad \frac{MI_{rel}(x_i; c_t)}{\sum_j MI_{rel}(x_j; c_t)} \tag{37}$$

- The InfoNCE loss can now be defined as:

$$\mathcal{L}_{InfoNCE} = -\,\mathbb{E}_{\mathcal{X}} \log \frac{MI_{rel}(x_i; c_t)}{\sum_j MI_{rel}(x_j; c_t)} \tag{38}$$

  where $\mathbb{E}_{\mathcal{X}}$ is the expectation operator over all the batch samples.

- One last issue that remains to be resolved is: How would one measure $MI_{rel}(x_i; c_t)$ in code? The authors of InfoNCE loss recommend the following to that end:

$$MI_{rel}(x_i; c_t) \quad = \quad \exp\{z_i^T \cdot (W_i \cdot c_t)\} \tag{39}$$

  where z_i is the embedding-vector representation for the batch sample x_i and W_i a learnable projection matrix.

MI and InfoNCE Loss (contd.)

- I would not be surprised if, at this point in the slides, a reader is wondering as to why InfoNCE is considered to be an application of the basic idea of NCE.
- Note that the core idea in NCE is to learn by discriminating between the real data samples and the negative samples that are irrelevant to the real data samples.
- Let's say you are using InfoNCE for unsupervised clustering of some real data. Let's say the data has K classes and that you can associate a prototype vector (initially unknown) with each class. If we so choose, we can refer to each prototype vector as the concept c_i for the i^{th} class.
- During training, you take the inner product of all the prototype vectors c_i, $i = 1, \cdots, K$, with each batch instance x_i and you take note of the prototype vector, let's say it is c_t, that yields the largest inner product value.

MI and InfoNCE Loss (contd.)

- Continuing with the last bullet on the previous slide, for calculating the InfoNCE loss, you consider x_i in question as the positive sample vis-a-vis the concept c_t and you consider all other x_j's as negative samples. Thus, the InfoNCE loss calculated in this manner has the same "semantics" as the NCE loss you saw earlier.

- Finally, it is common to express the formula of Eq. (38) as:

$$\mathcal{L}_{InfoNCE} \quad = \quad -\mathbb{E}_{\mathcal{X}} \log \frac{\exp \frac{v \cdot v^{+}}{\tau}}{\exp \frac{v \cdot v^{+}}{\tau} + \sum_{n=1}^{N} \exp \frac{v \cdot v_n^{-}}{\tau}} \tag{40}$$

  where, in keeping with the explanation so far, v is the embedding for a batch sample and v^{+} the embedding for the corresponding prototype as made available by the ground-truth. On the other hand, $\{v_1^{-}, v_2^{-}, \ldots, v_N^{-}\}$ are the embeddings for the prototypes not relevant to the batch sample in question. The symbol τ is meant for scaling the dot products and it is commonly referred to as the *temperature*. Typically, it is a small number like 0.05.

MI and InfoNCE Loss (contd.)

- Note that learning with InfoNCE belongs to a class of techniques known as Learning with Negative Sampling that now has a very long history that goes all the way back to traditional machine learning. Around three decades back, it was used for designing powerful face recognition algorithms using the AdaBoost framework.

Extending InfoNCE Loss to PatchNCE Loss

- While InfoNCE chooses positive and negative samples on a full-image basis in a batch, PatchNCE creates the same through patches from each image separately. The idea of PatchNCE was first proposed by Park et el. in their 2020 paper "*Contrastive Learning for Unpaired Image-to-Image Translation*":

  https://arxiv.org/pdf/2007.15651

- Image-to-image translation means that you want to alter some of the visual attributes of an image while its core content remains unchanged. Example applications: Generating photorealistic images from object silhouettes, generating daytime photos of the outdoors from their nighttime versions or vice versa, turning a zebra into a horse or vice-versa, etc.

- Earliest work in image translation started with datasets of paired images, as exemplified by the 2017 paper "*Image-to-image translation with conditional adversarial networks*" by Isola et al.
  [**The source and target images in paired training datasets are in pixel registration and it is often sufficient to use the Adversarial Loss (See my Week 11 lecture) for creating the desired textures in the target images whereas the Reconstruction Loss (measured by either L1 or L2 metric) preserves the content (the shapes of the objects).**]

Extending InfoNCE to PatchNCE (contd.)

- A great deal of work now in image-to-image translation is with unpaired image datasets. With unpaired images for training, the goal is to learn the content distribution from the source images and the texture distributions from the target images. In order to make the learning spatially consistent, some of the earliest published papers in unpaired image-to-image translation used Cycle Consistency Loss, which seeks to establish a correspondence between the source and the target images by learning an inverse mapping from the target to the source. Cycle Consistency was first introduced by Zhu et el. in their 2020 paper "*Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*".

- An easier to implement and easier to train alternative to Cycle Consistency is Contrastive Learning based on InfoNCE, but based solely on the mining the negative patches from the same image as the positive patch.

Extending InfoNCE to PatchNCE (contd.)

- As formulated by Zhu et el. in their 2020 publication, a novel feature of extending InfoNCE to PatchNCE is that the positive and negative samples are pulled from the different layers of the image Encoder. As you saw in my previous lecture dealing with semantic segmentation networks, training images are typically processed by Encoder-Decoder architectures in which the different layers in the Encoder produce spatially more compact but channel-wise richer representation of an input image. Subsequently, the Decoder progressively enlarges the output of the Encoder while infusing it with the desired texture, color, and other attributes.

- PatchNCE applies InfoNCE to the pixels pulled randomly from the different layers of the Encoder. Since each pixel in a layer of the Encoder represents a patch of the input image, comparing the corresponding pixels in the Encoder and the Decoder is tantamount to comparing patches in the input and the generated output.

Extending InfoNCE to PatchNCE (contd.)

- In the formula shown below, L represents the layers of the Encoder chosen for sampling the pixels (that represent patches in the input image) for calculating the PatchNCE loss, and S_l represents the set of pixel coordinates to be used in layer $l \in L$. The notation $\hat{z}_l$ stands for the embedding associated with a pixel in the Decoder layer $l \in L$, the notation z_l for the embedding for the corresponding pixel in the Encoder, and $z_l^{S \setminus s}$ for all the embeddings associated with all other pixels chosen in the same layer of the Encoder. Note that each pixel, in the Encoder or the Decoder, has the channel dimension associated with it.

$$\mathcal{L}_{PatchNCE} = \mathbb{E}_{\mathcal{X}} \sum_{l=1}^{L} \sum_{s=1}^{S_l} \ell(\hat{z}_l^s, z_l^s, z_l^{S \setminus s}) \tag{41}$$

where, in keeping with the InfoNCE formula in Eq. (40), ℓ is given by

$$\ell(v, v^+, \{v^-\}) = -\log \frac{\exp \frac{v \cdot v^+}{\tau}}{\exp \frac{v \cdot v^+}{\tau} + \sum_{n=1}^{N} \exp \frac{v \cdot v_n^-}{\tau}} \tag{42}$$

Implementing InfoNCE and PatchNCE with `nn.CrossEntropyLoss`

- If you look at the RHS in Eq. (42) and the formula implemented by `nn.CrossEntropyLoss` as presented in Slide 24 of my Week 7 lecture, they are essentially the same. Does that mean that you can directly call `nn.CrossEntropyLoss` to do the job? **Obviously not — because that loss function requires you to supply an integer index for its second argument.** When used for calculating the loss in a classification network, the integer index represents the class label of the input image in the ground-truth.

- The next slide presents the pseudocode reproduced from the Park et al. paper that shows how you can arrange the positive and negative samples in a batch so that you can call `nn.CrossEntropyLoss` for calculating the PatchNCE loss. This should also work for calculating the InfoNCE loss.

nn.CrossEntropyLoss for InfoNCE and PatchNCE (contd.)

```
##  Reproduced from Park et al. "Contrastive Learning for Unpaired Image-to-Image Translation", ECCV 2020

import torch
cross_entropy_loss = torch.nn.CrossEntropyLoss()

# Input: f_q (BxCxS) and sampled features from H(G_enc(x))
# Input: f_k (BxCxS) are sampled features from H(G_enc(G(x))
# Input: tau is the temperature used in NCE loss.
# Output: PatchNCE loss

def PatchNCELoss(f_q, f_k, tau=0.07):                                                              ## (A)
    # batch size, channel size, and number of sample locations
    B, C, S = f_q.shape                                                                            ## (B)
    # calculate v * v+: BxSx1
    l_pos = (f_k * f_q).sum(dim=1)[:, :, None]                                                     ## (C)
    # calculate v * v-: BxSxS
    l_neg = torch.bmm(f_q.transpose(1, 2), f_k)                                                    ## (D)

    # The diagonal entries are not negatives. Remove them.
    identity_matrix = torch.eye(S)[None, :, :]                                                     ## (E)
    l_neg.masked_fill_(identity_matrix, -float('inf'))                                             ## (F)

    # calculate logits: (B)x(S)x(S+1)
    logits = torch.cat((l_pos, l_neg), dim=2) / tau                                                ## (G)
    # return NCE loss
    predictions = logits.flatten(0, 1)                                                             ## (H)
    targets = torch.zeros(B * S, dtype=torch.long)                                                 ## (I)

    return cross_entropy_loss(predictions, targets)                                                ## (J)
```

Outline

CLIP — Metric Learning in a Multi-Modal Embedding Space

- A great deal of the current focus in deep learning is on "multi-modal learning". It owes its origins to OpenAI's CLIP model that was trained with a dataset of 400 million images along with the associated text strings scraped from the web. The goal was to be able predict images from text and vice versa. This work is described in the 2020 publication "*Learning Transferable Visual Models From Natural Language Supervision*" by Radford et al.

- The design of CLIP was based on the insights gained from OpenAI's work in LLMs (Large Language Modeling as covered in my Week 15 lecture): that, on their own, natural languages can be a rich source of supervision for training models that reflect language understanding. [**For example, for LLMs, the text itself can provide supervision if the goal of the model is to predict the next token, the next word, the next sentence, or, for that matter, even the next para.**]

- The same thing can be done with the data that consists of the images scraped from the web along with the text associated with the images (when such text is available — which, as it turns out, is there in great abundance).

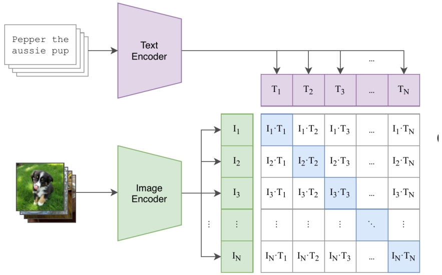
Metric Learning in a Multi-Modal Embedding Space (contd.)

- To continue with the thought in the last bullet on the previous slide, what is particularly interesting is that, in this case, the nature of supervision enabled by the text allows for multiple variants of the same text to be associated with the same image. That is, the different words — such as cat, kitty, feline, etc. — could all predict the image of a cat. This is unlike what you get with the more traditional neural classification networks that require precise names for the categories.

- However, learning the different possible text associations with the same image (and, also learning the different possible images associated with the same text) extracts a price: training with humongously large $(image, text)$ datasets. CLIP was trained originally with a dataset of 400 million $(image, text)$ pairs. Subsequent CLIP-like contributions have boasted of using datasets involving over a billion $(image, text)$ pairs.

Metric Learning in a Multi-Modal Embedding Space (contd.)

- CLIP learns to map both the images and the text captions to embedding vectors in their own spaces and then to project the embedding vectors into a common metric space. The goal of learning is to minimize the distance between the image embedding vectors and the embedding vectors for the captions associated with the images. For obvious reasons, you need a metric for the distance between the vectors. For the metric, CLIP uses the Cosine Similarity Distance between pairs of vectors.


Pepper the aussie pup
Text Encoder
Image Encoder
T_1 T_2 T_3 ... T_N
I_1 I_2 I_3 ⋮ I_N
$I_1 \cdot T_1$ $I_1 \cdot T_2$ $I_1 \cdot T_3$ ... $I_1 \cdot T_N$
$I_2 \cdot T_1$ $I_2 \cdot T_2$ $I_2 \cdot T_3$ ... $I_2 \cdot T_N$
$I_3 \cdot T_1$ $I_3 \cdot T_2$ $I_3 \cdot T_3$ ... $I_3 \cdot T_N$
$I_N \cdot T_1$ $I_N \cdot T_2$ $I_N \cdot T_3$ ... $I_N \cdot T_N$

We want to minimize the sum of the distances on the diagonal and maximize the sum of all off-diagonal distances.

Calculation of Loss in CLIP

- The "matrix" display shown in the figure on the previous slide is of size $N \times N$ if N is the batch size. The rows of the matrix represent the image embedding vectors and the columns the corresponding text embedding vectors.

- Our goal is to minimize the N Cosine distances between the image and the text vectors along the diagonal of the display and maximize the sum of the $N^2 - N$ distances in all the other cells of the display.

- The minimization along the diagonal and the maximization of the off-diagonal inner products can be cast in the form of InfoNCE loss in Section 12 of this lecture and, in your code, calculated with the nn.CrossEntropyLoss as explained previously on Slides 141 and 142.

- In the pseudocode for the loss shown on Slide 149, note how the images and the text are treated symmetrically. The image model used gives us the feature vector I_f for the images in a batch and the text model used gives us T_f for the text strings associated with the images.

Calculating the Loss in CLIP (contd.)

- Subsequently, the learned projection matrices, W_i for images and W_t for text, map the feature vectors into embedding vectors of the same dimensionality. The projections are denoted I_e for images and T_e for text on the next slide.

- With respect to each row of the dot-product matrix in Slide 146, the element on the diagonal plays the role of the positive sample v^+ in the InfoNCE formula shown in Eq. (40) and all the other elements correspond to the negative samples $\{v_1^-, v_2^-, \ldots\}$.

- As mentioned on the previous slide, the minimization with respect to the positive samples while maximizing the losses with respect to the negative samples can be carried out by using nn.CrossEntropyLoss.

- Note that supplying the second argument to nn.CrossEntropyLoss as the integer label for the "true class" is easier here as you can see on the next slide. It is easier compared to what was the case for PatchNCE on Slide 142.

Calculating the Loss in CLIP (contd.)

```
##  Pseudocode reproduced from the original CLIP paper by Radford et el.
##    "Learning Transferable Visual Models From Natural Language Supervision"

#  image_encoder  --  ResNet or Vision Transformer                                     ## (A)
#  text_encoder   --  CBOW or Text Transformer                                         ## (B)
#  I[B, H, W, C]  --  Batch of aligned images                                          ## (C)
#  T[B,L]         --  Batch of aligned texts                                           ## (D)
#  W_i[d_e,d_e]   --  Learned proj of image to embeddings                              ## (E)
#  W_t[d_t,d_e]   --  Learned proj of text to embeddings                               ## (F)
#  t              --  Learned temperature parameter                                    ## (G)

#  Extract feature representations of each modality
I_f   =  image_encoder(I)                            # [B, d_i]                        ## (H)
T_f   =  text_encoder(T)                             # [B,d_t]                         ## (I)

#  Joint multimodal embedding in [B, d_e]
I_e   =  l2_normalize( np.dot( I_f, W_i ), axis=1 )                                    ## (J)
T_e   =  l2_normalize( np.dot( T_f, W_t ), axis=1 )                                    ## (K)

#  Scaled pairwise Coside similarities  [B,B]
logits  =  np.dot( I_e, T_e.T ) * np.exp( t )                                          ## (L)

#  Symmetric loss function
labels  =  np.arange(B)                                                                ## (M)
loss_i  =  cross_entropy_loss( logits, labels, axis=0 )                                ## (N)
loss_t  =  cross_entropy_loss( logits, labels, axis=1 )                                ## (O)

loss    =  ( loss_i  +  loss_t ) / 2                                                   ## (P)
```

Outline

Experimenting with CLIP

- In order to experiment with CLIP, bear in mind the following: OpenAI has NOT made available either the source code for CLIP or the dataset of 400 million images+text pairs they used for training the model.
- Fortunately, a non-profit organization from Germany, LAION, has provided comparable image+text datasets and very successfully replicated the CLIP model (which is known as OpenCLIP).

[**LAION is a acronym for "Large-scale Artificial Intelligence Open Network" and their main purpose appears to be to provide large-scale datasets of images and associated captions extracted from the data scraped from the internet by Common Crawl. By captions is meant the text that website developers frequently associate with images with the HTML tag "ALT". Such text strings are meant to convey textually what is depicted by the image — which can be useful to a viewer should their web browser not be able to display the image itself for whatever reason. Here is the main website for LAION:**

`https://laion.ai/`

In order to save yourself the frustration of looking for an image+text dataset that you think might be hosted by LAION and not finding it there, visit the Wikipedia page on LAION. To the best of what I know, you are not going to be able to find the 400M (for 400 million image+text pairs) dataset that first made LAION well-known since its size matched OpenAI's 400M dataset that was used to train the original CLIP model. The reasons for the disappearance of datasets are well described at the Wikipedia page.]

Experimenting with CLIP (contd.)

- As you are thinking of playing with the CLIP model in your own lab, also think about the logistics and compute needs of working with humongously large datasets. Think about what it would take to download the dataset that was used for training the original CLIP model — 400 million image+text pairs. And what about the largest 5B dataset currently made available by LAION which is of size 5 billion image+text pairs?

- Even if the network bandwidth and the cost of storage were not your issues, think about the ownership issues related to the images downloaded by scraping the web. Since most image content in the web is now copyrighted, might there arise legal issues in the future related to your use of those images? In addition, you also have to worry about the possibility that an unbounded sweep of the internet for collecting images makes it highly likely that some of that data would be from websites that are seedy and violent.

Experimenting with CLIP (contd.)

- For some of the reasons mentioned on the previous slide, what LAION makes available are not the actual datasets of image+text pairs, but just the URLs to those. So if you are creating your own CLIP-like model, you will need to create a streaming data downloader that, at one end of the pipeline, downloads images from the URLs in the LAION dataset, and, on the other, feeds the image+text pairs into the model being trained.
- But, obviously, training your own CLIP like model with a humongous dataset is also going to require a heavy-duty compute infrastructure with a large number of high-end GPUs and with a distributed training protocol based on, say, the Python Lightning module.
- Given the challenges involved in training your own CLIP like model, your experiments with CLIP are more likely to consist of downloading from LION a dataset of image embeddings and associated text strings and then writing your own code for downstream tasks such as image retrieval from the dataset.

Experimenting with CLIP (contd.)

- Toward the goal mentioned in the last bullet on the previous slide, execute the following two `wget` commands in a directory meant for your CLIP experiments:

```
wget  https://deploy.laion.ai/8f83b608504d46bb81708ec86e912220/embeddings/img_emb/img_emb_0.npy

wget https://deploy.laion.ai/8f83b608504d46bb81708ec86e912220/embeddings/metadata/metadata_0.parquet
```

- These commands will deposit in your directory the following two archives:

```
img_emb_0.npy                       [Size: 1024458880  (over 1 GB)]

metadata_0.parquet                  [Size: 154933853  (over 154 MB)]
```

- The first of these is an archive of the embeddings for over one million images and the second a Parquet archive of the metadata related to the embeddings. If you are not familiar with the Parquet format, it is like a CSV file, with one row for each embedding in the embeddings archive. The next slide lists the different fields in each row.

Experimenting with CLIP (contd.)

- What's in each row of the Parquet archive corresponds to the following column headings:

```
Column Headings:  'image_path', 'caption', 'NSFW', 'similarity', 'LICENSE', 'url', 'key', 'shard_id', \
           'status', 'error_message', 'width', 'height', 'exif', 'original_width', 'original_height'
```

  with

```
    image_path:  A numeric ID for each image embedding vector in the img_emb_0.npy dataset

    caption:     The caption associated with the image

    NSFW:        The acronym stands for "Not Suitable for Work" for identifying adult content
    ...

    url:         The URL to the actual image
    ...
```

- After downloading the two archives mentioned on the previous slide — the embeddings archive and the associated metadata archive — it is time for you to create an **Inverted Index** for the over one million embedding vectors in your embeddings archive. You will create the Inverted Index with the `autofaiss` utility that will name the index file `knn.index`.

Experimenting with CLIP (contd.)

- As to why you need the Inverted Index `knn.index` for your archive of embeddings, please visit Section 6 of this lecture. Here is a quick summary of that material:

  [**An Inverted Index (often abbreviated as IVF for "Inverted File Index") solves the problem that, in practice, when you are finding the N nearest neighbors of a query vector, it is NOT practical to directly compare the query vector with every data vector in your embedding vector space. Therefore, in keeping with the discussion in Section 6, you partition the vector space into Vornoi cells and have each cell centroid point to the embedding vectors that reside in that cell. Subsequently, for a given query vector, you just find the nearest cell centroid and you compare the query vector with the dataset vectors in just that cell. A set of centroids for the Vornoi cells and having each such centroid to the dataset vectors that reside in that cell is what is provided by the file** `knn.index`]

- The work mentioned above is carried out by the `autofaiss` utility from the Python module of the same name. However, before you invoke this utility, it's best if you create a subdirectory named *embeddings* and move the two archive files `img_emb_0.npy` and `metadata_0.parquet` into that subdirectory.

  [**Autofaiss is a Python library designed specifically for creating the knn index. It sits on top of the FAISS (the Facebook AI Similarity Search) library that carries out efficient search and clustering of of dense vectors. Autofaiss is a wrapper around FAISS for the purpose of simplifying the process of building a high-performing index.**]

Experimenting with CLIP (contd.)

- Now for creating the Inverted Index file `knn.index`, execute the command

```
autofaiss  build_index --embeddings="embeddings"  --index_path="knn.index"  --index_infos_path="infos.json"   \
                                          --metric_type="ip"  --max_index_query_time_ms=5  --max_index_memory_usage="1GB"
```

- About the arguments to `autofaiss`:

```
--embedding               :  names the DIRECTORY containing the embedding files in npy format.
--index_path              :  names the destination folder for the index that is constructed
--index_infos_path        :  names the JSON with high-level info regarding the index
--metric_type             :  mames the similarity metric to use; the choices are "ip" for inner product and
                                                                               "l2" for Euclidean distance
--max_index_query_time_ms :  an approximate speed constraint for queries on the index that is constructed
--max_index_memory_usage  :  maximum sie in GB for the created index  [strictly followed]
```

- After you have run the `autofaiss` command, you will find the following items deposited in the directory in which you executed that command:

```
knn.index                              [size: 321397040  (321 MB)]

infos.json
```

  The first is what you are after and the second is useful information related to the first.

Experimenting with CLIP (contd.)

- Now that you have created a `knn.index` for your dataset of embedding vectors, you can start to write your own scripts for retrieving the embedding vectors that are closest to the embeddinv vector for for your query image or text.
- Next I'll show some example scripts for interacting with the downloaded CLIP embeddings archive.
- The script shown next, `image_to_image_retrieval_with_clip.py` chooses an integer at random and uses it as an index to retrieve a CLIP image to serve as a query image. To be more precise, it uses that integer index to retrieve the image embedding vector and the image URL that is at that index in the downloaded CLIP embeddings archive.
- It subsequently retrieves from the `knn.index` the 5 closest embedding vectors in the database. Using the integer indexes associated with those embedding vectors, it can retrieve the URLs from the parqet file and fetch the corresponding images from the URLs.

Experimenting with CLIP (contd.)

```
##  image_to_image_retrieval_with_clip.py
##  Avi Kak
##  See the previous slide for what this script is meant to do.

import clip
import faiss
import numpy as np
from pathlib import Path
import pandas as pd
import torch
import requests
import PIL
from PIL import Image
from io import BytesIO
from sklearn.metrics.pairwise import cosine_similarity

import matplotlib as mpl
import matplotlib.pyplot as plt
import textwrap
import sys

ind = faiss.read_index("knn.index")
data_dir = Path("embeddings")
##  "df" stands for "dataframe" in Panda
df = pd.concat( pd.read_parquet(parquet_file) for parquet_file in data_dir.glob('*.parquet') )
print(df.head(2))

##  The string args to "df[ ]" you see below are the column headings in the parquet archive
##  The parquet has 15 columns by the way.
image_list = df["image_path"].tolist()            # The 'image_path' is just an integer index associated with
                                                  #         each image (and therefore also with each embedding) in the parquet
caption_list = df["caption"].tolist()             # The captions for all of the 1 million images in this list
url_list = df["url"].tolist()                     # The URLs for ALL of the 1 million images are in this list

##  Let's now randomly choose an index value for the image we will use as the Query Image. The randomly chosen
##  integer index will take us to the image URL and its caption in the parquet database.  And the Image URL
##  will take us to the actual image.
INDEX = np.random.randint(1, len(image_list))
print("\n\nWill use the image indexed %d as the Query Image\n\n" % INDEX)
laion_embedding = np.load("./embeddings/img_emb_0.npy")[INDEX]
laion_embedding_query = np.expand_dims(laion_embedding, 0)
url = pd.read_parquet("./embeddings/metadata_0.parquet")["url"][INDEX]
```

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

Experimenting with CLIP (contd.)

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

```
caption = pd.read_parquet("./embeddings/metadata_0.parquet")["caption"][INDEX]
print("\n\nQuery image url: ", url)
print("\nQuery caption: ", caption)
device = "cuda" if torch.cuda.is_available() else "cpu"
##  Load the Vision Transformer model specified. "B" stands for the  Base model and 32 for the patch size.
##  The model's job is to map a query image to its embedding. The "preprocess" function object returned
##  by "load()" knows how to preprocess an image to make it ready for the CLIP model:
##
model, preprocess = clip.load("ViT-B/32", device=device)
try:
    response = requests.get(url, timeout=10)                          ##  timeout set to 10 seconds
    img = Image.open(BytesIO(response.content))
except requests.exceptions.Timeout:
    sys.exit("\n\nThe request for the query image indexed %d from URL %s timed out.  Timeout is set to 10 seconds." % (INDEX, url))
except Exception:
    sys.exit("\n\nThe image indexed %d is not available at its URL %s." % (INDEX, url))
##  The following block first asks "proprocess()" to get the image ready for the Encoder.  This section
##  of the code also prepares two other images --- "not_available.jpg" and "timeout.jpg" --- for displaying
##  the retrieved results for those cases when the URL did not supply the image and when there was a
##  network timeout on the URL request:
##
image_query = preprocess(img).unsqueeze(0).to(device)
not_available = Image.open("not_available.jpg").resize((64,64), resample=Image.Resampling.BILINEAR)
not_available_im = preprocess(not_available).to(device)
timeout = Image.open("timeout.jpg").resize((64,64), resample=Image.Resampling.BILINEAR)
timeout_im = preprocess(timeout).to(device)
with torch.no_grad():
    clip_embedding_query = model.encode_image(image_query).cpu().numpy()
print("Pre-norm difference between query-image embedding in the Laion database and its embedding produced by the CLIP model:", (laion
clip_embedding_query = clip_embedding_query / np.linalg.norm(clip_embedding_query)
print("Post-norm difference between Laion embedding and the model-produced embedding for the query:", (laion_embedding - clip_embeddi
print("Cosine sim between the two:", cosine_similarity(laion_embedding_query, clip_embedding_query))
##  Display the query image
fig1 = plt.figure(1)
fig1.set_size_inches(15, 8)
```

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

Experimenting with CLIP (contd.)

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

```
plt.title("Query image and text", fontsize=30)
axis = fig1.add_subplot(121)
cap_txt = textwrap.fill( caption, 30 )
axis.text(0.1, 0.5, cap_txt, fontsize=20, transform=axis.transAxes, wrap=True, verticalalignment='center')
axis = fig1.add_subplot(122)
##  The "img" below is the original version of the Query image_query.  That is, before it is fed
##  to the preprocess()
axis.imshow(img)
plt.show()
print("\n\nQuery: Type of laion_embedding:", type(laion_embedding_query))
print("Shape of laion_embedding:", laion_embedding_query.shape)
print("Data type of laion_embedding:", laion_embedding_query.dtype)
print("Dimension of Faiss index:", ind.d)
##  The following command returns from the knn.index the 5 closest embeddings to the embedding in
##  'laion_embedding'.  The variable D contains the distances to these closest neighbors and I
##  contains index integers that point to the embeddings (and thus also to the images). Subsequently,
##  the integer index values in I can be used to retrieve the URLs and the captions associated with
##  the most similar images.
print("\n\nFetching five most similar results:\n\n")
D, I = ind.search( laion_embedding_query.astype('float32'), 5)
top_five_image_arr = []
top_five_caption_arr = []
similarity_scores = []
for sim_dist, i in zip(D[0], I[0]):
    caption_for_image = pd.read_parquet("./embeddings/metadata_0.parquet")["caption"][i]
    url_neigh = pd.read_parquet("./embeddings/metadata_0.parquet")["url"][i]
    try:
        response_in_url = requests.get(url_neigh, timeout=5)                       ### timeout is set to 10 seconds
        imag = Image.open(BytesIO(response_in_url.content))
    except requests.exceptions.Timeout:
        top_five_image_arr.append(timeout)
        top_five_caption_arr.append(caption_for_image)
        similarity_scores.append(0.0)
        print("Index=%d" % i)
```

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

Experimenting with CLIP (contd.)

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

```
        print("Caption=%s" % caption_for_image)
        print("Similarity=%f" % 0.0)
        print(" ")
        continue
    except Exception:
        top_five_image_arr.append(not_available)
        top_five_caption_arr.append(caption_for_image)
        similarity_scores.append(0.0)
        print("Index=%d" % i)
        print("Caption=%s" % caption_for_image)
        print("Similarity=%f" % 0.0)
        print(" ")
        continue
    print("Index=%d" % i)
    print("Caption=%s" % caption_for_image)
    print("Similarity=%f" % sim_dist)
    similarity_scores.append(sim_dist)
    top_five_image_arr.append(imag)
    top_five_caption_arr.append(caption_for_image)
    print(" ")
print("\n\nAll captions for the retrieved images [from the parquet archive]: ", top_five_caption_arr)
print("\n\nAll similarity scores for the retrieved images: ", similarity_scores)
#sys.exit("delib")

##  A more 'elegant' way to construct the 2x5 plot below be to first call "fig,axes = plt.subplots(2,5)" and
##  to then fill axis object (meaning each cell of the 2x15) array with the image and/or text meant for that
##  cell.  However, I ran into a problem with that because when you try to fill a cell with a TEXT OBJECT
##  with a call like "axes[1,3].text(0.5,0.5, caption_string, fontsize=20, transform=ax.transAxes),
##  matplotlib does not like that because it sees "axes[1,3]" as an numpy.ndarray object for which an option
##  like "transform=ax.transAxes" for the purpose of breaking long text lies is not defined.

fig2 = plt.figure(1)
fig2.set_size_inches(35, 12)
plt.title("Retrieved images and text", fontsize=30)
axis = fig2.add_subplot(251)
cap_txt = textwrap.fill( top_five_caption_arr[0], 30 )
axis.text(0.1, 0.5, cap_txt, fontsize=20, transform=axis.transAxes, wrap=True, verticalalignment='center')
axis = fig2.add_subplot(252)
```

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

Experimenting with CLIP (contd.)

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

```
cap_txt = textwrap.fill(top_five_caption_arr[1], 30 )
axis.text(0.1, 0.5, cap_txt, fontsize=20, transform=axis.transAxes, wrap=True, verticalalignment='center')
axis = fig2.add_subplot(253)
cap_txt = textwrap.fill( top_five_caption_arr[2], 30 )
axis.text(0.1, 0.5, cap_txt, fontsize=20, transform=axis.transAxes, wrap=True, verticalalignment='center')
axis = fig2.add_subplot(254)
cap_txt = textwrap.fill( top_five_caption_arr[3], 30 )
axis.text(0.1, 0.5, cap_txt, fontsize=20, transform=axis.transAxes, wrap=True, verticalalignment='center')
axis = fig2.add_subplot(255)
cap_txt = textwrap.fill( top_five_caption_arr[4], 30 )
axis.text(0.1, 0.5, cap_txt, fontsize=20, transform=axis.transAxes, wrap=True, verticalalignment='center')
axis = fig2.add_subplot(256)
im = top_five_image_arr[0]
axis.set_xlabel("similarity score: %f" % similarity_scores[0], fontsize=20)
axis.imshow(im)
axis = fig2.add_subplot(257)
im = top_five_image_arr[1]
axis.set_xlabel("similarity score: %f" % similarity_scores[1], fontsize=20)
axis.imshow(im)
axis = fig2.add_subplot(258)
im = top_five_image_arr[2]
axis.set_xlabel("similarity score: %f" % similarity_scores[2], fontsize=20)
axis.imshow(im)
axis = fig2.add_subplot(259)
im = top_five_image_arr[3]
axis.set_xlabel("similarity score: %f" % similarity_scores[3], fontsize=20)
axis.imshow(im)
axis = fig2.add_subplot(2,5,10)
im = top_five_image_arr[4]
axis.set_xlabel("similarity score: %f" % similarity_scores[4], fontsize=20)
axis.imshow(im)
plt.show()
```

Experimenting with CLIP (contd.)

- As mentioned on Slide 158, when you execute the script just presented by

  ```
  python3   image_to_image_retrieval_with_clip.py
  ```

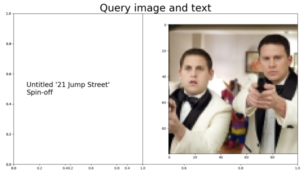
  it will start by displaying a randomly chosen image it is going to use as a query image. The image corresponds to an integer index chosen randomly by the script.

- After choosing an image to serve as the query, the script fetches from the inverted index file knn.index the URLs to the five most similar images+text pairs from the embeddings database.

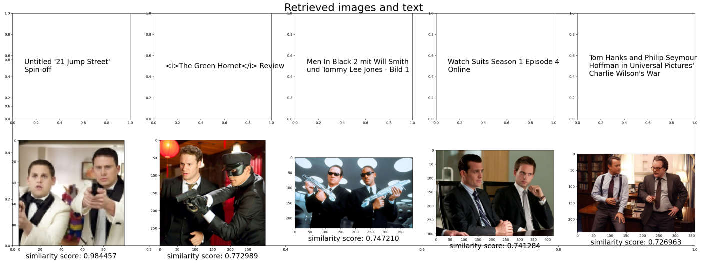
- The next slide shows at the top the query image that was used and, at the bottom, the retrieval results.

- I have included another similar retrieval on Slide 166 just to convey the fact that a large number of image URLs in the CLIP database are outdated. That result displays a "Not available" message for such a URL.

Experimenting with CLIP (contd.)

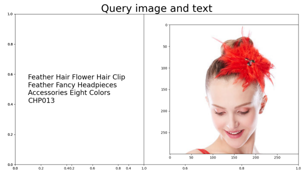


A query image image chosen randomly

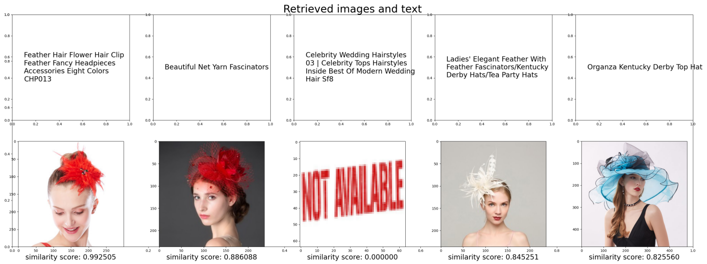

Retrieved images and text
Untitled '21 Jump Street'
Spin-off
<i>The Green Hornet</i> Review
Men In Black 2 mit Will Smith
und Tommy Lee Jones - Bild 1
Watch Suits Season 1 Episode 4
Online
Tom Hanks and Philip Seymour
Hoffman in Universal Pictures'
Charlie Wilson's War
similarity score: 0.984457
similarity score: 0.772989
similarity score: 0.747210
similarity score: 0.741284
similarity score: 0.726963

Five most similar images retrieved along with their text captions

Experimenting with CLIP (contd.)



A query image image chosen randomly


Retrieved images and text
Feather Hair Flower Hair Clip
Feather Fancy Headpieces
Accessories Eight Colors
CHP013
Beautiful Net Yarn Fascinators
Celebrity Wedding Hairstyles
03 | Celebrity Tops Hairstyles
Inside Best Of Modern Wedding
Hair Sf8
Ladies' Elegant Feather With
Feather Fascinators/Kentucky
Derby Hats/Tea Party Hats
Organza Kentucky Derby Top Hat
NOT AVAILABLE
similarity score: 0.992505
similarity score: 0.886088
similarity score: 0.000000
similarity score: 0.845251
similarity score: 0.825560

Five most similar images retrieved along with their text captions

Experimenting with CLIP (contd.)

- With the next script you can draw a sketch on a TKinter canvas with your mouse pointer and have CLIP find for you similar sketches in its database.
- You can invoke the script by

  ```
  python3  sketch_and_retrieve.py
  ```

  and it will present on your screen a canvas on which you can draw a sketch. The canvas comes with "save" and "retrieve" buttons. After you are done sketching, you can save it as a ".png" image and, by clicking on "retrieve", have CLIP deliver to you the five most similar sketches from its database along with their text captions.
- I have included a sketch-based retrieval result in these slides after the script shown next.

Experimenting with CLIP (contd.)

```
##  sketch_and_retrieve.py
##  Avi Kak, Nov 2, 2025

##  See the previous slide for the purpose of this script

import tkinter as tk
from tkinter import filedialog
import PIL
from PIL import Image, ImageDraw
import clip
import faiss
import numpy as np
from pathlib import Path
import pandas as pd
import torch
import requests
from io import BytesIO
from sklearn.metrics.pairwise import cosine_similarity
import matplotlib as mpl
import matplotlib.pyplot as plt
import textwrap
import sys

class SketchApp:
    def __init__(self, master):
        self.master = master
        master.title("Tkinter Sketchpad")

        self.canvas_width = 1200
        self.canvas_height = 900
        self.canvas = tk.Canvas(master, bg="white", width=self.canvas_width, height=self.canvas_height)
        self.canvas.place(relx=0.5, rely=0.5, anchor=tk.CENTER)

        # Create a PIL Image object to draw on in memory
        self.image = Image.new("RGB", (self.canvas_width, self.canvas_height), "white")
        self.draw = ImageDraw.Draw(self.image)

        self.old_x, self.old_y = None, None
        self.canvas.bind("<B1-Motion>", self.paint)                                              ## (A)
        self.canvas.bind("<ButtonRelease-1>", self.reset)                                        ## (B)

        self.save_button = tk.Button(master, text="Save Sketch", command=self.save_sketch)        ## (C)
        self.save_button.pack()
        self.clear_button = tk.Button(master, text="Clear Canvas", command=self.clear_canvas)     ## (D)
        self.clear_button.pack()
```

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

Experimenting with CLIP (contd.)

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

```
    self.retrieve_button = tk.Button(master, text="Retrieve Similar", command=self.retrieve_similar)    ## (E)
    self.retrieve_button.pack()

##  The paint() method defined below is called when the mouse is dragged (<B1-Motion>) as
##  you can from Line (A) above. It draws a line on both the Tkinter canvas and the
##  in-memory PIL image.
def paint(self, event):
    x, y = event.x, event.y
    if self.old_x and self.old_y:
        self.canvas.create_line(self.old_x, self.old_y, x, y,
                                width=5, fill="black", capstyle=tk.ROUND, smooth=tk.TRUE)             ## use width to set the thi
        self.draw.line([self.old_x, self.old_y, x, y], fill="black", width=2)
    self.old_x = x
    self.old_y = y

##  The reset() method defined below is called when the mouse button is released as you can
##  tell from Line (B) above that defines the button action "(<ButtonRelease-1>)". It
##  resets the old_x and old_y coordinates to None, effectively ending the current line
##  segment.
def reset(self, event):
    self.old_x, self.old_y = None, None

##  The save_sketch() defined below is called when the "Save Sketch" button is clicked as
##  you can tell from Line (C) above. It opens a file dialog using
##  "filedialog.asksaveasfilename" to prompt the user for a file name and location, then
##  saves the in-memory self.image to the specified file path.
def save_sketch(self):
    file_path = filedialog.asksaveasfilename(defaultextension=".png",
                                             filetypes=[("PNG files", "*.png"),
                                                        ("JPEG files", "*.jpg"),
                                                        ("All files", "*.*")])
    if file_path:
        self.image.save(file_path)
        print(f"Sketch saved to {file_path}")

##  The clear_canvas() method defined below is called when the "Clear Canvas" button is
##  clicked, as you can tell from Line (D) above. It clears all items from the Tkinter
##  canvas and creates a new, blank PIL.Image to reset the in-memory drawing.
```

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

Experimenting with CLIP (contd.)

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

```
def clear_canvas(self):
    self.canvas.delete("all")
    self.image = Image.new("RGB", (self.canvas_width, self.canvas_height), "white")
    self.draw = ImageDraw.Draw(self.image)

##  The retrieve_similar() method defined below is called when the "Retrieve Similar"
##  button is clicked as you can tell from Line (E) above.  After the first statement
##  that calls for saving the current figure on the canvas, the rest of the code is
##  the same as in my script:
##                              querying_with_your_own_image.py
##
def retrieve_similar(self):
    self.image.save("sketch_for_retrieval.jpg")
    ind = faiss.read_index("knn.index")
    data_dir = Path("embeddings")
    df = pd.concat(
        pd.read_parquet(parquet_file)
        for parquet_file in data_dir.glob('*.parquet')
    )
    image_list = df["image_path"].tolist()
    caption_list = df["caption"].tolist()
    url_list = df["url"].tolist()
    device = "cuda" if torch.cuda.is_available() else "cpu"
    model, preprocess = clip.load("ViT-B/32", device=device)
    query_image_name = "sketch_for_retrieval.jpg"
    query_image = Image.open( query_image_name )
    query_image_im = preprocess(query_image).to(device)
    query_image_embedding = model.encode_image( torch.unsqueeze( query_image_im, dim=0 ) )
    query_image_embedding = query_image_embedding.cpu().detach().numpy().astype('float32')
    not_available = Image.open("not_available.jpg").resize((64,64), resample=Image.Resampling.BILINEAR)
    not_available_im = preprocess(not_available).to(device)
    timeout = Image.open("timeout.jpg").resize((64,64), resample=Image.Resampling.BILINEAR)
    timeout_im = preprocess(timeout).to(device)
    D, I = ind.search( query_image_embedding, 5)
    top_five_image_arr = []
    top_five_caption_arr = []
    similarity_scores = []
```

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

Experimenting with CLIP (contd.)

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

```
for sim_dist, i in zip(D[0], I[0]):
    caption_for_image = caption_list[i]
    url_neigh = url_list[i]
    try:
        response_in_url = requests.get(url_neigh, timeout=5)                       ### timeout is set to 10 seconds
        imag = Image.open(BytesIO(response_in_url.content))
    except requests.exceptions.Timeout:
        top_five_image_arr.append(timeout)
        top_five_caption_arr.append(caption_for_image)
        similarity_scores.append(0.0)
        print("Index=%d" % i)
        print("Caption=%s" % caption_for_image)
        print("Similarity=%f" % 0.0)
        print(" ")
        continue
    except Exception:
        top_five_image_arr.append(not_available)
        top_five_caption_arr.append(caption_for_image)
        similarity_scores.append(0.0)
        print("Index=%d" % i)
        print("Caption=%s" % caption_for_image)
        print("Similarity=%f" % 0.0)
        print(" ")
        continue
    print("Index=%d" % i)
    print("Caption=%s" % caption_for_image)
    print("Similarity=%f" % sim_dist)
    similarity_scores.append(sim_dist)
    top_five_image_arr.append(imag)
    top_five_caption_arr.append(caption_for_image)
    print(" ")

# Display the query image
fig3 = plt.figure(3)
fig3.set_size_inches(15, 8)
plt.title("Your sketch", fontsize=30)
axis = fig3.add_subplot(111)
```

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

Experimenting with CLIP (contd.)

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

```
##  The "query_image" below is the original version of the Query image_query.  That is, before it is fed
##  to the preprocess()
axis.imshow(query_image)
plt.show()

fig2 = plt.figure(2)
fig2.set_size_inches(35, 12)
plt.title("Retrieved images and text", fontsize=30)
axis = fig2.add_subplot(251)
cap_txt = textwrap.fill( top_five_caption_arr[0], 30 )
axis.text(0.1, 0.5, cap_txt, fontsize=20, transform=axis.transAxes, wrap=True, verticalalignment='center')
axis = fig2.add_subplot(252)
cap_txt = textwrap.fill(top_five_caption_arr[1], 30 )
axis.text(0.1, 0.5, cap_txt, fontsize=20, transform=axis.transAxes, wrap=True, verticalalignment='center')
axis = fig2.add_subplot(253)
cap_txt = textwrap.fill( top_five_caption_arr[2], 30 )
axis.text(0.1, 0.5, cap_txt, fontsize=20, transform=axis.transAxes, wrap=True, verticalalignment='center')
axis = fig2.add_subplot(254)
cap_txt = textwrap.fill( top_five_caption_arr[3], 30 )
axis.text(0.1, 0.5, cap_txt, fontsize=20, transform=axis.transAxes, wrap=True, verticalalignment='center')
axis = fig2.add_subplot(255)
cap_txt = textwrap.fill( top_five_caption_arr[4], 30 )
axis.text(0.1, 0.5, cap_txt, fontsize=20, transform=axis.transAxes, wrap=True, verticalalignment='center')
axis = fig2.add_subplot(256)
im = top_five_image_arr[0]
axis.set_xlabel("similarity score: %f" % similarity_scores[0], fontsize=20)
axis.imshow(im)
axis = fig2.add_subplot(257)
im = top_five_image_arr[1]
axis.set_xlabel("similarity score: %f" % similarity_scores[1], fontsize=20)
axis.imshow(im)
axis = fig2.add_subplot(258)
im = top_five_image_arr[2]
axis.set_xlabel("similarity score: %f" % similarity_scores[2], fontsize=20)
axis.imshow(im)
```

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

Experimenting with CLIP (contd.)

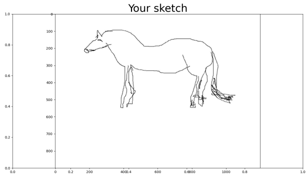
(...... continued from the previous slide)

```
        axis = fig2.add_subplot(259)
        im = top_five_image_arr[3]
        axis.set_xlabel("similarity score: %f" % similarity_scores[3], fontsize=20)
        axis.imshow(im)
        axis = fig2.add_subplot(2,5,10)
        im = top_five_image_arr[4]
        axis.set_xlabel("similarity score: %f" % similarity_scores[4], fontsize=20)
        axis.imshow(im)

        plt.show()

if __name__ == "__main__":
    root = tk.Tk()                          ##  Creates a Tk() instance for the main window.
    root.geometry("1500x1200")              ##  Set the size of the overall Tkinter window
    app = SketchApp(root)                   ##  Initialize the SketchApp object
    root.mainloop()                         ##  Start te Tkinter event loop
```

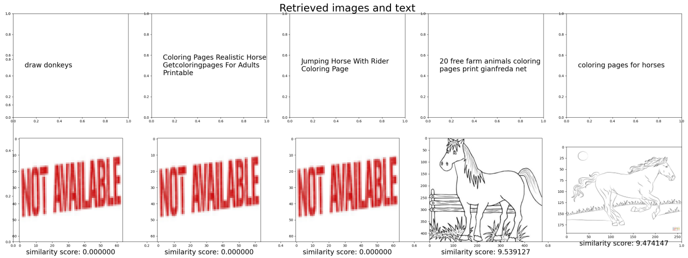
Experimenting with CLIP (contd.)

- Shown on the next slide is a sample result obtained by interacting with the sketch_and_retrieve.py script as mentioned on Slide 167,
- Shown at the top is a silly little sketch I drew with a horse in mind using my mouse pointer on the TKinter canvas created by the script. Subsequently, when I clicked on the "retrieve" button of the GUI, the script received from CLIP the five embedding vectors in the database that were closest to that of my horse sketch.
- As you can see, when it tried to download the actual images from the URLs associated with the closest embedding vectors, it was able to do so for just two of the five URLs. Nonetheless, I thought it was cool that, despite my very poor drawing of a horse, the two skethces that were retrieved successfully are correct.

Experimenting with CLIP (contd.)



A query sketch drawn on the canvas created by the script `sketch_and_retrieve.py`


Retrieved images and text
draw donkeys
Coloring Pages Realistic Horse Getcoloringpages For Adults Printable
Jumping Horse With Rider Coloring Page
20 free farm animals coloring pages print gianfreda net
coloring pages for horses
similarity score: 0.000000
similarity score: 0.000000
similarity score: 0.000000
similarity score: 9.539127
similarity score: 9.474147

Five most similar sketch drawings retrieved along with their text captions