Deep Metric Learning

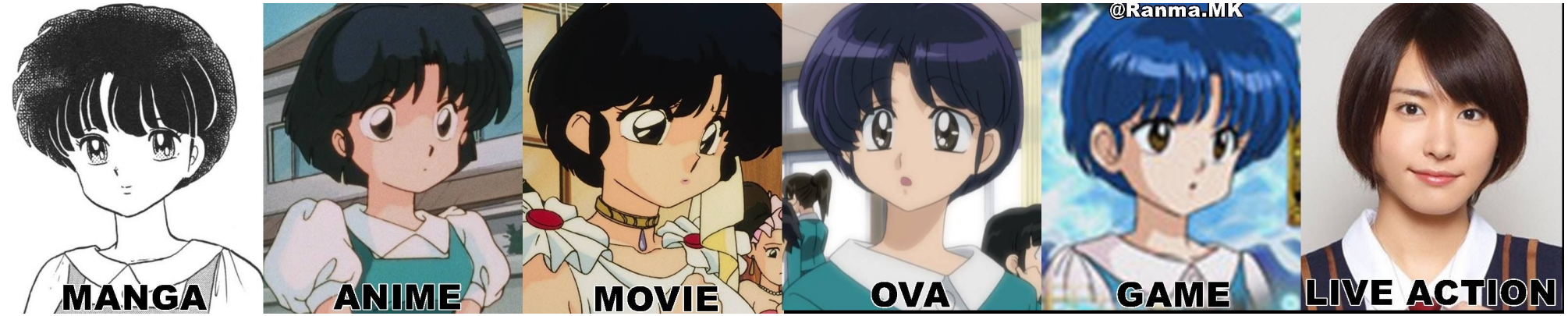
Computer Vision Master, module M5, course 2021-22

Joan Serrat

Contents

1. What is and why
2. Architectures and loss functions
3. Implementation
4. Mining
5. Embedding visualization
6. Applications : image retrieval, face verification, re-id, few-shot learning, patch correspondence, tracking
7. I want to try

1. What is and why



Metric

Function $f(x, y)$ that defines a distance between a pair of elements in a set. Distance is thus a measure of similarity.

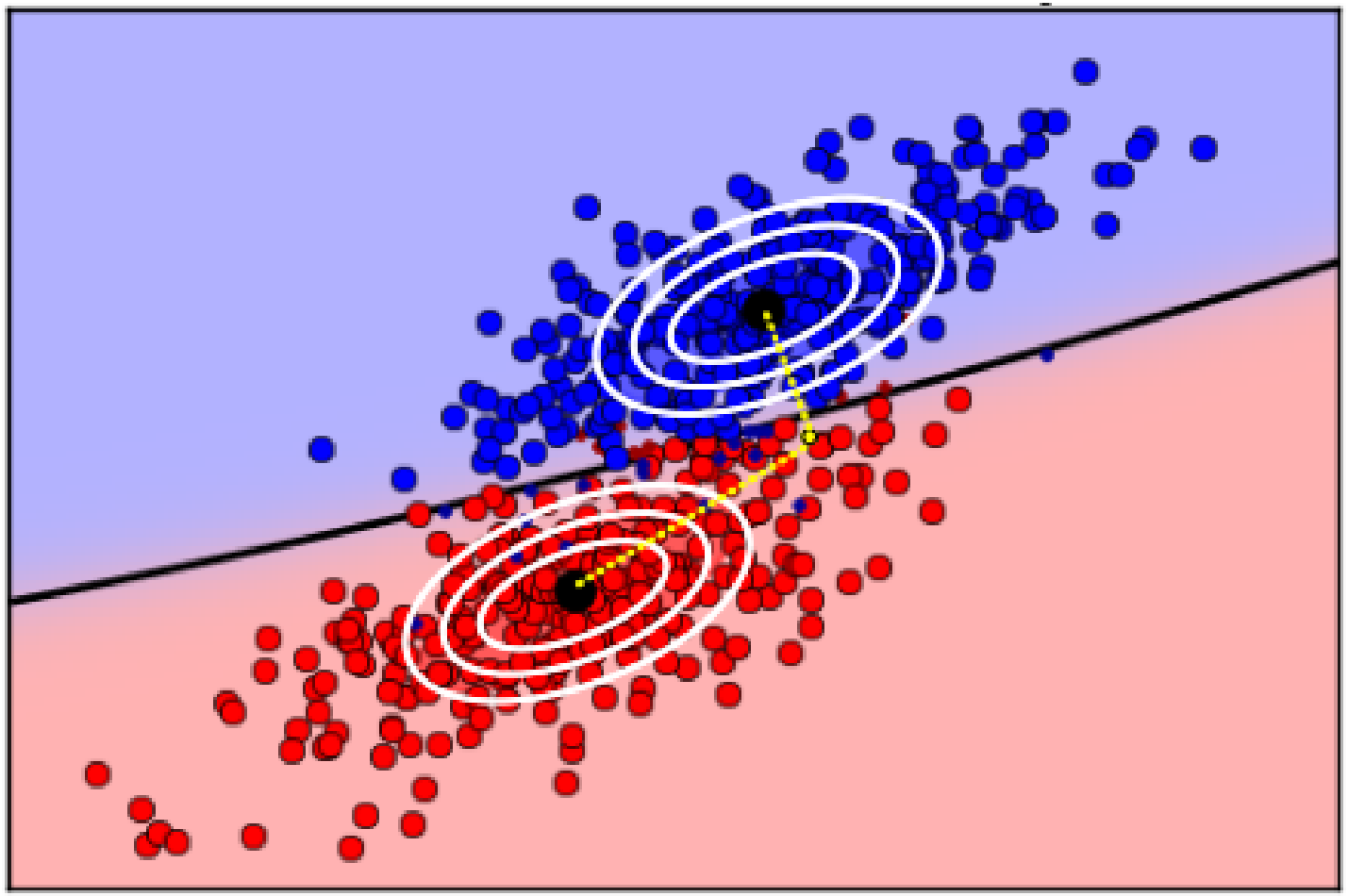
Satisfies

- $f(x, y) \geq 0$
- $f(x, y) = 0$ iif $x = y$
- $f(x, y) = f(y, x)$
- $f(x, z) \leq f(x, y) + f(y, z)$

Example :

- Euclidean distance $||x - y||_2^2$
- Mahalanobis distance $(x - \mu)^t \Sigma^{-1} (x - \mu)$

Mahalanobis: 2d Gaussian samples with non-diagonal $\Sigma_1 = \Sigma_2, \mu_1 \neq \mu_2$



We'll be more interested in distances **learnable** from data, like $(x-y)^\intercal A (x-y)$ in order to **impose our own notion of similarity**

Let S set of pairs of samples considered **similar** (e.g same class) and D set of **dissimilar** pairs.

It is possible to optimize

$$\Sigma^{-1} = \arg\min_A \sum_{x_i, x_j \in S} (x_i - x_j)^\intercal A (x_i - x_j)$$

subject to $$\sum_{x_i, x_j \in D} (x_i - x_j)^\intercal A (x_i - x_j) \geq 1$$

and $A \succeq 0$ (semidefinite positive : $x^\intercal A x \geq 0, \ \forall x$)

Why learn a distance ?

Being able to compare two things, i.e. measure their degree of similarity is **ubiquitous**

- **classification** : distance between samples and class prototypes for nearest class mean, or k-nearest neighbors
  - regular classification : what is x ?
  - distance-based classification : what is x *like* ?
- **image retrieval** : obtain the k most similar images in the *gallery* to a *query* image
- **tracking** : get the most similar bounding box to a target's bbox in previous frame
- **stereo** : find corresponding patch in I_{left} to a patch in I_{right} as the most similar

plus several other tasks we will show in the applications part

Why learn a *deep* metric ?

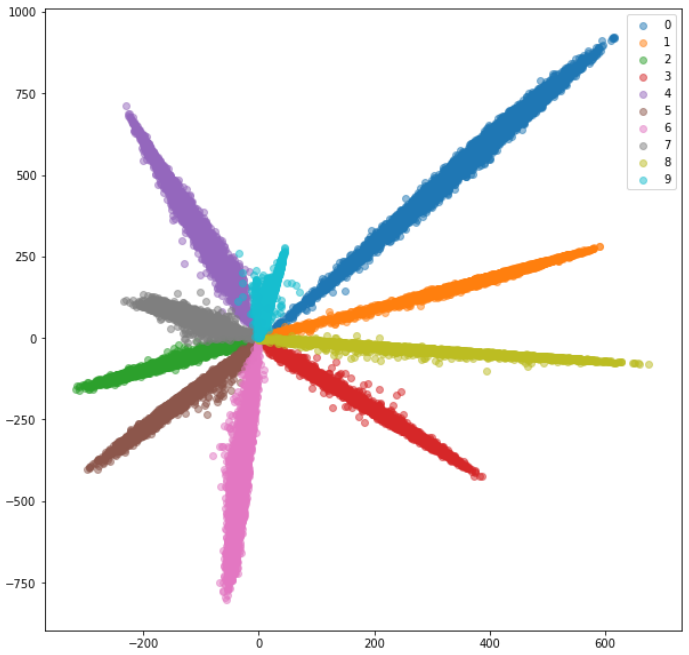
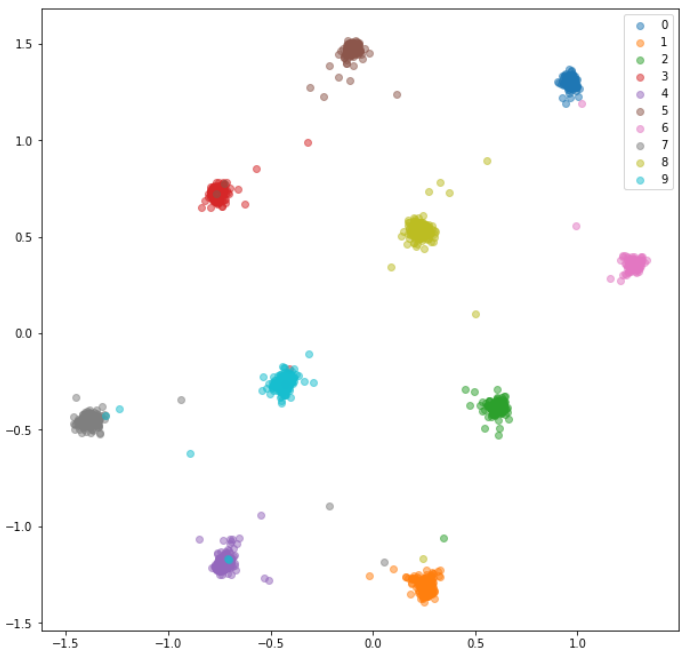
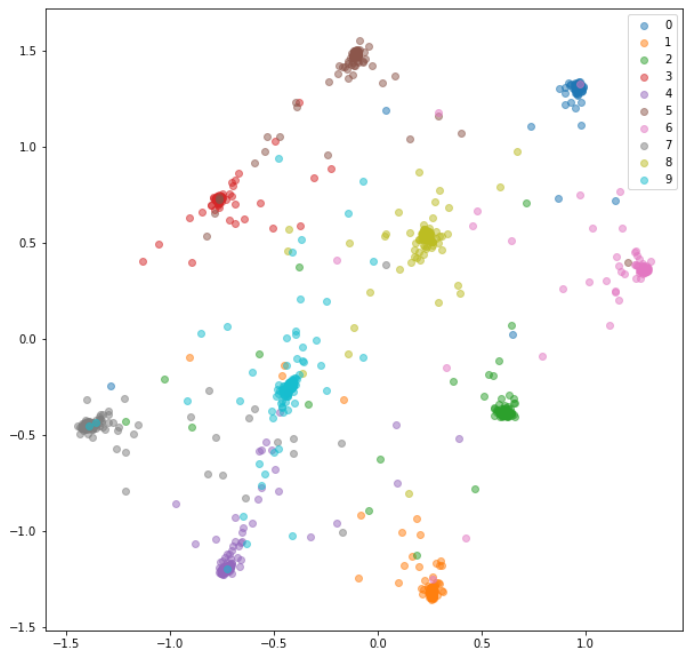
- like in traditional classification, detection, semantic segmentation ... in classic metric learning, metrics were learned from **handcrafted features** $\Rightarrow$ **subobtimal**
- in deep learning **features are learned**
  - transform the represention of an object to a new representation
  - in an **embedding space**, with less dimensions
  - such that objects we say are similar, will be close, and different will be far
- even if the images are from **different domains** ! eg. image and sketch

Example 1 : classification

- learn a distance between MNIST digits according to their class, equivalently
- learn an embedding for MNIST mapping images of a same digit to a cluster, separated from cluster of other 9 digits

Figures from https://github.com/adambielski/siamese-triplet

CNN conv 5x5x32 $\rightarrow$ PReLU $\rightarrow$ MaxPool 2x2 $\rightarrow$ conv 5x5x64 $\rightarrow$ PReLU $\rightarrow$ MaxPool 2x2 $\rightarrow$ Dense 256 $\rightarrow$ PReLU $\rightarrow$ Dense 256 $\rightarrow$ PReLU $\rightarrow$ **Dense 2** $\rightarrow$ PReLU $\rightarrow$ Dense 10 $\rightarrow$ Softmax, Cross-entropy

Cross entropy, Dense 2	Metric learning train	test
		

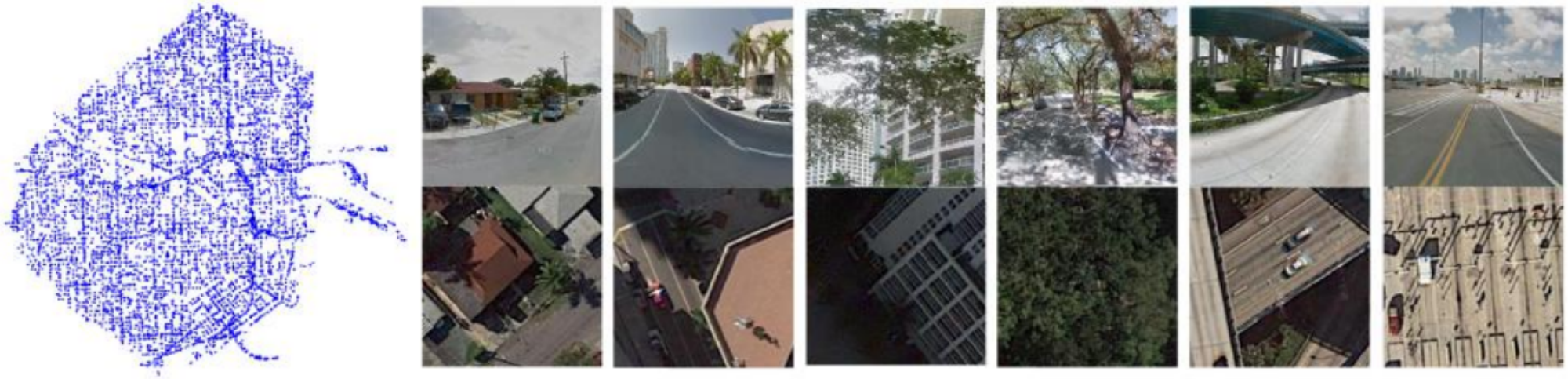
Possibe to classify by distance. But cross-entropy performs better on large datasets[1]

[1] **Significance of Softmax-based features in comparison to distance metric learning-based features. Shota Horiguchi, Daiki Ikami, Kiyoharu Aizawa. T-PAMI, Vol. 42, No. 5, 2020.**

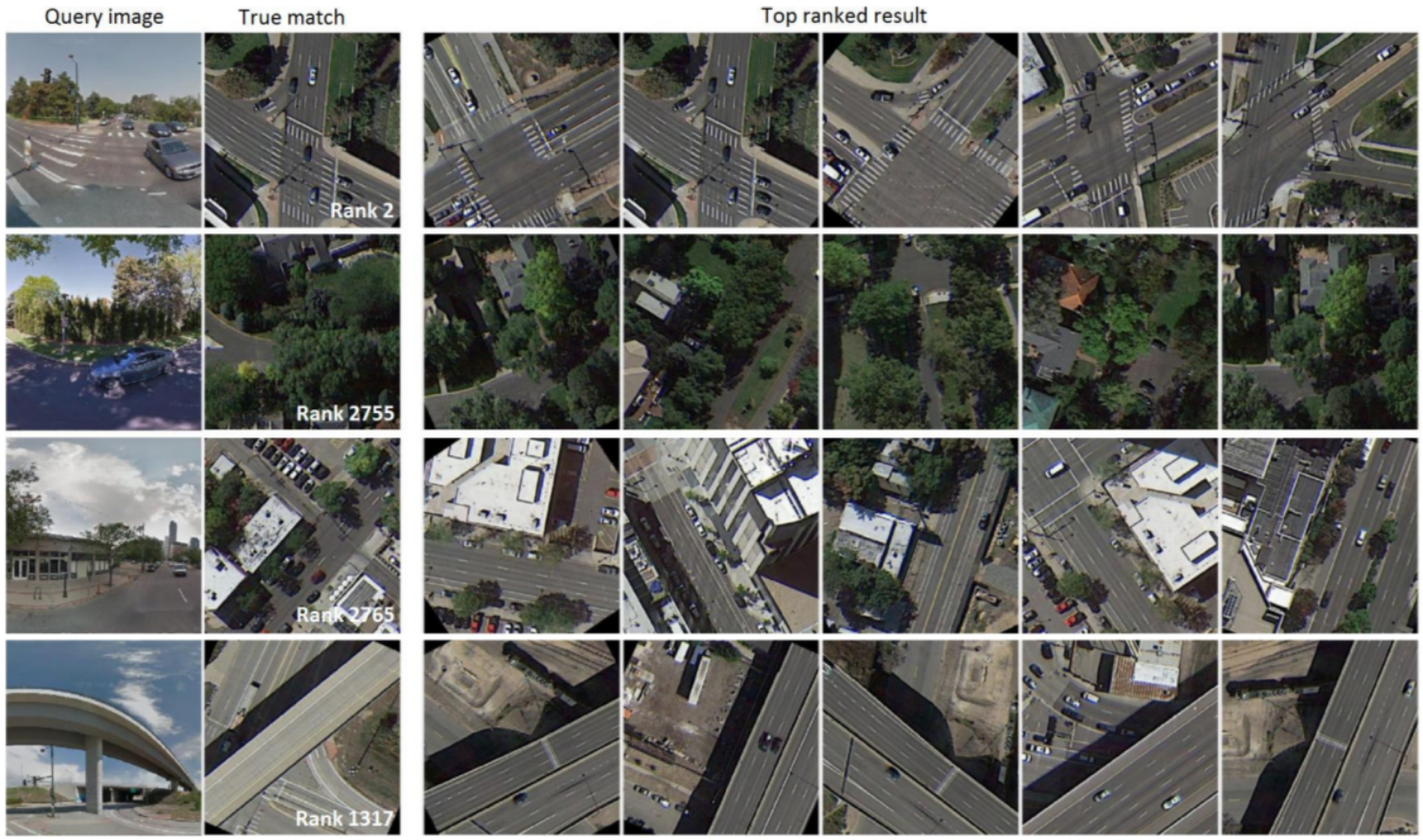
Example 2 : localization

Localizing and orienting street views using overhead imagery. Nam N. Ho, James Hays. *arXiv*:160800161v2.

Train with 900K corresponding pairs (street view, aerial view) from 8 US cities



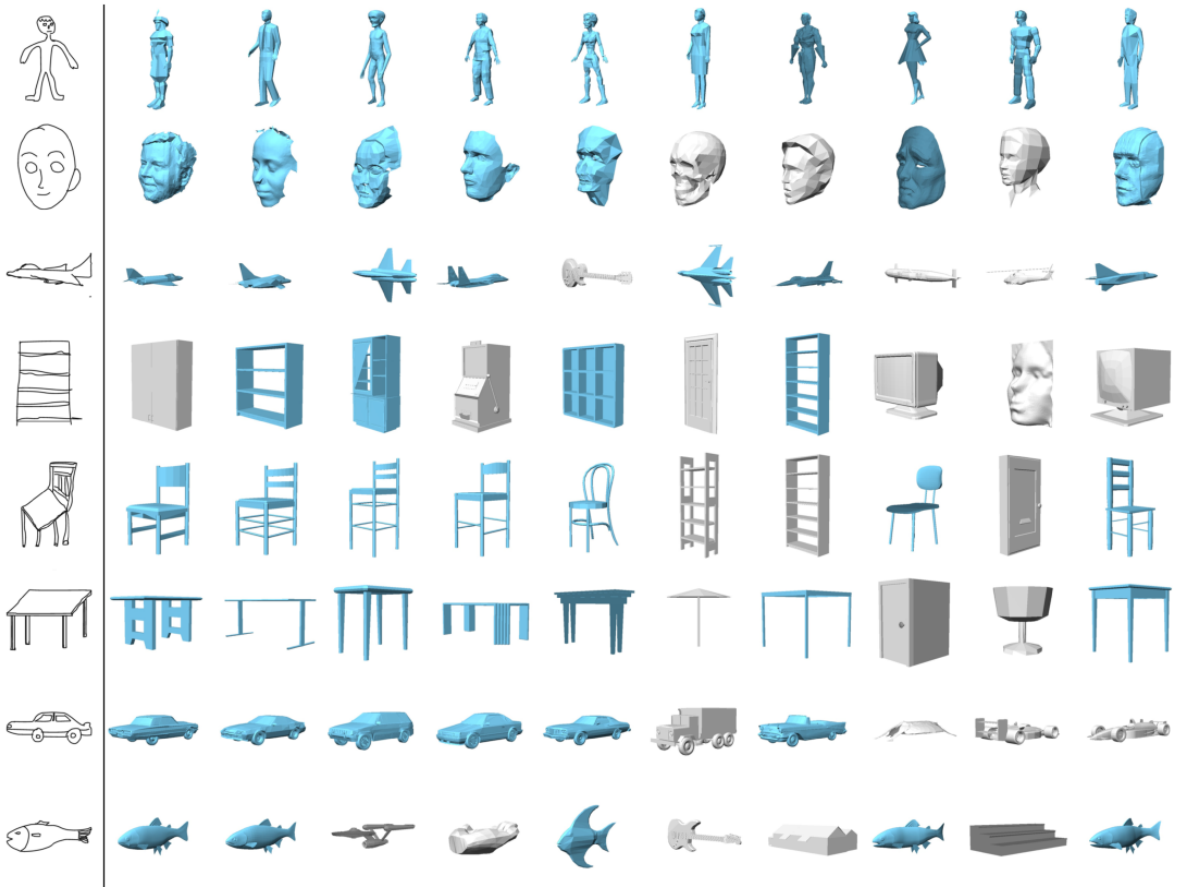
Example 2 : localization



Test with ~70K pairs per city of **3 other cities** $\rightarrow$ recall@1% = 700 images = 0.5, ie, for half of the queries the true match is in the top 1%

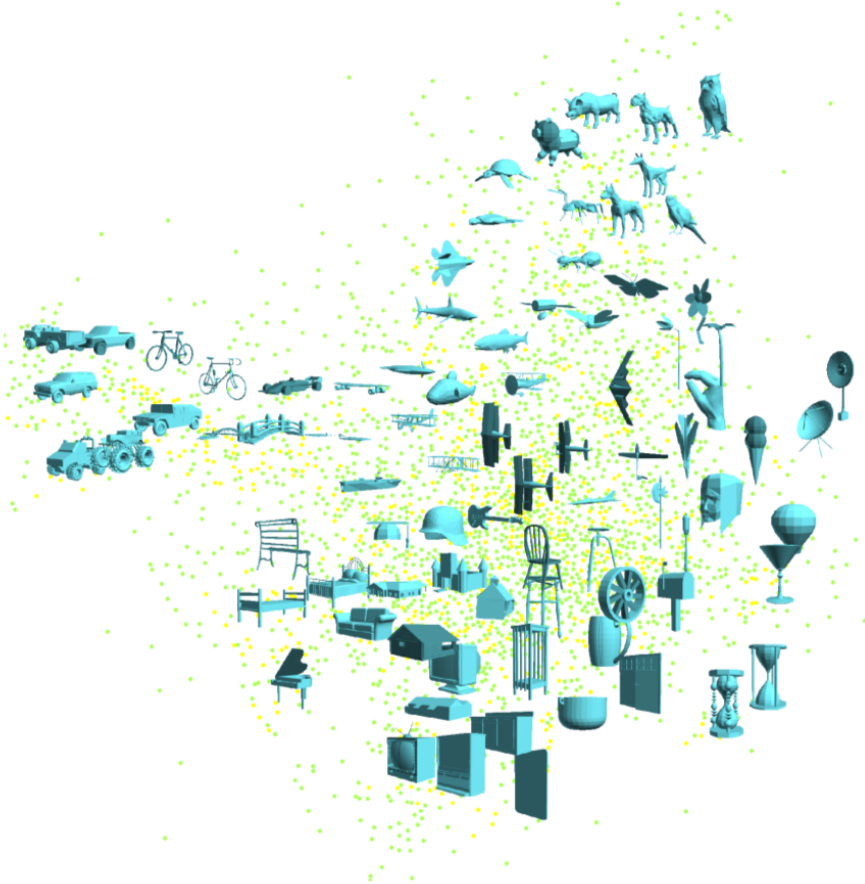
Example 3 : sketch to 3d

Sketch-based 3D Shape Retrieval using Convolutional Neural Networks. Fang Wang, Le Kang, Yi Li. *arXiv*:1504.03504v1.



Try it in this cool online demo

Sketch queries and top 10 answers

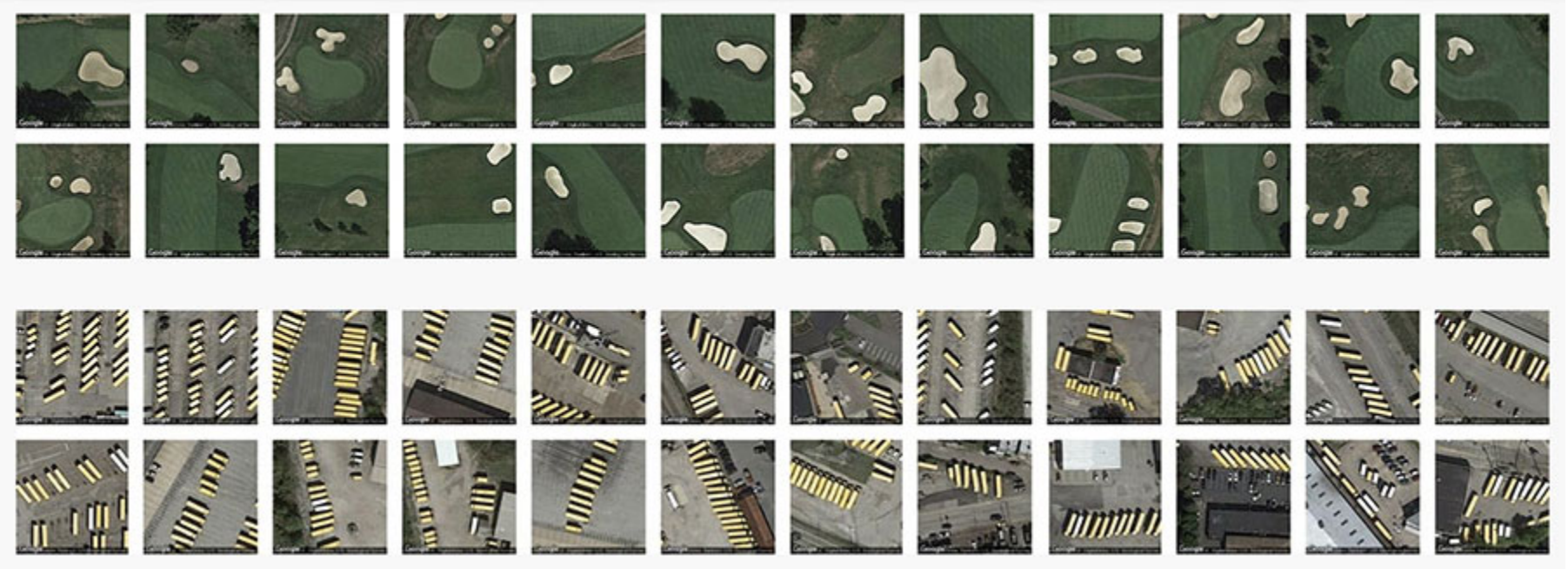


$2d$ PCA projection of sketches and $3d$ models in the $64d$ learned common embeding

Example 4 : *Terrapattern*

"Visual serach engine for satellite imagery" = image retrieval by metric learning.

CMU project 2016



https://www.youtube.com/watch?v=AErDoe-5Ol4 play 1:11 to 3:56

2. Architectures and loss functions

Seminal works

J. Bromley, I. Guyon, Y. Lecun et al. Signature Verification using a "Siamese" Time Delay Neural Network. NIPS'94

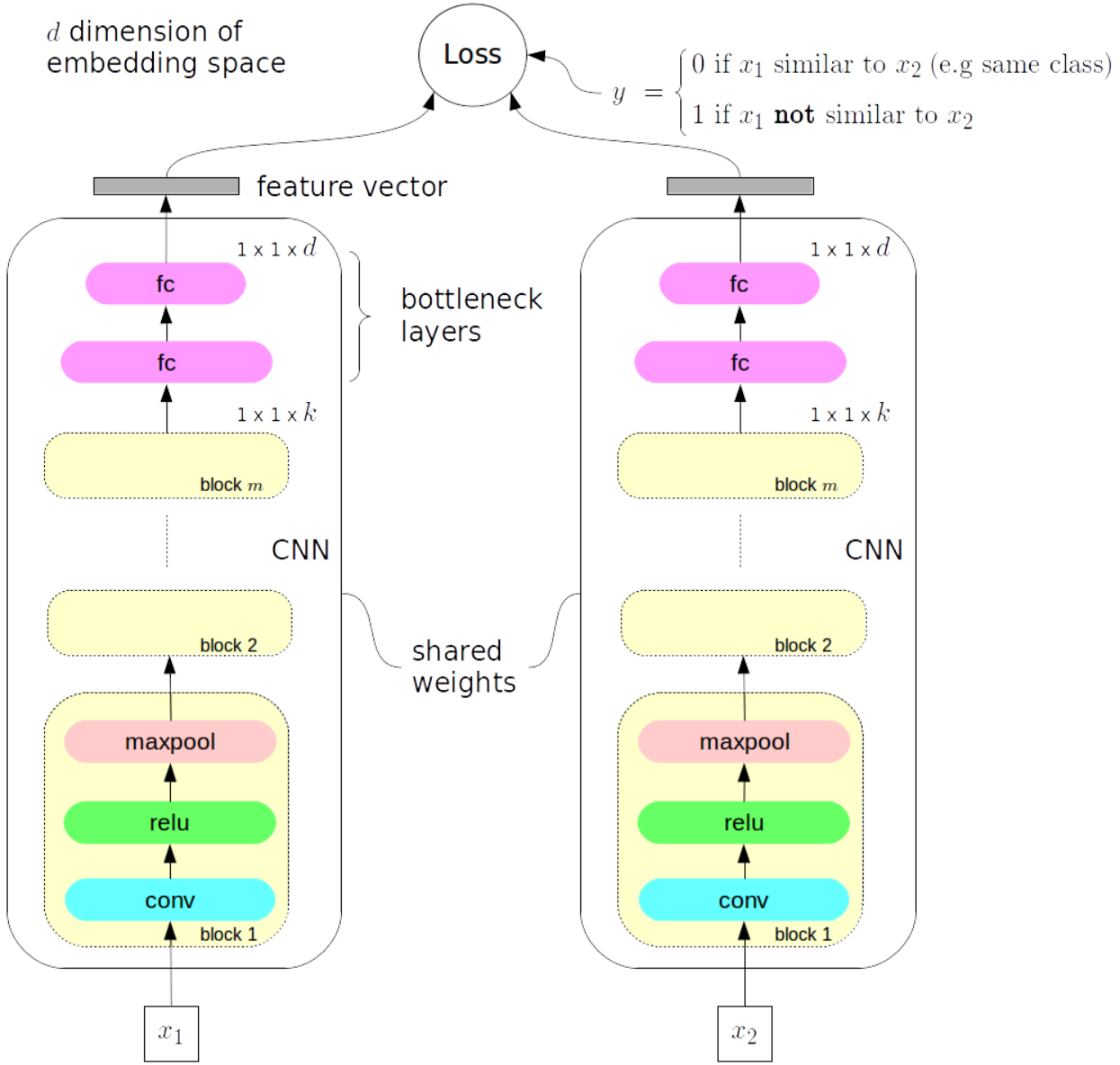
S. Chopra, R. Hadsell, Y. LeCun. Learning a similarity metric discriminatively, with application to face verification. CVPR'05

R. Hadsell, S. Chopra, Y. LeCun. Dimensionality reduction by learning an invariant mapping. CVPR'06

In 2014 several papers at top conferences on descriptors for patch matching, face verification and recognition. On *Labeled Faces in the Wild*, 99.15% accuracy!

Since then, variant architectures, new applications, loss functions, mining strategies ...

Typical Siamese architecture



Contrastive loss

Let be

- $f(x)$ the d-dimensional vector produced by the network branch
- x_1, x_2 inputs to branches
- $y = 0$ if x_1 is deemed similar to x_2 and 1 if not
- m a margin value, usually 1.0

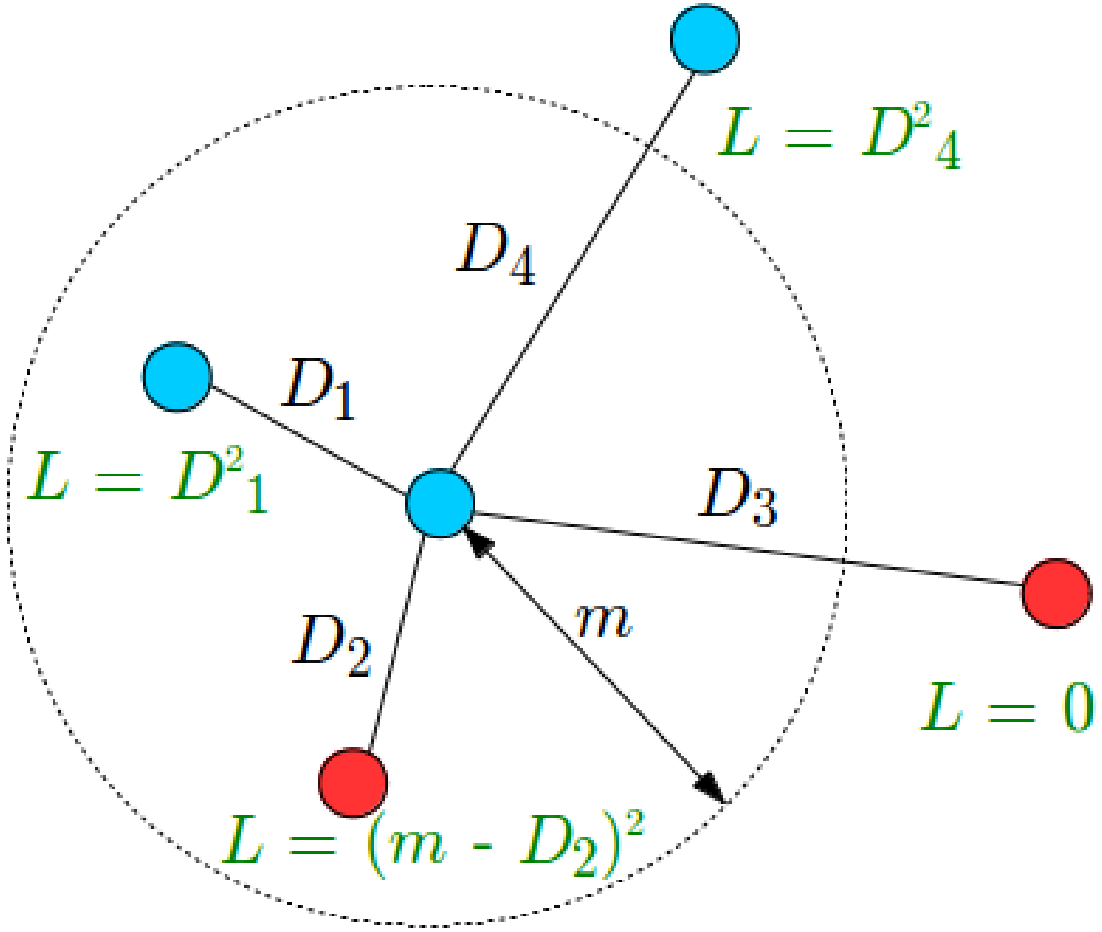
$$D = ||f(x_1) - f(x_2)||_2$$

$$L(x_1, x_2, y) = (1 - y)\ D^2 + y\ (\max(0, m - D))^2$$

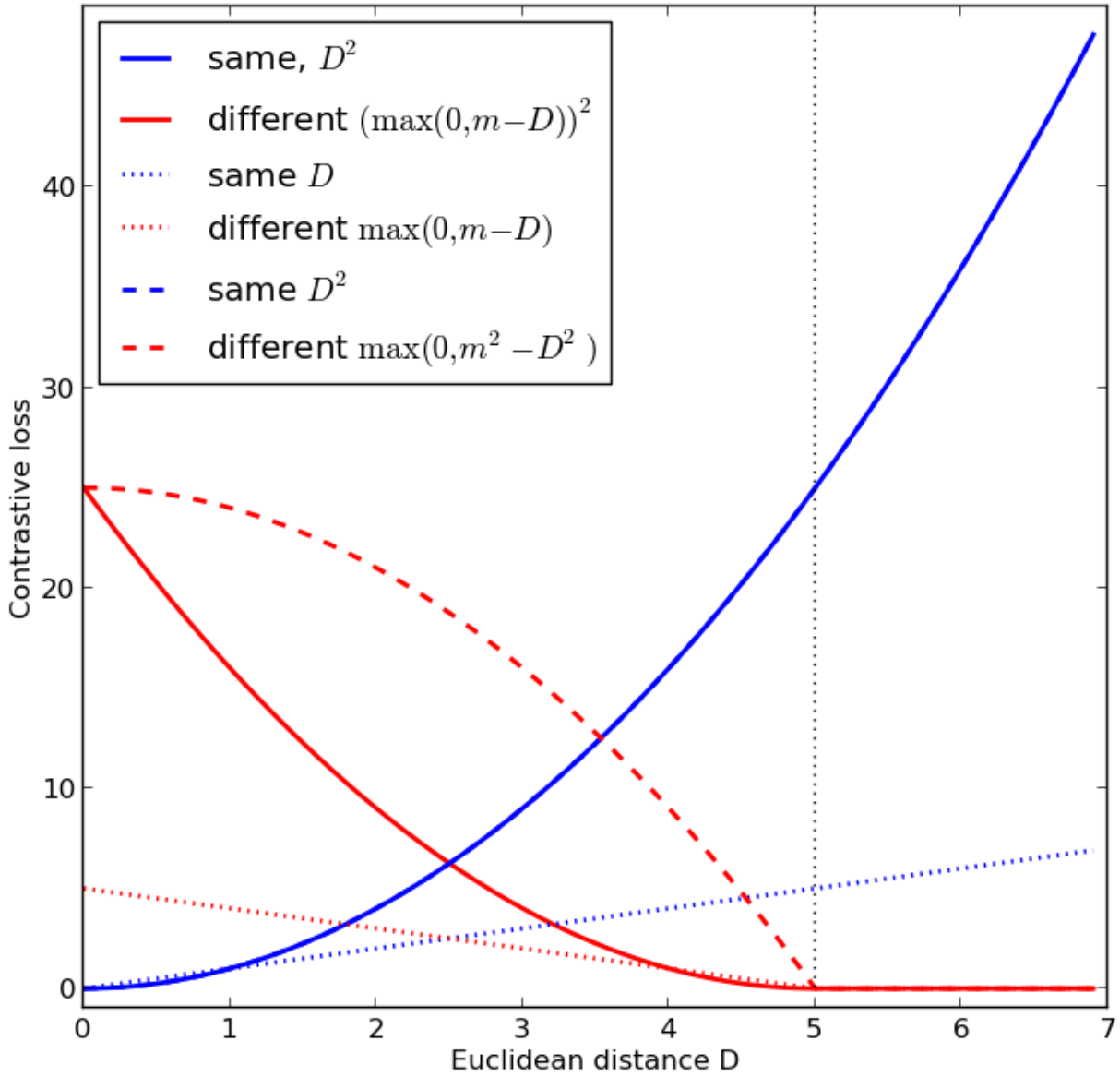
Meaning:

- if x_1, x_2 similar, loss is their squared distance
- if not, loss is zero if m or more appart, and $(m - D)^2$ if not

Contrastive loss



Contrastive loss



Contrastive loss with double margin

Pure contrastive loss similar instances keep *always* being pulled closer, even though they already form tight clusters.

Classes have always some variability, not realistic to ask samples to project to a single point in the embedding.

A **margin m_2 for similar samples** tells the optimizer not to bother with samples already very close.

$$D = ||f(x_1) - f(x_2)||_2^2$$

$$L(x_1, x_2, y) = (1 - y)\ (\max(0,\ D - m_1))^2$$

$$+\ y\ (\max(0,\ m_2 - D))^2$$

Cosine distance

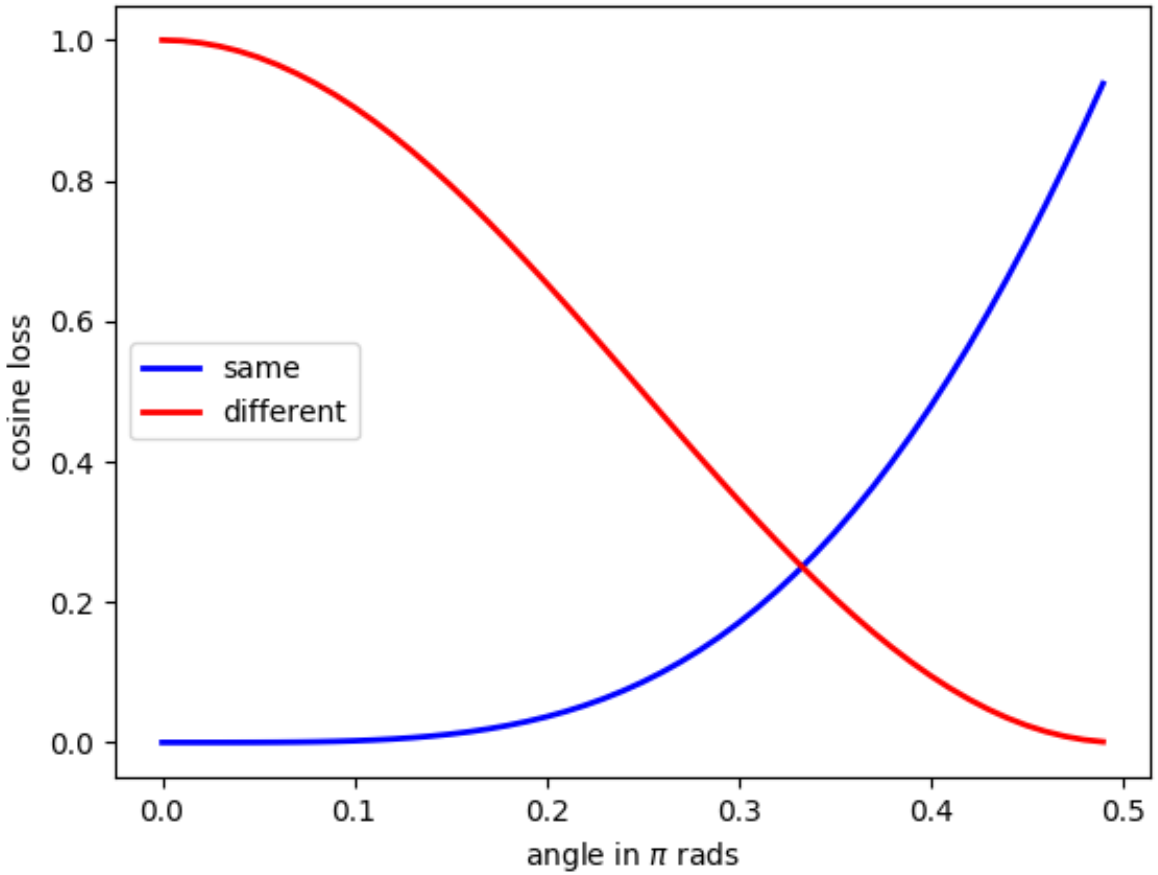
Sometimes the features are L2 normalized

$$f(x) \rightarrow \frac{f(x)}{||f(x)||_2}$$

They live in the surface of a d-dimensional sphere. The Euclidean distance of normalized vectors is the **cosine distance** or scalar product.

$$L(x_i, x_2, y) = \frac{1}{2}\left(y - \frac{f(x_1) \cdot f(x_2)}{||f(x_1)||_2||f(x_2)||_2}\right)^2$$

Cosine distance



Unlike Euclidean distance, this is bounded $\rightarrow$ minimization of contrastive loss is not dominated by pairs generating extreme gradients.

Distance based logistic

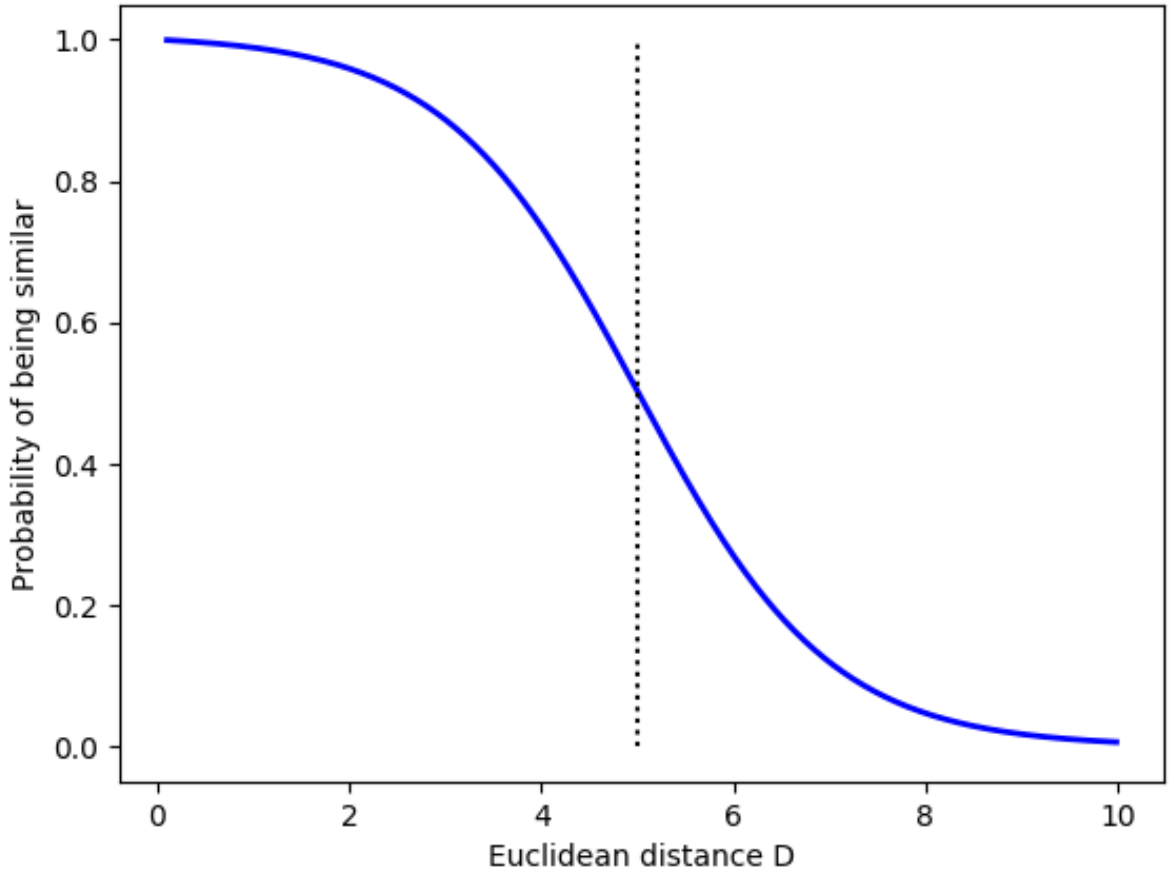
Convert the **Euclidean distance into a probability** of being similar instances, and then use the standard **cross-entropy** loss (also called log-loss) of classification networks :

$$p(x_1, x_2) = \frac{1 + \exp(-m)}{1 + \exp(D(x_1, x_2) - m)}$$

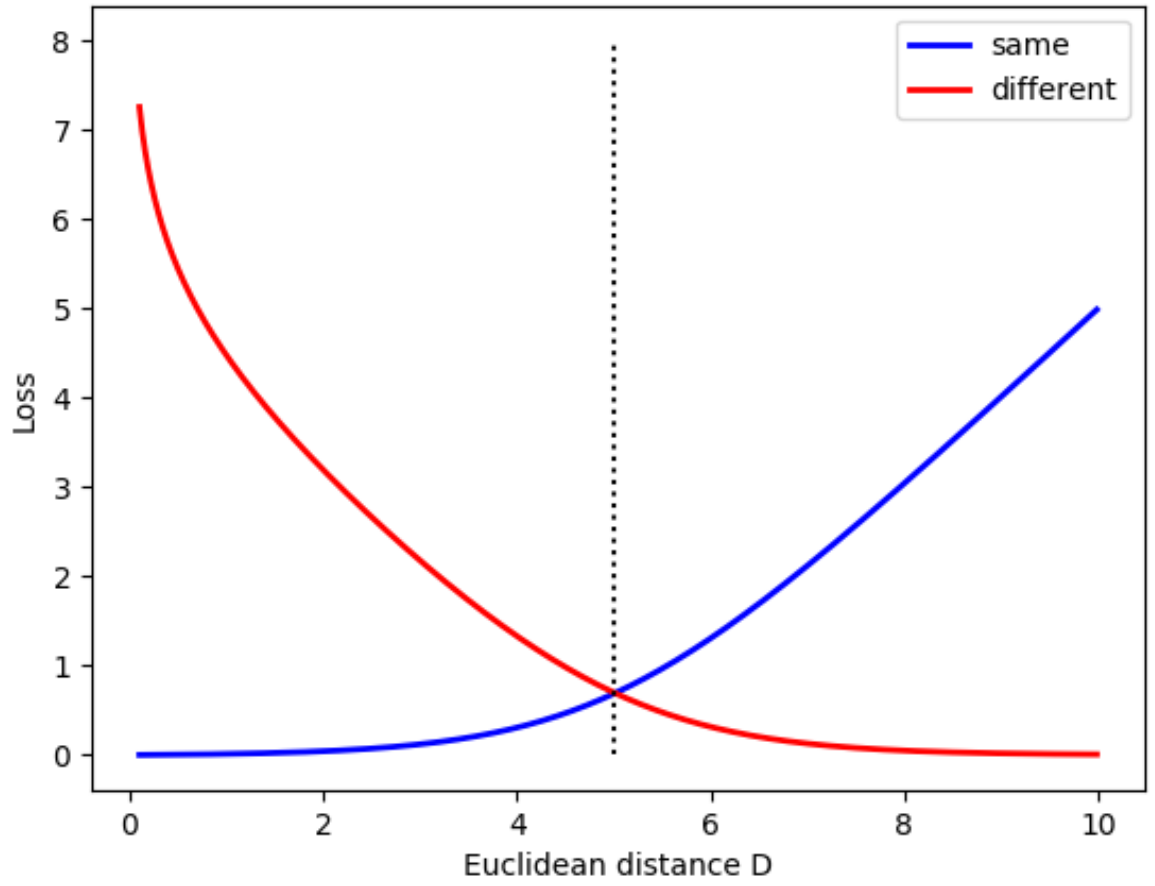
$$L(x_1, x_2, y) = -\big((1-y) \log p(x_1, x_2) + y \log(1 - p(x_1, x_2)) \big)$$

$y = 0$ similar, $y = 1$ different

Distance based logistic $p(x_1, x_2)$



Distance based logistic $L(x_1, x_2, y)$



Distance based logistic

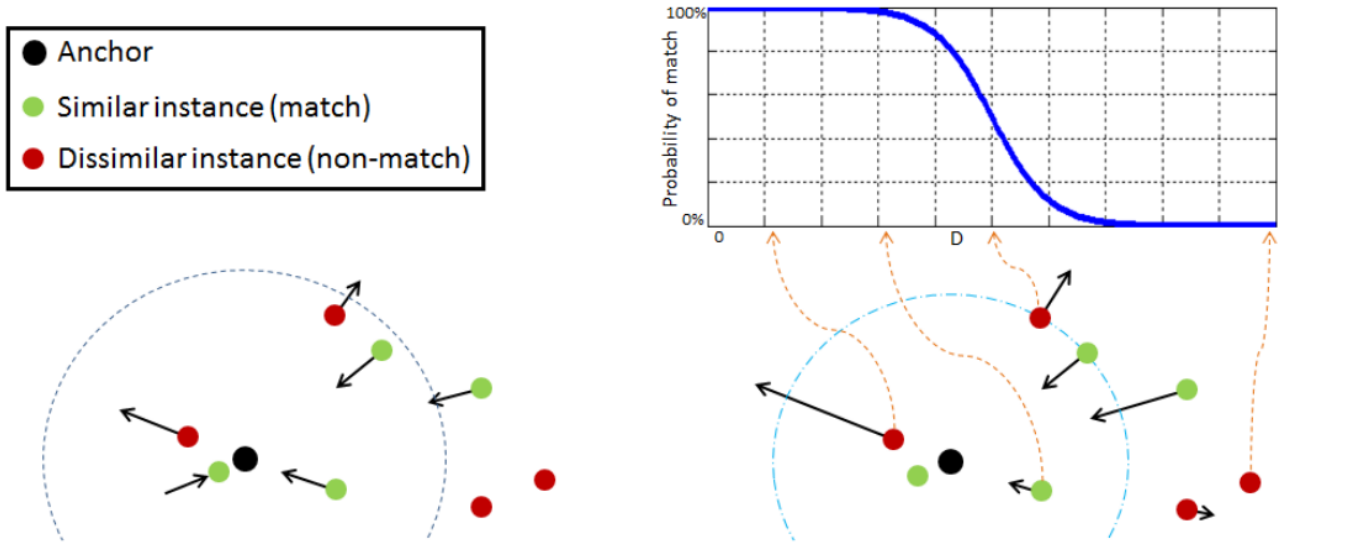


Fig. 4. Visualization of Siamese network training. We represent other instances (matches and non-matches) relative to a fixed instance (called the anchor). Left: with contrastive loss, matched instances keep being pulled closer, while non-matches are pushed away until they are out of the margin boundary, Right: log-loss with DBL: matched/nonmatched instances are pushed away from the "boundary" in the inward/outward direction.

Localizing and orienting street views using overhead imagery. Nam N. Ho, James Hays. *arXiv*:160800161v2.

Variations on the architecture

- all weights of all layers of the two branches are **shared**
- or only those for **first layers** : low level features are the same
- or none, same layers but **independent weights**
- or **different architecture** in each branch

See patch matching in Applications.

Triplet networks

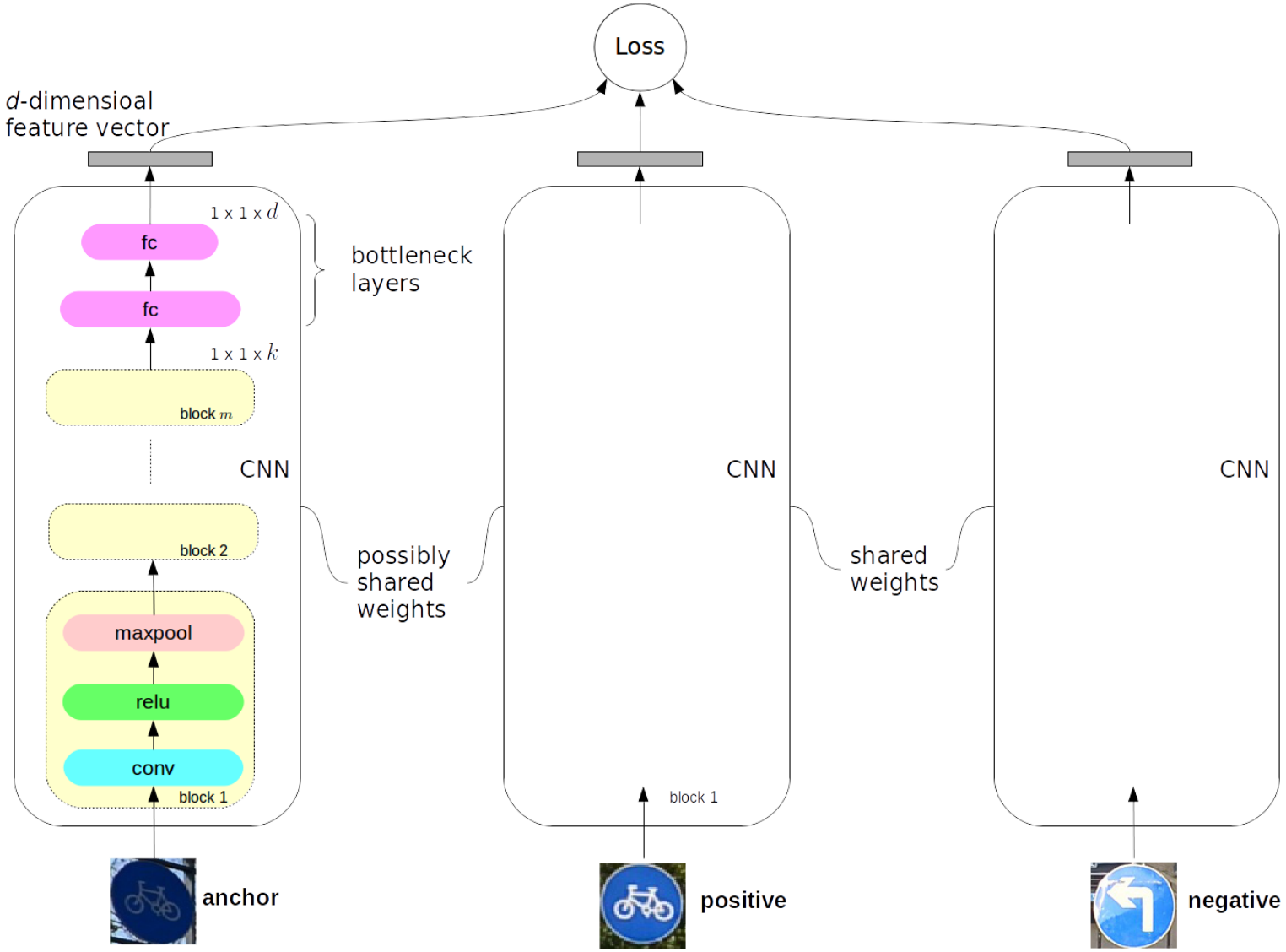
First proposed in

Deep metric learning using triplet network. Elad Hoffer, Nir Ailon. ICLR'15, arXiv:1412:6622v3.

However, the idea of using triplets of samples instead of pairs was already used in *Large-margin nearest neighbours* (LMNN), a pre-deep metric learning method.

Since they were proposed, they compete with Siameses. Not clear who's better, depends on the task and the dataset.

Architecture



Triplet loss

In contrastive loss we asked for x_1, x_2 to be the closest possible if similar, and at least m far away if dissimilar.

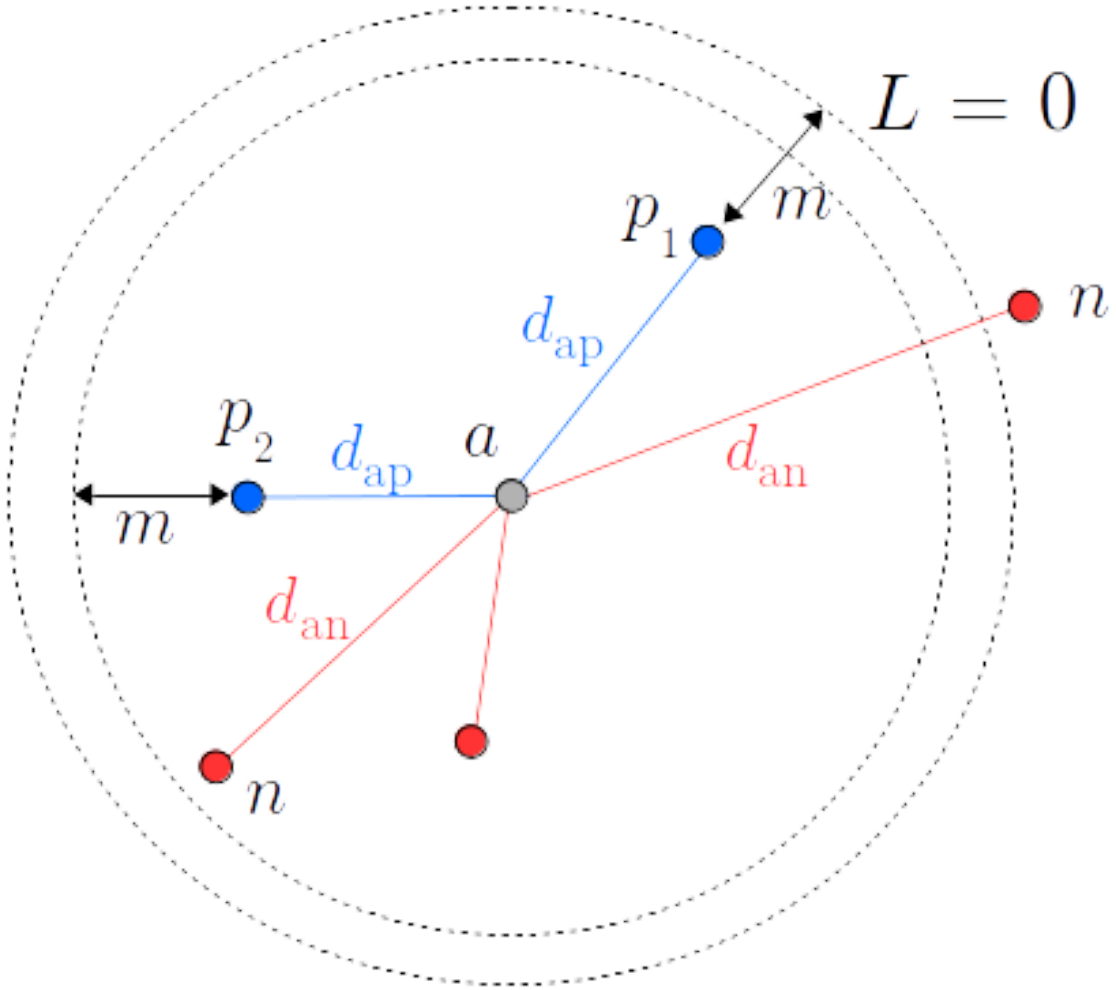
Now we want $distance(\text{anchor, negative})$ to be at least $m + distance(\text{anchor, positive})$

$$d(p,q) = ||p-q||_2 \quad , \quad d(p,q) = 1 - \frac{p \cdot q}{||p||_2 ||q||_2}$$

$$L = \max\{\, 0, m + d(a,p) - d(a,n) \,\}$$

$$L = \max\{\, 0,\ 1 - \frac{d(a,n)}{m + d(a,p)} \,\}$$

Triplet loss



$$L = m + d^2_{ap} - d^2_{an}$$

Triplet vs Siamese

- learn to *rank* 3 samples, **more context** $\rightarrow$ intuitively better than Siamese for image retrieval
- harder to (naif) train : n samples in dataset, $O(n^3)$ **triplets**
- so **mining** is even more necessary
- **clusters are not required to collapse** to a single point, similar samples only need to be closer to each other than to any dissimilar

Triplet vs Siamese

Other losses

NCA loss[1]

Not the most popular but simple and specially adapted to retrieval and classification by k-nearest neighbors.

Originally proposed to learn a linear transform, later easily adapted to deep learning.

[1] **Neighbourhood components analysis. J. Goldberger, S. Roweis, G. Hinton, R. Salakhutdinov. NIPS'05**

NCA loss

k-NN classifier $f(x; w)$

- $x_1, x_2 \dots x_n \in \mathbb{R}^D$ training data
- $c_1, c_2 \dots c_n$ groundtruth class labels
- $\phi(x; w)$ transform with parameters w
- $x \in \mathbb{R}^D,\ \phi(x_{(1)}) \dots \phi(x_{(k)})$ the k nearest neighbours of $\phi(x)$
- $\hat{c} = mode(c_{(1)} \dots c_{(k)})$, most frequent label

NCA loss

Goal: find w such that average classification accuracy for a new (test) data x by k-NN is maximized

Problems

- an *infinitesimal* change in w may change the neighbour graph and affect accuracy by a *finite* amount $\rightarrow f$ piece-wise, non-differentiable
- of course, test data are unknown

Solution

- instead of yes/no NN, *probability* of being NN
- optimize w.r.t training set, as usual

NCA loss

- $p_{ij} = \frac{\exp -||\phi(x_i) - \phi(x_j)||^2}{\sum_{k \neq j} \exp -||\phi(x_i) - \phi(x_k)||^2}$ (softmax of Euclidean distances)

  probability that x_j nearest neighbour of x_i, define $p_{ii} = 0$
- $C_i = \{j \mid c_j = c_i\}$ index of samples of same class than x_i
- $p_i = \sum_{j \in C_i} p_{ij}$

  probability that nearest neighbour of x_i is of its same class = probability x_i correctly classified by 1-NN
- finally, $\hat{w} = \arg\max_w \sum_i p_i$
- remember $\phi(x; w)$ output of neural network with parameters w

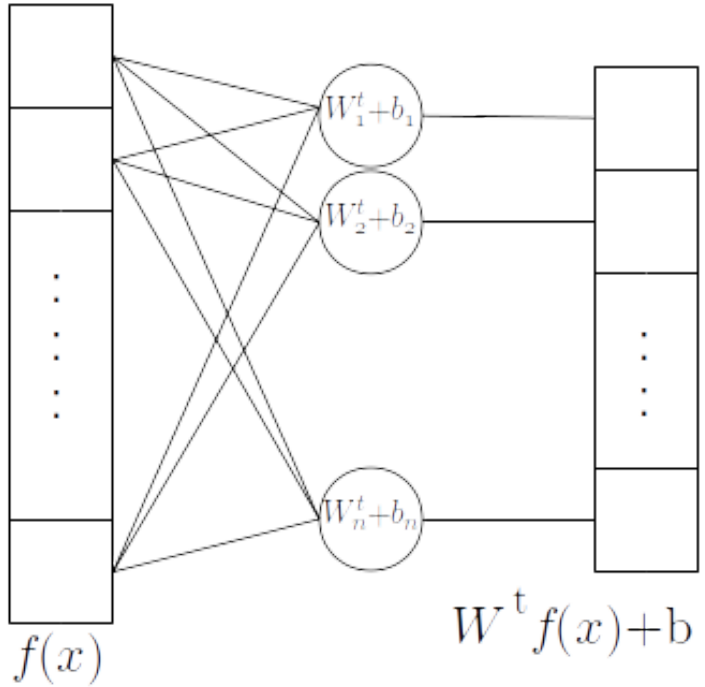
Normalized softmax losses

Cross-entropy is the preferred loss for classification

- batch $(x_i, y_i),\ i = 1 \ldots m$, $m =$ batch size
- x_i image
- $y_i \in [1 \ldots n]$, groundtruth label, with $n =$ number of classes
- $f(x)$ logits or betwork output before last fully connected layer
- fc layer is $W^t f(x) + b$

$$L_{xe} = -\frac{1}{m} \sum_{i=1}^{m} \text{one-hot}(y_i) \ \ \log \ \ \text{softmax}\ f(x_i)$$

$$= -\frac{1}{m} \sum_{i=1}^{m} \log \frac{e^{W_{y_i}^t f(x_i) + b_{y_i}}}{\sum_{j=1}^{n} e^{W_{y_j}^t f(x_i) + b_{y_j}}}$$



Key idea

- by slightly transforming cross-entropy we can, in addition to perform classification, **learn an embedding**
- **typical classification** by $\arg\max_n f(x)$ is equivalent to **minimize a distance** to class "centers" in a certain embedding space

A series of papers use this idea to perform face recognition and verification:

Verification: given two faces *of never seen persons*, do they belong to the same person?

Need to compute a simmilarity or distance between pairs of images.

How ?

- in the fc layer, set the bias to zero, $b_j = 0, j = 1 \dots n$
- normalize (L2) the logits $f(x) \rightarrow \tilde{f}(x)$ such that $||\tilde{f}(x)||_2 = 1$
- also normalize the columns of W, $||\tilde{W}_j^t||_2 = 1$
- then $\tilde{W}_j^t \tilde{f}(x) = 1 \cdot 1 \cdot \cos\theta$, **cosine similarity**
- for unitary vectors u, v, maximize the cosine similarity $=$ minimize their **Euclidean distance** : $||u - v||_2^2 = 2 - 2u^t v = 2 - \cos\theta_{u,v}$
- to minimize

$$L = -\frac{1}{m} \sum_{i=1}^{m} \log \frac{e^{\tilde{W}_{y_i}^t \tilde{f}(x_i)}}{\sum_{j=1}^{n} e^{\tilde{W}_{y_j}^t \tilde{f}(x_i)}}$$

means to make every $\tilde{f}(x_i)$ close to its $\tilde{W}_{y_i}^t$, that becomes the **"center" or representative** of the class $y_i \in [1 \dots n]$

NormFace: L2 Hypersphere Embedding for Face Verification.	$-\frac{1}{m}\sum_{i=1}^{m} log\frac{e^{s\tilde{W}_{y_i}^T\tilde{\mathbf{f}}_i}}{\sum_{j=1}^{n} e^{s\tilde{W}_j^T\tilde{\mathbf{f}}_i}}$
SphereFace: Deep Hypersphere Embedding for Face Recognition.	$\frac{1}{N}\sum_i -\log\left(\frac{e^{\|\boldsymbol{x}_i\|\psi(\theta_{y_i,i})}}{e^{\|\boldsymbol{x}_i\|\psi(\theta_{y_i,i})}+\sum_{j\neq y_i} e^{\|\boldsymbol{x}_i\|\cos(\theta_{j,i})}}\right)$
CosFace: Large Margin Cosine Loss for Deep Face Recognition.	$\frac{1}{N}\sum_i -\log\frac{e^{s(\cos(\theta_{y_i,i})-m)}}{e^{s(\cos(\theta_{y_i,i})-m)}+\sum_{j\neq y_i} e^{s\cos(\theta_{j,i})}}$
ArcFace: Additive Angular Margin Loss for Deep Face Recognition.	$-\frac{1}{N}\sum_{i=1}^{N}\log\frac{e^{s(\cos(\theta_{y_i}+m))}}{e^{s(\cos(\theta_{y_i}+m))}+\sum_{j=1,j\neq y_i}^{n} e^{s\cos\theta_j}}$

All 2017-18. They need a scale $s \gg 1$ to make the gradients larger so that the loss converges.

CosFace: effect of margin m on MNIST. Top $f(x)$, bottom embedding $\tilde{f}(x)$

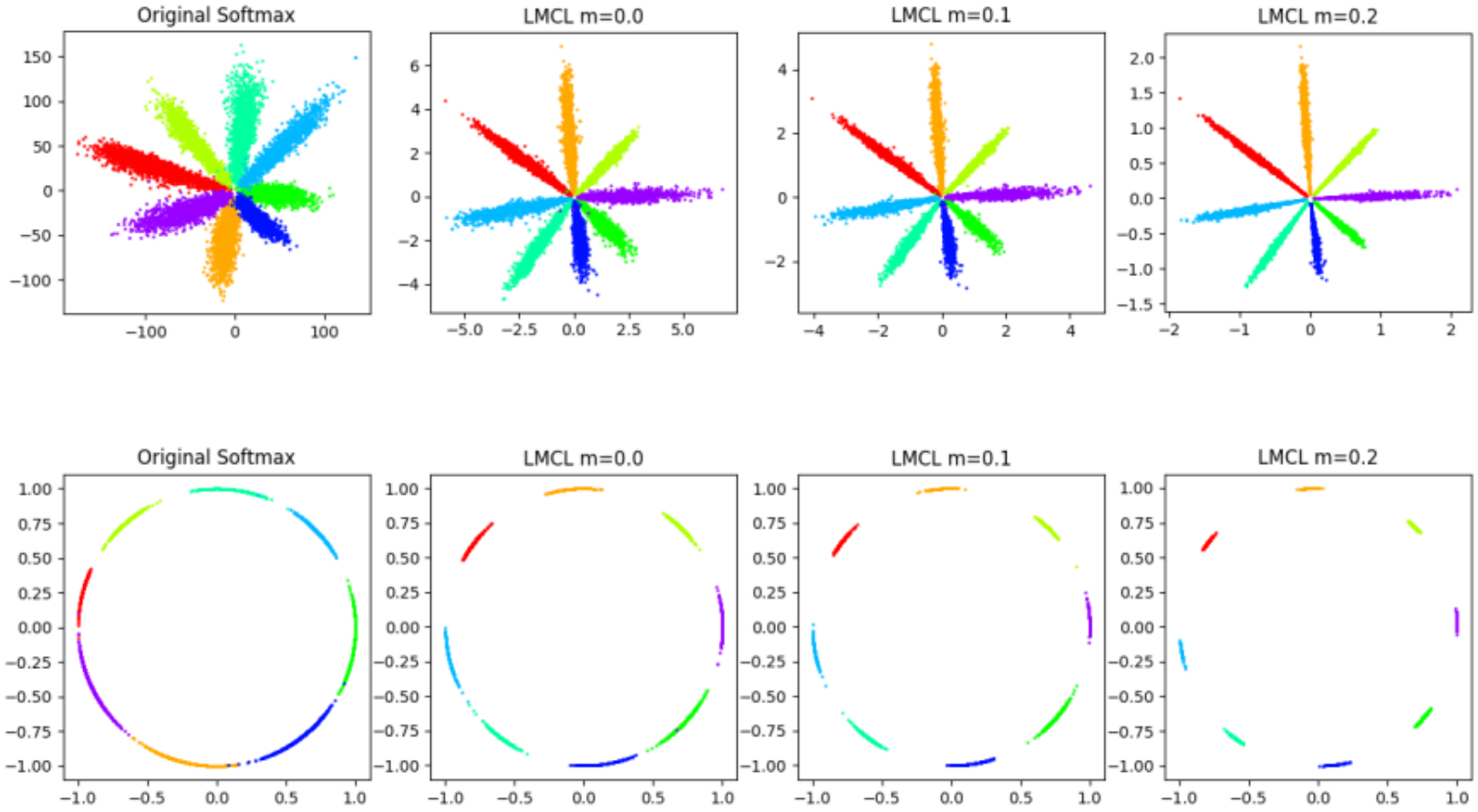


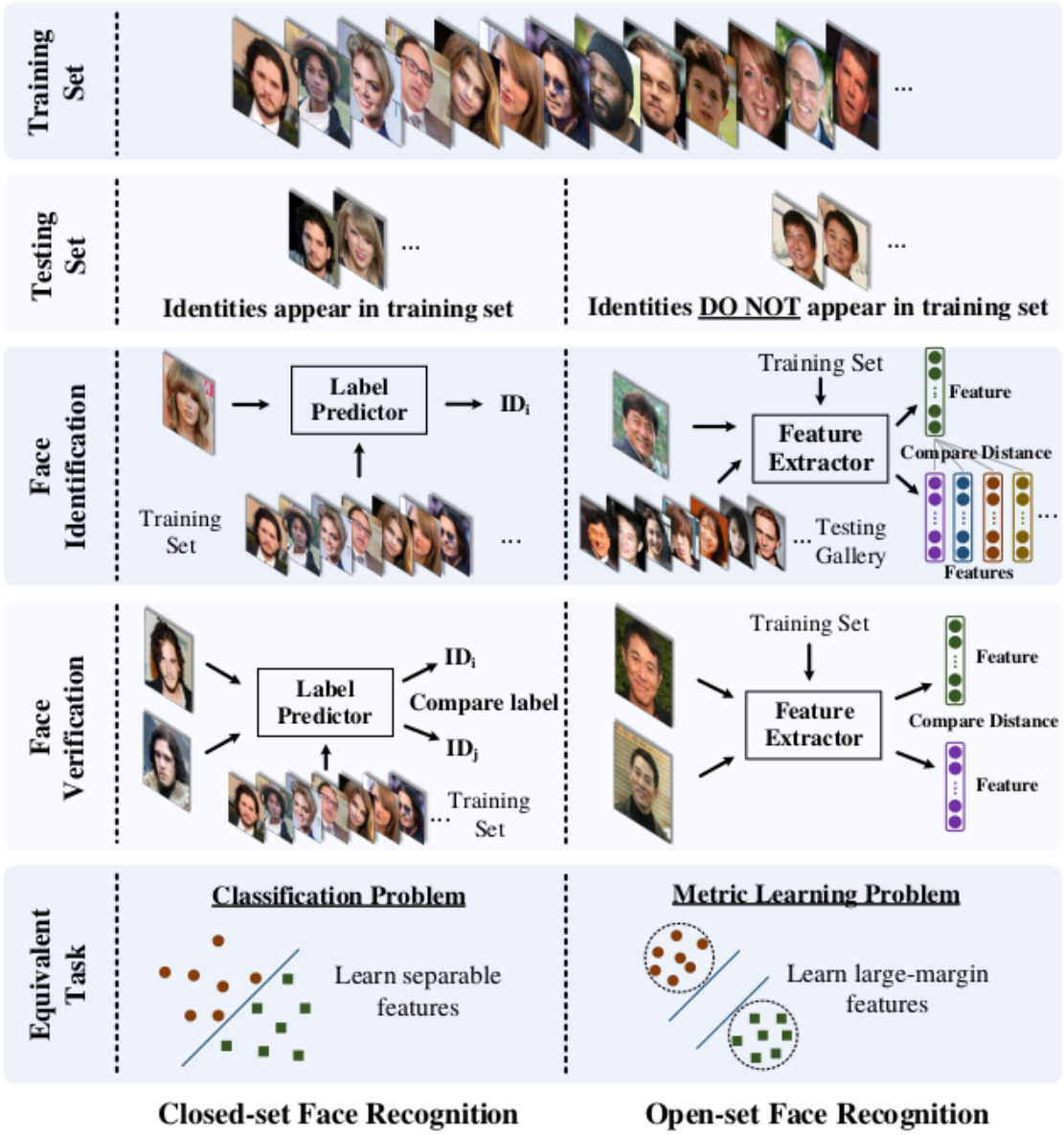
Figure 4. A toy experiment of different loss functions on 8 identities with 2D features. The first row maps the 2D features onto the Euclidean space, while the second row projects the 2D features onto the angular space. The gap becomes evident as the margin term m increases.

Application: face recognition and verification.

Closed classification : new images of known classes at test time

Open : **new classes** at test time.

Open is possible becase they have learned an embedding = a similarity measure, a distance to compare things.

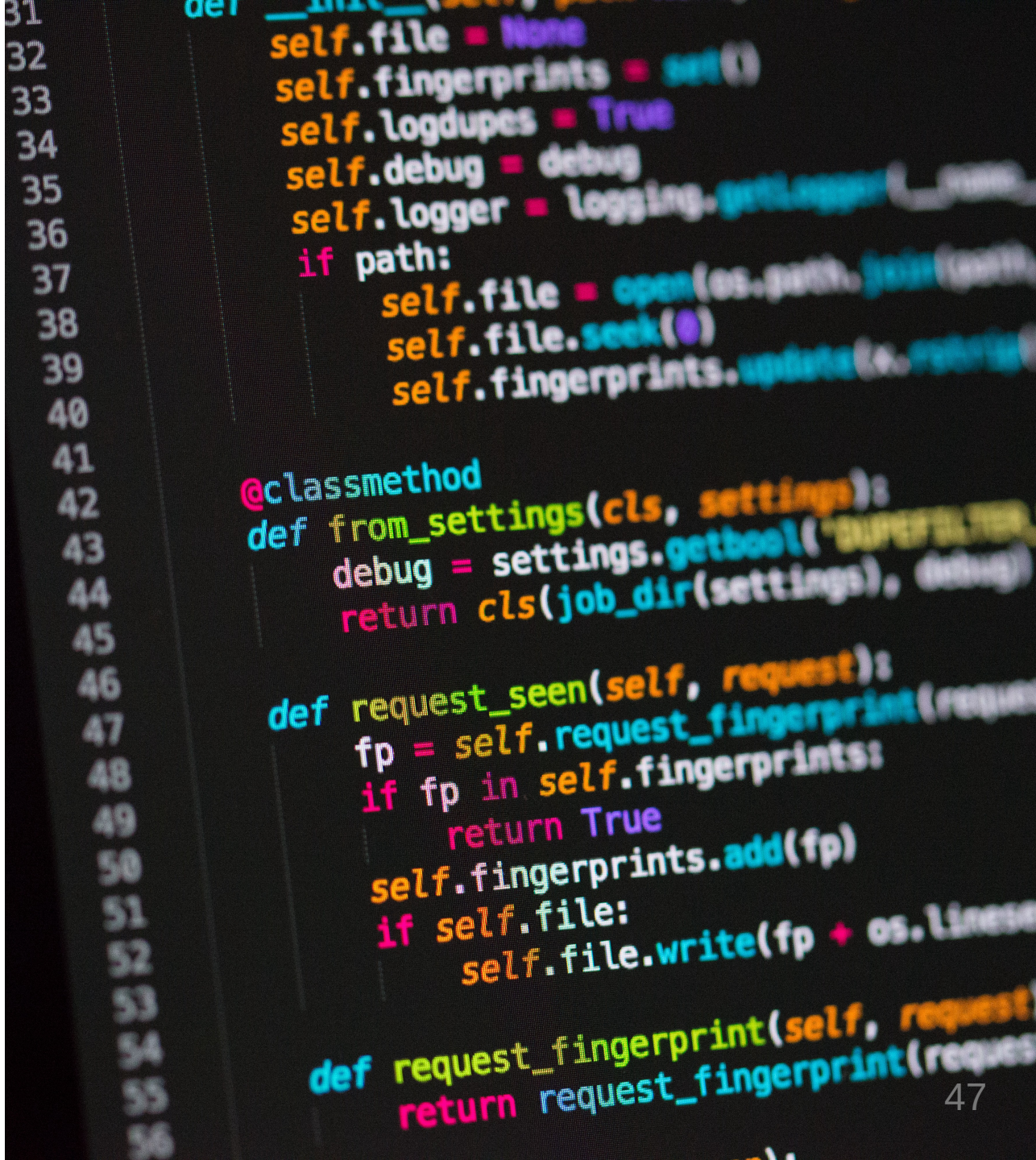


Still other losses

Many other losses proposed along the past years: see a long list here Pytorch Metric Learning losses

What's the best loss ? More on this later...

3. Simple implementation in Pytorch



A Siamese network

By Adam Bielski https://github.com/adambielski/siamese-triplet

```
class EmbeddingNet(nn.Module):
    def __init__(self):
        super(EmbeddingNet, self).__init__()
        self.convnet = nn.Sequential(nn.Conv2d(1, 32, 5), nn.PReLU(),
                                     nn.MaxPool2d(2, stride=2),
                                     nn.Conv2d(32, 64, 5), nn.PReLU(),
                                     nn.MaxPool2d(2, stride=2))
        self.fc = nn.Sequential(nn.Linear(64 * 4 * 4, 256),
                                nn.PReLU(),
                                nn.Linear(256, 256),
                                nn.PReLU(),
                                nn.Linear(256, 2))
    def forward(self, x):
        output = self.convnet(x)
        output = output.view(output.size()[0], -1)
        output = self.fc(output)
        return output

    def get_embedding(self, x):
        return self.forward(x)
```

```
class EmbeddingNetL2(EmbeddingNet):
    def __init__(self):
        super(EmbeddingNetL2, self).__init__()

    def forward(self, x):
        output = super(EmbeddingNetL2, self).forward(x)
        output /= output.pow(2).sum(1, keepdim=True).sqrt()
        return output

    def get_embedding(self, x):
        return self.forward(x)
```

Same thing but normalizes output to 1.0, for cosine similarity for instance.

```
class SiameseNet(nn.Module):
    def __init__(self, embedding_net):
        super(SiameseNet, self).__init__()
        self.embedding_net = embedding_net

    def forward(self, x1, x2):
        output1 = self.embedding_net(x1)
        output2 = self.embedding_net(x2)
        return output1, output2

    def get_embedding(self, x):
        return self.embedding_net(x)
```

`embedding_net` is an object of class `EmbeddingNetL2` or `EmbeddingNet` or anything with a forwarding method `get_embedding()` and a `forward()` with two inputs.

Making a triplet is also a piece of cake.

Triplet network

```
class TripletNet(nn.Module):
    def __init__(self, embedding_net):
        super(TripletNet, self).__init__()
        self.embedding_net = embedding_net

    def forward(self, x1, x2, x3):
        output1 = self.embedding_net(x1)
        output2 = self.embedding_net(x2)
        output3 = self.embedding_net(x3)
        return output1, output2, output3

    def get_embedding(self, x):
        return self.embedding_net(x)
```

Contrastive loss

$$y\,||x_1 - x_2||_2^2 + (1-y)\left(\max\left(0,\, m - ||x_1 - x_2||_2\right)\right)^2$$

```
class ContrastiveLoss(nn.Module):
    """
    Takes embeddings of two samples and a target label == 1 if
    samples are from the same class and label == 0 otherwise
    """

    def __init__(self, margin):
        super(ContrastiveLoss, self).__init__()
        self.margin = margin
        self.eps = 1e-9

    def forward(self, output1, output2, target, size_average=True):
        distances = (output2 - output1).pow(2).sum(1)  # squared distances
        losses = 0.5 * (target.float() * distances +
                        (1 + -1 * target).float() * F.relu(self.margin
                          - (distances + self.eps).sqrt()).pow(2))

        return losses.mean() if size_average else losses.sum()
```

Triplet loss

$$L = \max\{\, 0, m + d(a,p) - d(a,n) \,\}$$

d Euclidean or squared Euclidean

```
class TripletLoss(nn.Module):
    """
    Takes embeddings of an anchor sample, a positive sample
    and a negative sample
    """
    def __init__(self, margin):
        super(TripletLoss, self).__init__()
        self.margin = margin

    def forward(self, anchor, positive, negative, size_average=True):

        distance_positive = (anchor - positive).pow(2).sum(1)  # .pow(.5)
        distance_negative = (anchor - negative).pow(2).sum(1)  # .pow(.5)
        losses = F.relu(distance_positive - distance_negative + self.margin)

        return losses.mean() if size_average else losses.sum()
```

Siamese *online* contrastive loss

You can go **much faster** by making the most of the already computed embeddings of a batch of pairs.

For a batch of n pairs we compute $2n$ embeddings but only n terms are used to minimize the loss.

Why not create **pairs of embeddings** ? $\rightarrow$ much more terms at zero cost $=$ without reading new images or compute new embeddings

- pass through the branch a **batch of n images** $x_k,\ k = 1 \dots n$, being c_k their classes
- take the n embeddings $f(x_k)$ and **make all pairs** $\left(f(x_i), f(x_j), y_{ij}\right)$ for $i = 1 \dots n, j > i$, with $y_{ij} = 1$ if $c_i \neq c_j$ and 0 else
- now we have $n(n-1)/2$ total terms to minimize the contrastive loss

Siamese *online* contrastive loss

```
class OnlineContrastiveLoss(nn.Module):
    """
    Takes a batch of embeddings and corresponding labels. Pairs are generated
    using pair_selector object that take embeddings and targets and return
    indices of positive and negative pairs
    """
    def __init__(self, margin, pair_selector):
        super(OnlineContrastiveLoss, self).__init__()
        self.margin = margin
        self.pair_selector = pair_selector

    def forward(self, embeddings, target):
        positive_pairs, negative_pairs = self.pair_selector.get_pairs(embeddings, target)
        if embeddings.is_cuda:
            positive_pairs = positive_pairs.cuda()
            negative_pairs = negative_pairs.cuda()
        positive_loss = (embeddings[positive_pairs[:, 0]]
                         - embeddings[positive_pairs[:, 1]]).pow(2).sum(1)
        negative_loss = F.relu(self.margin
            - (embeddings[negative_pairs[:, 0]] - embeddings[negative_pairs[:, 1]])
            .pow(2).sum(1).sqrt()).pow(2)
        loss = torch.cat([positive_loss, negative_loss], dim=0)
        return loss.mean()
```

Pair selector

```
def pdist(vectors):
    distance_matrix = -2 * vectors.mm(torch.t(vectors))
                      + vectors.pow(2).sum(dim=1).view(1,-1
                      + vectors.pow(2).sum(dim=1).view(-1, 1)
    return distance_matrix

class PairSelector:
    def __init__(self):
        pass
    def get_pairs(self, embeddings, labels):
        raise NotImplementedError
```

`pdist` Pytorch's analogous to SciPy's `scipy.spatial.distance.pdist` computes the distance between every pair of vectors in a list

`PairSelector` abstract class, template for other pair selectors doing mining.

```
from itertools import combinations

class AllPairsSelector(PairSelector):
    """
    Discards embeddings and generates all possible pairs given labels.
    If balance is True, negative pairs are a random sample to match
    the number of positive samples
    """
    def __init__(self, balance=True):
        super(AllPositivePairSelector, self).__init__()
        self.balance = balance

    def get_pairs(self, embeddings, labels):
        labels = labels.cpu().data.numpy()
        all_pairs = np.array(list(combinations(range(len(labels)), 2)))
        all_pairs = torch.LongTensor(all_pairs)
        positive_pairs = all_pairs[(labels[all_pairs[:, 0]]
                                    == labels[all_pairs[:, 1]]).nonzero()]
        negative_pairs = all_pairs[(labels[all_pairs[:, 0]]
                                    != labels[all_pairs[:, 1]]).nonzero()]
        if self.balance:
            negative_pairs = negative_pairs[torch.randperm(len(negative_pairs))
                                            [:len(positive_pairs)]]
            # as much negatives as positives
        return positive_pairs, negative_pairs
```

Note: tensor `labels` gpu → cpu → numpy → `all_pairs` → gpu tensor. Better do `combinations` in gpu, but how? `torch.cartesian_prod(tensor_a, tensor_b)`

Balanced sampler

However, the vast majority of **pairs** are of dissimilar classes, $y_{ij} = 1$. Instead, program a **balanced batch sampler** inheriting from Pytorch's `BatchSampler` to get k random samples for each of p random classes and gain control on same/different pairs. See class `BalancedBatchSampler`

Same goes for **triplet loss** : from a batch of $3n$ images you can get $6n^2 - 4n$ valid triplets instead of just n.

Advantages

- **each image is read and passes through the network just once**, but appears in many pairs
- in the middle, you can introduce hard pairs **mining** (hard positive and/or hard negative *batch* mining for triplets) by **selecting interesting pairs/triplets** of embeddings.
- now you can train with batches of **1s-10s-100s of thousands of pairs, triplets**, depending on image size.
- consequently, training
    - is much **faster**
    - may converge to a **better minimum**
- it's the normal way found in papers.

4. Mining

What

Mining is to filter or select the training samples to keep the most difficult ones. Here samples = pairs or triplets.

Why

- the number of possible pairs / triplets is large, $O(n^2)/O(n^3)$, $n = 10^k$
- the network learns relatively quickly to cluster similar samples and separate dissimilar ones **when they are "easy"**
- this **seems good but it's not**: from then on, in a batch a large fraction of them become uninformative, that is, produce zero cost $\rightarrow$ zero gradient $\rightarrow$ no weights updates
- therefore, convergence takes much **longer** and probably reach a **not so good local minimum**

How

- look for **hard** $=$ difficult cases and train with them
- a sample is hard means it breaks the margin constraint and has a high cost
- in Siamese, pairs (p, q) similar (e.g. same class) but far away in the embedding space, or the opposite,

$$d(p_a, q_b) < m, a \neq b, \quad d(p_a, q_b) \gg 0, a = b$$

- in triplets, $\{(a, p, n) \mid d(a, n) < d(a, p) + m\}$
- we can mine
  - **hard negatives** $q_{b \neq a}$ given p_a, n given a, p and/or
  - **hard positives** q_a given p_a, p given a, n

hard positives hard negatives

How to mine

- **offline** hard mining : transform the **whole training set** and look for the hardest pairs / triplets, from time to time
- **online**, also batch or semi-hard : do this only for the samples of **current batch**
- offline is much more costly and can make training go bad as it always samples **too difficult** cases
- online is much faster and **not so hard**, as only a small subset of the training set is considered each time

A mining strategy for contrastive loss

Deep Metric Learning via Lifted Structured Feature Embedding. Hyun Oh Song, Yu Xiang, Stefanie Jegelka, Silvio Savarese (Stanford U., MIT). CVPR'16.

Standard contrastive loss for batch of pairs in batch B

$$J = \sum_{(i,j) \in B} y_{ij}\, d_{i,j}^2 + (1 - y_{ij})\, [m - d_{ij}^2]_+$$

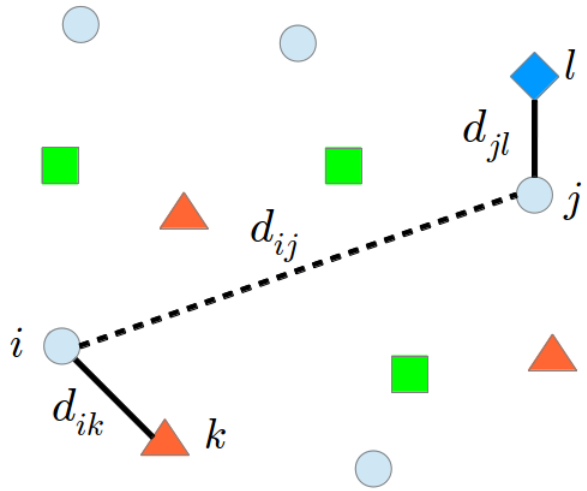
Idea: instead of just $|B|$ pairs (positive and negative), consider **all possible pairs and mine hardest negative for each positive pair**

P, N set of positive, negative pairs **in batch** B

$$J = \sum_{(i,j) \in P} \max\left(0, J_{ij}\right)$$

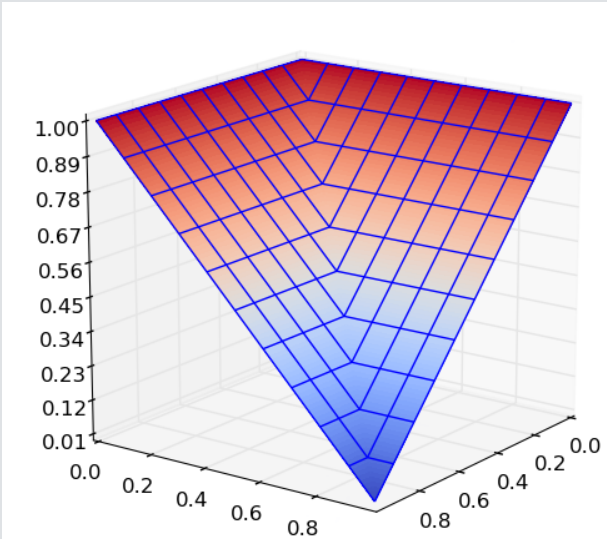
$$J_{ij} = \max\left(\max_{(i,k) \in N} (m - d_{ik}), \max_{(j,l) \in N} (m - d_{jl}) \right) + d_{ij}$$

maximize dist to hardest negative wrt i and hardest negative wrt j, minimize dist positives i, j

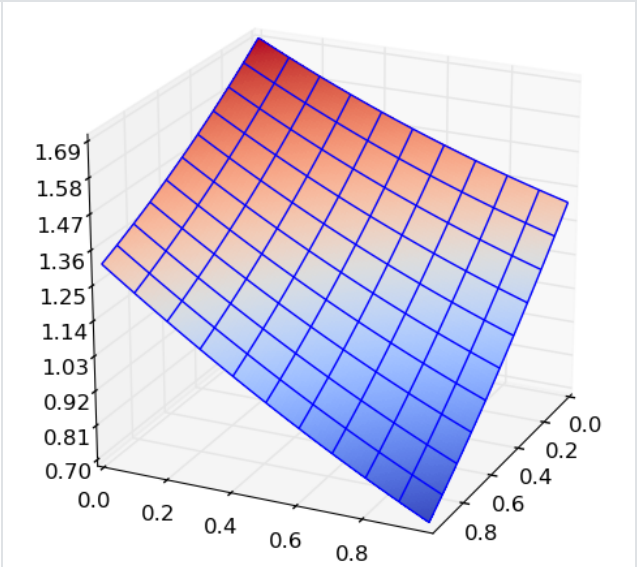


But they observe nested maxs and hardest negatives to cause convergence to bad local minima $\rightarrow$ replace loss by **smooth upper bound**

$$J_{ij} = \log\left(\sum_{(i,k) \in N} e^{m - d_{ik}} + \sum_{(j,l) \in N} e^{m - d_{jl}} \right) + d_{ij}$$



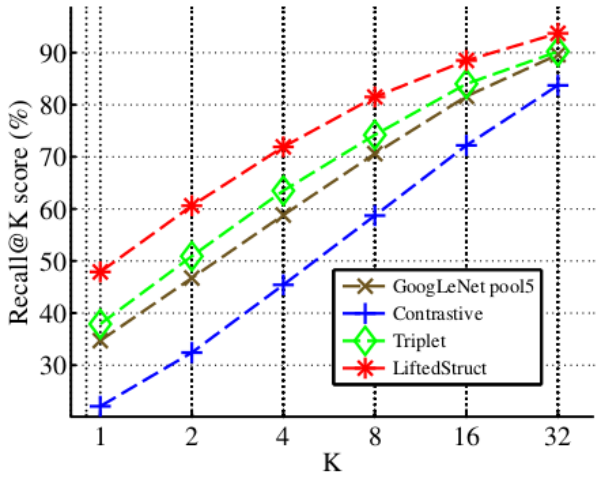
$\max(x, y)$



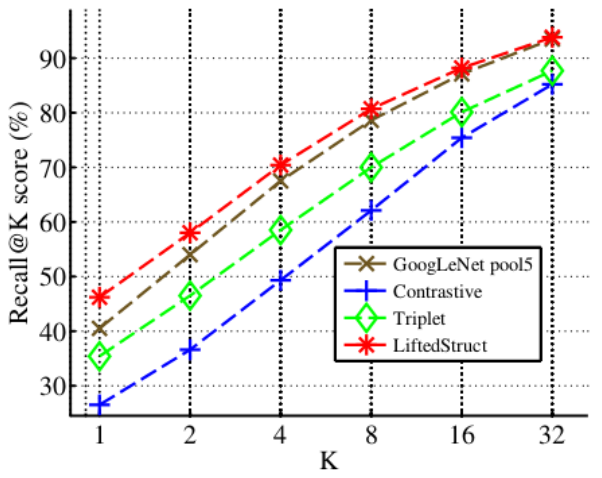
$\log(e^x + e^y)$

Does it make a difference ?

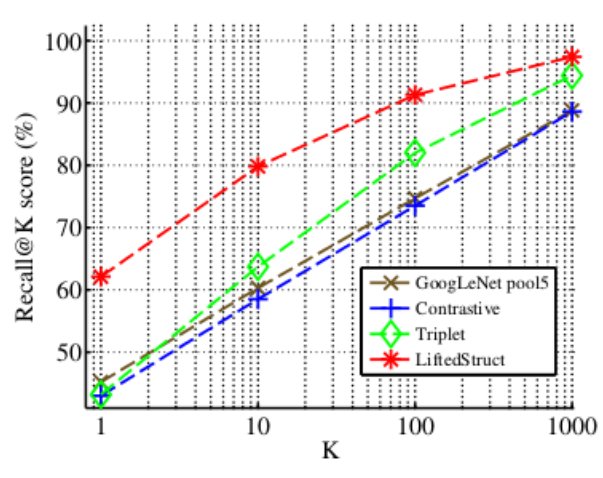
Image retrieval, Recall@K on three datasets



Cars-196

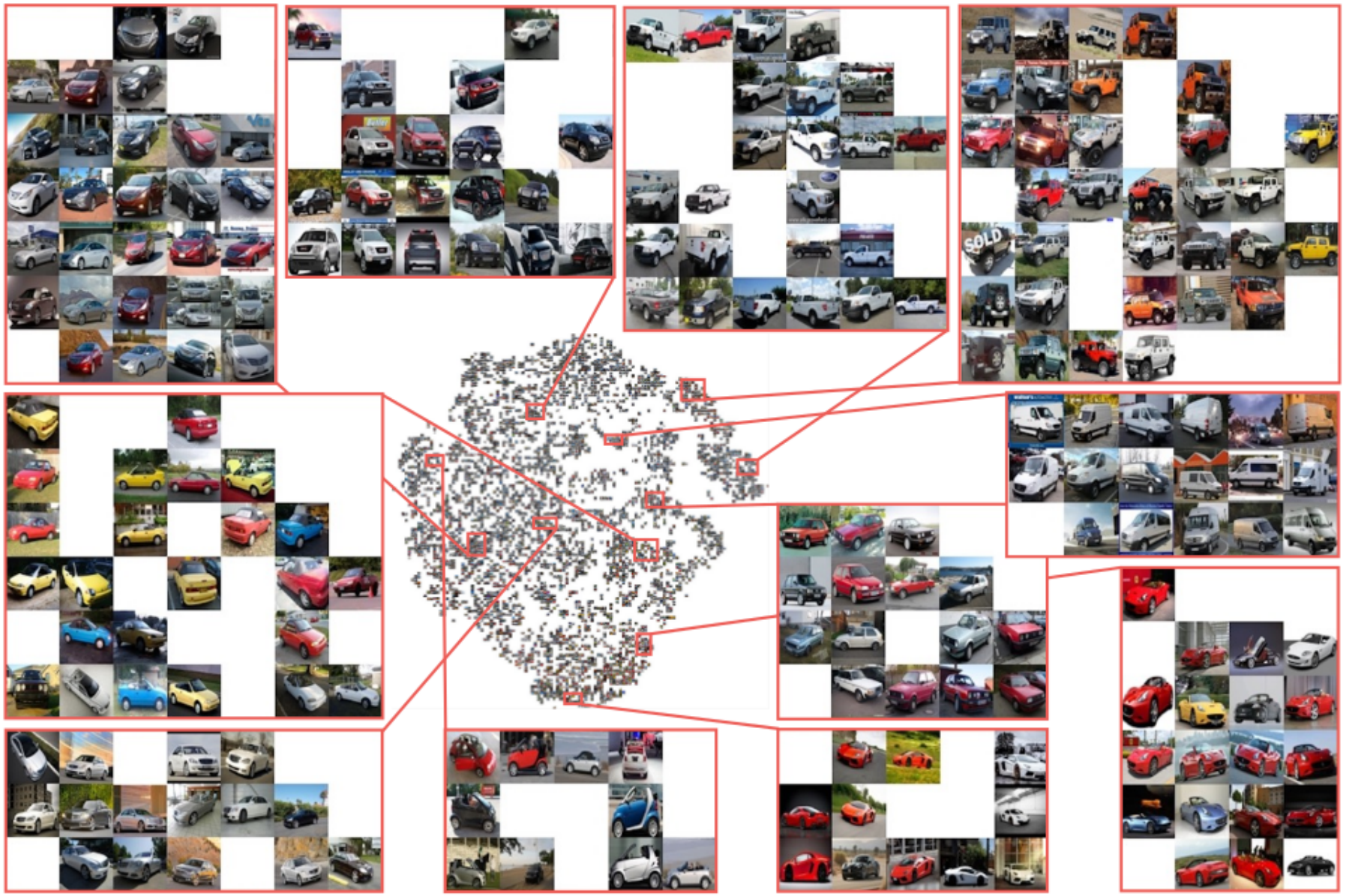


CUB-200



Stanford products

2d t-SNE of 64-d embedding for Cars-196



Some mining estrategies for triplets

In defense of the triplet loss for person re-identification. arXiv:1703.07737v4

Re-Id : given a query image, find its id (who is) as the one of the **closest image in the gallery**, made of at least one image per id.

Re-Id = **recognition with many classes**, one per id, and very few samples per class.

Dataset Market-1501, 750 ids train and **751 ids test**. Not so many? MS-Celeb-1M has **50K test identities**!

This papers compares different **triplet mining strategies for re-id** wrt one same dataset.

Standard triplet loss

A batch of size B is made of ***B* triplets** $\{ (a_i, p_i, n_i), i = 1 \dots B \}$ where a_i and p_i come from one same id and n from a different one. Since $B <$ number of ids, we can afford $id(a_i) \neq id(a_j), \forall\, i, j$.

$$L_{tri} = \sum_{i=1}^{B} \left[m + d(a_i, p_i) - d(a_i, n_i) \right]_+$$

B triplets $\rightarrow 3B$ images but B terms

Batch hard loss

A batch is made by sampling

- P ids without replacement
- K images per id

with $3B \approx PK$ images.

If x_j^i means the network output for j-th image of i-th id ,

$$L_{BH} = \sum_{i=1}^{P} \sum_{a=1}^{K} \left[m + \max_{p=1\ldots K} d(x_a^i, x_p^i) \; - \min_{j \neq i \,,\, n=1\ldots K} d(x_a^i, x_n^j) \right]_+$$

for all images as anchor, sum $\left[m + \text{ hardest positive} - \text{ hardest negative} \right]_+$

$3B$ images $\rightarrow$ $3B$ terms

Batch all loss

$$L_{BA} = \sum_{i=1}^{P} \sum_{a=1}^{K} \sum_{p=1, p\neq a}^{K} \sum_{j=1, j\neq i}^{P} \sum_{n=1}^{K} \left[m + d(x_a^i, x_p^i) - d(x_a^i, x_n^j) \right]_+$$

for all images as anchor, for all positives, for all negatives, sum $\left[m + d(a, p) - d(a, n) \right]_+$

$3B$ images $\rightarrow PK(PK - K)(K - 1) = 6B^2 - 4B$ terms

Batch all non-zero loss

$L_{BH\neq 0}$ same as before but sum only the non-zero terms $[m + \ldots]_+$

Different because **all losses are averaged later**.

Lifted embedding loss for triplets

$$L_L = \sum_{i=1}^{B} \Big[d(a_i, p_i) + \log \sum_{n \neq a_i, n \neq p_i} \left(e^{m - d(a_i, n)} + e^{m - d(p_i, n)} \right) \Big]_+$$

Similar to batch all only that for a batch size B,

- B pairs (anchor, positive)
- negatives are all but the anchor and positive ($3B - 2$)
- replace hard margin by smooth upper bound

Lifted embedding loss for triplets

$$L_{LG} = \sum_{i=1}^{P}\sum_{a=1}^{K}\Big[\log \sum_{p=1,p\neq a}^{K} e^{d(x_a^i, x_p^i)} + \log \sum_{i=1,j\neq i}^{P}\sum_{n=1}^{K} e^{m-d(x_a^i, x_n^j)}\Big]$$

all anchors all positives, all negatives

Do they make a difference ?

	margin 0.1		margin 0.2		margin 0.5		margin 1.0		soft margin	
	mAP	rank-1	mAP	rank-1	mAP	rank-1	mAP	rank-1	mAP	rank-1
Triplet ($\mathcal{L}_{\text{tri}}$)	40.80	59.23	41.71	60.78	43.51	60.87	43.61	61.63	*48.40*	66.37
Triplet ($\mathcal{L}_{\text{tri}}$) + OHM	16.6*	36.6*	*61.40*	82.95	32.0*	57.1*	41.45	59.42	46.63	65.43
Batch hard ($\mathcal{L}_{\text{BH}}$)	**65.09**	83.51	**65.27**	84.55	**65.12**	83.39	63.78	82.48	***65.77***	84.69
Batch hard ($\mathcal{L}_{\text{BH}\neq 0}$)	63.10	83.04	*64.19*	83.42	63.71	82.29	**64.06**	84.50	-	-
Batch all ($\mathcal{L}_{\text{BA}}$)	59.43	79.24	60.48	79.99	60.30	79.52	*62.08*	80.55	61.04	80.65
Batch all ($\mathcal{L}_{\text{BA}\neq 0}$)	63.29	83.65	64.31	83.37	*64.41*	83.98	**64.06**	82.90	-	-
Lifted 3-pos. ($\mathcal{L}_{\text{LG}}$)	64.00	82.71	63.87	82.86	63.61	84.55	*64.02*	84.17	-	-
Lifted 1-pos. ($\mathcal{L}_{\text{L}}$) [32]	61.95	81.35	*63.68*	81.73	63.01	82.48	62.28	82.34	-	-

- all online mining losses are better (and faster!) than simple offline OHM
- vanilla triplet loss L_{tri} performs poorly
- batch hard consistently better than batch all, "washing out" of the few non-zero terms in BA confirmed by BA$\neq 0$
- improvement for a range of margins m
- no claims out of the re-id problem and market-1501 dataset

A Mining implementation

```
class HardNegativePairSelector(PairSelector):
    """
    Creates all possible positive pairs. For negative pairs, pairs with smallest distance
    are taken into consideration, matching the number of positive pairs.
    """
    def __init__(self, cpu=True):
        super(HardNegativePairSelector, self).__init__()
        self.cpu = cpu

    def get_pairs(self, embeddings, labels):
        if self.cpu:
            embeddings = embeddings.cpu()
        distance_matrix = pdist(embeddings)

        # like above, make positive_pairs and negative_pairs
        ...

        negative_distances = distance_matrix[negative_pairs[:, 0],
                                             negative_pairs[:, 1]]
        negative_distances = negative_distances.cpu().data.numpy()
        top_negatives = np.argpartition(negative_distances, len(positive_pairs))
                        [:len(positive_pairs)]
        top_negative_pairs = negative_pairs[torch.LongTensor(top_negatives)]

        return positive_pairs, top_negative_pairs
```

Note: again gpu → numpy → gpu to use `np.argpartition`

What's the best loss and mining ?

[1] **A Metric learning reality check. Kevin Musgrave, Serge Belongie, Ser-Nam Lim. Cornell Tech, Facebook AI. ECCV'20.**

- deep ML papers from past 4 years have claimed **great advances in accuracy**
- they take a closer look to **see if this is actually true**
- find **flaws in the experimental setups** = common malpractices
- probably not exclusive of deep metric learning, but more general
- **properly compare** a dozen of metric learning papers = loss + miner
- experiments show **improvements have been marginal at best** for image retrieval in 3 benchmark datasets

1. Unfair comparison

or tricks to improve accuracy over past works

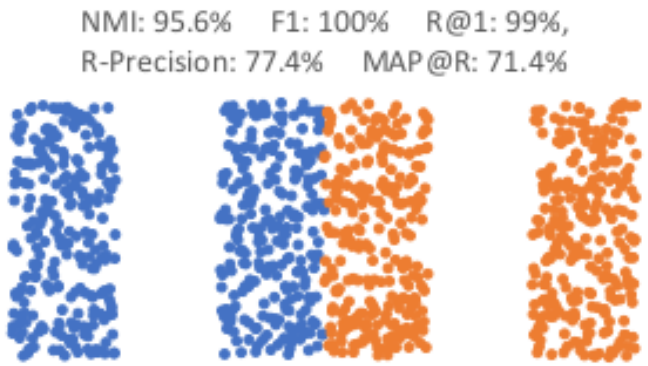
- use of **better architectures**, eg GoogleNet $\rightarrow$ ResNet50
- **increase the dimension** of the embedding space
- **better data augmentation** strategies, sometimes those reported $\neq$ from coded
- different **optimizers** and **learning rates**
- omission of **key details** with impact on the result (as read in code comments)
- **no confidence intervals** from several runs, only results of best trial

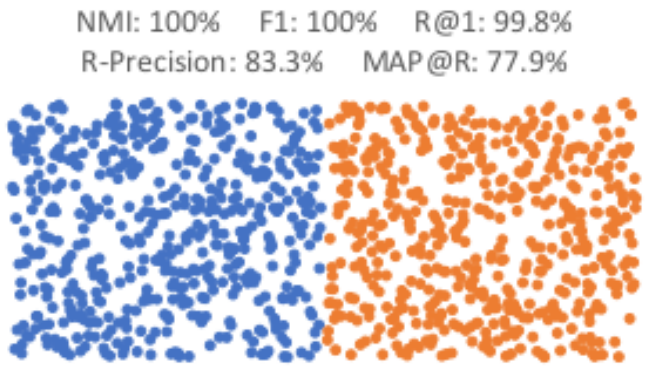
2. Training with test set feedback

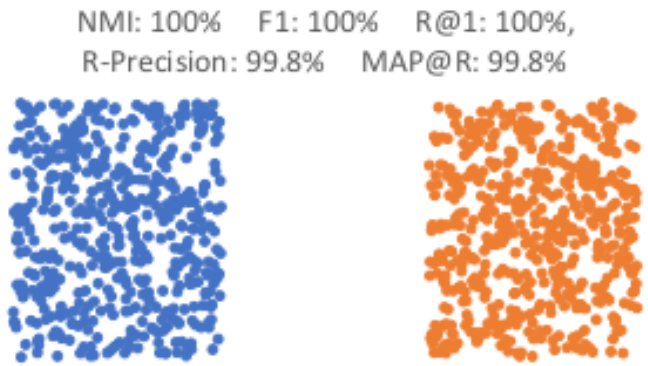
- at training, the test set accuracy of the model is checked at regular intervals
- and the best test set accuracy is reported
- no validation set : **model selection and hyperparameter tuning done with feedback from the test set** $\rightarrow$ overfitting

3. Weak IR accuracy metrics

- most ML papers report Recall@K, Normalized Mutual Information, F1 score
- on these three 2d-embeddings the three metrics report the **same values** despite the **varying degree of class clusterization**



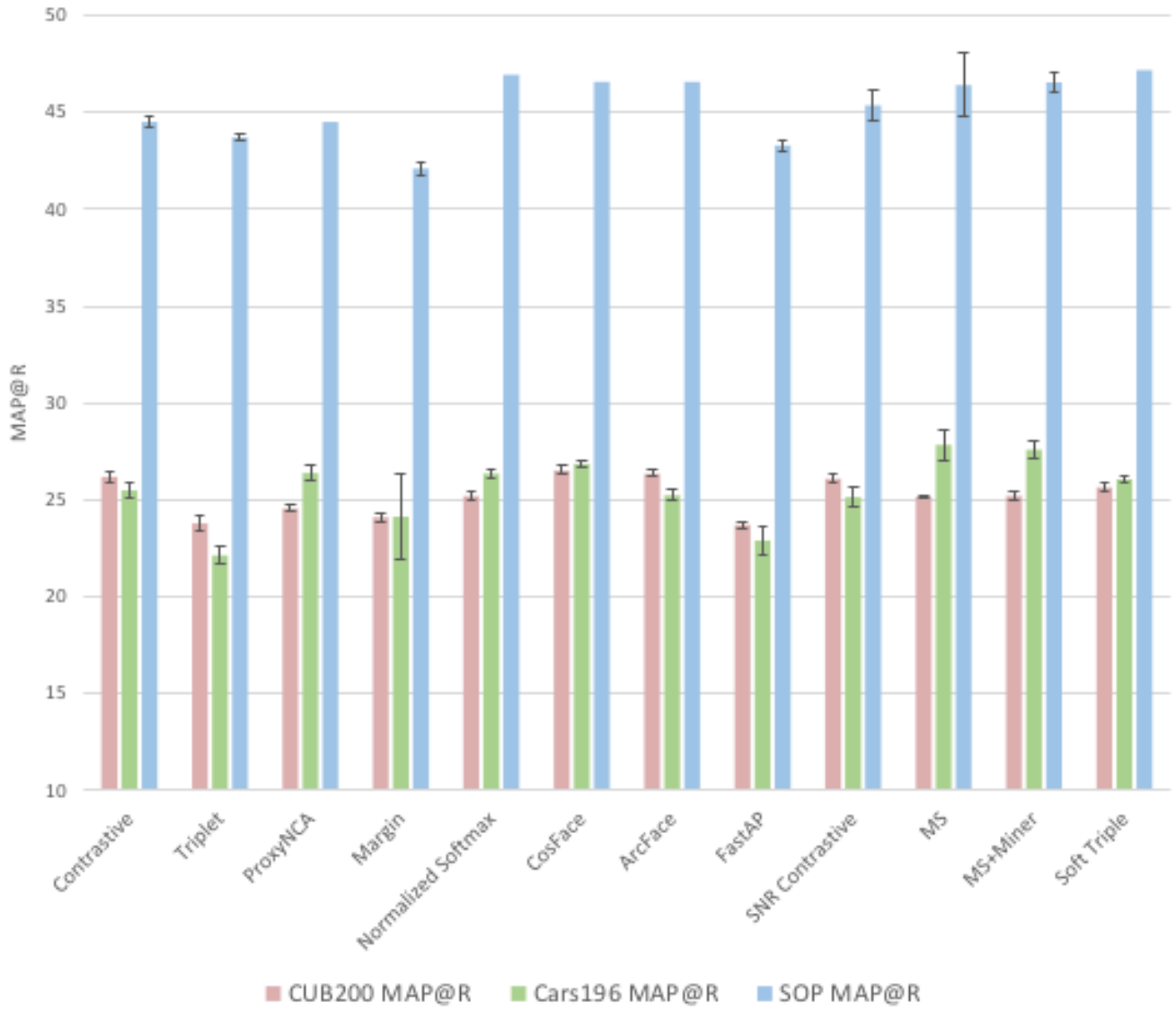




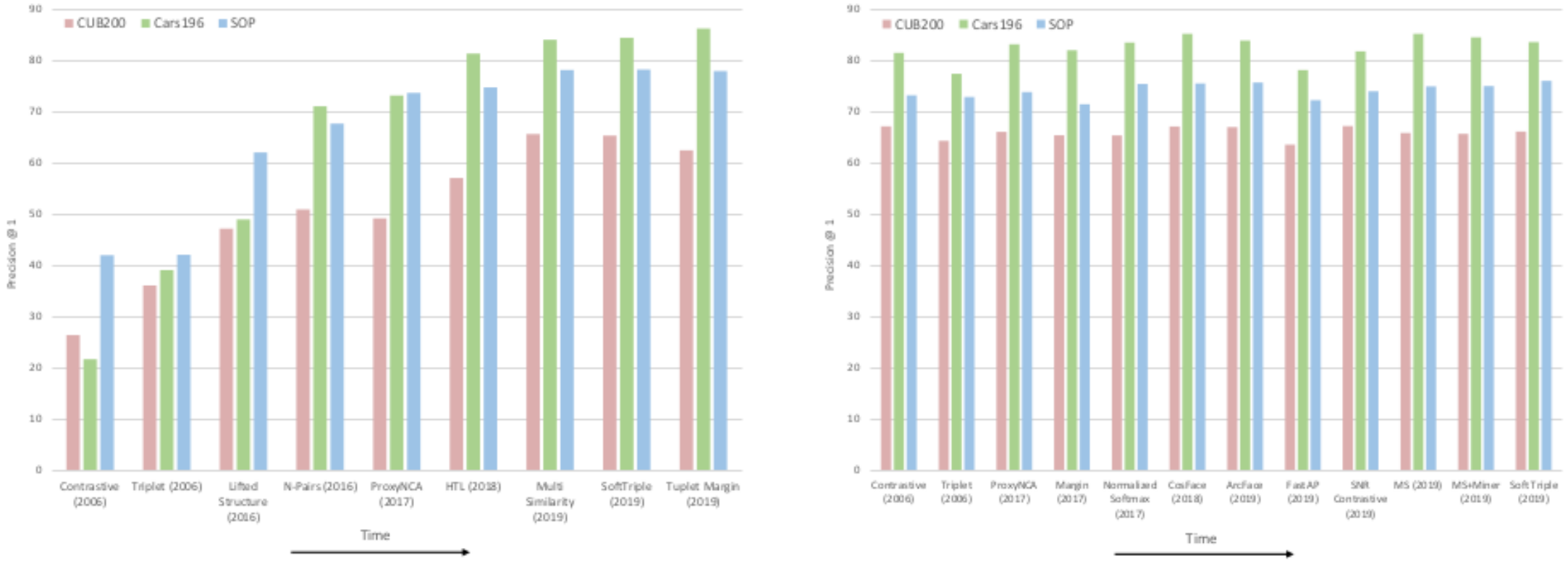
What do they do ?

- fix the architectures, embedding dimensions
- fix training process & hyperparameters : batch size, learning rate, data augmentation, optimizer, train / test class split, cross-validation etc. etc.
- new, better metric **MAP@R** = MAP of the R nearest neighbors
  - given a test sample, find the R nearest neighbors
  - count how many of them are of the same class as the sample
  - the more, the better
  - if closest neighbors are those of the same class, it's better
- also report Recall@1 = **Precision@1**
  - how many times the nearest sample is of the same class

MAP@R for 12 ML methods under same conditions, on 3 datasets : **no clear winner**



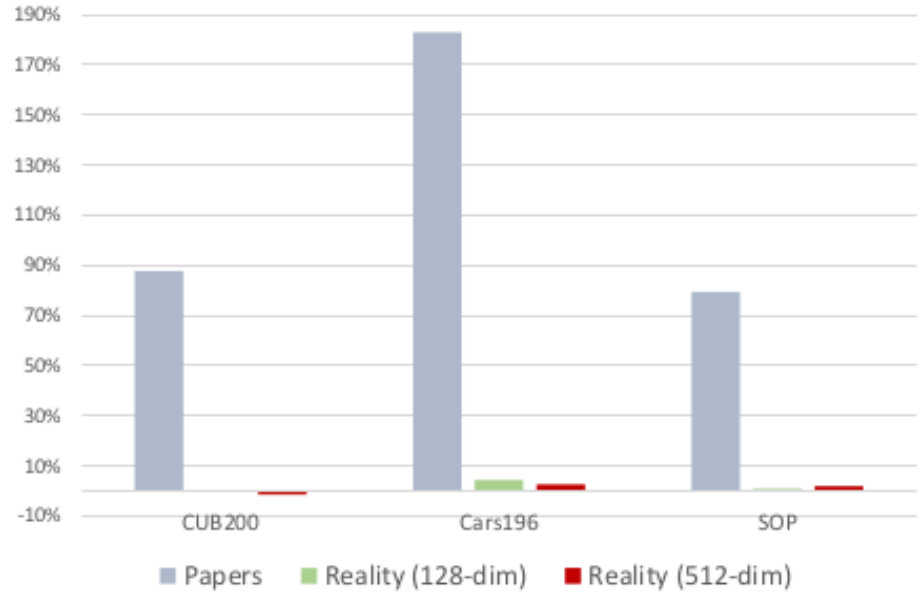
Same for **Precision@1**



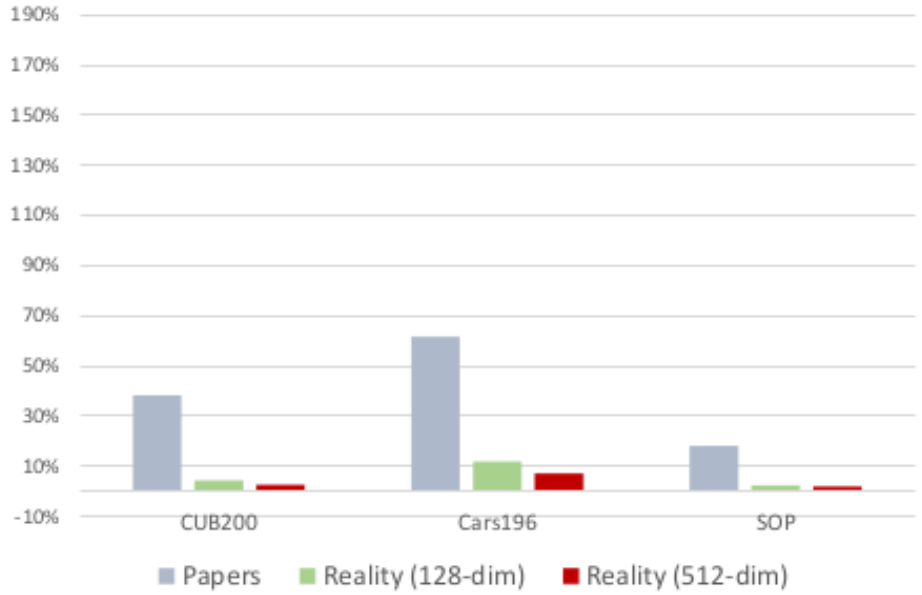
(a) The trend according to papers

(b) The trend according to reality

Fig. 4. Papers versus Reality: the trend of Precision@1 of various methods over the years.



(a) Relative improvement over the contrastive loss



(b) Relative improvement over the triplet loss

Fig. 5. Papers versus Reality: we look at the results tables of 20 metric learning papers from the years 2017-2019 (2 from 2017, 3 from 2018, 15 from 2019.) First, we note that 5 of the papers from 2019 did not include either the contrastive or triplet losses in their tables. Among the remaining 15 papers, 7 include the contrastive loss, and 12 include the triplet loss. For each paper, we compute the relative percentage improvement of their proposed method over the contrastive or triplet loss, and then take the average improvement across papers (grey bars in the above figures). The green and red bars are the average relative improvement that we obtain, in the separated 128-dim and concatenated 512-dim settings, respectively.

Conclusions

- Contrastive and triplet losses (+ mining) are strong baselines
- Authors just copied numbers of previous papers instead of trying to obtain a more reasonable baseline by implementing the losses themselves and try them on proposed networks etc.
- *"With good implementations of those baseline losses, a level playing field, and proper machine learning practices, (...) methods perform similarly to one another, whether they were introduced in 2006 or 2019"*
- but this is only for image retrieval, on 3 datasets

5. Embedding visualization

You have trained a network to learn an embedding, a space where similar samples are close and far from dissimilar samples.

But **how good is the embedding, *qualitatively*** ? How are the classes spread on it ?
You want to *visualize* the embedding = view your *test* samples on it.

Obvious problem: embedding is a high-dimensional space, $d \gg 3$ surely.

Solution: low-dimensional projection with PCA ?

PCA looks for the rotation that concentrates variance (variability) along a few axes and then you can select the 2 or 3 axes with the highest variance.

It turns out that high variance is not the best way to perform dimensionality reduction.
Best means to preserve relative distances when projecting

t-SNE

Classically (paper published in 2008), t-SNE has been a much used tool / algorithm for high-dimensional embedding projection, for visualization purposes *only*.

> t-SNE is capable of capturing much of the local structure of the high-dimensional data very well while also revealing global structure such as the presence of clusters at several scales.

Goal : given a set of points $x_i \in \mathbb{R}^D$, convert them to $y_i \in \mathbb{R}^d, d \ll D$ *keeping as much as possible the distance relationships in the high-dimensional space*

How does it work ? In a nutshell[1],

1. convert distances in high-dim space to probabilities

$$p_{j|i} = \frac{\exp\left(-||x_i - x_j||^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-||x_i - x_k||^2 / 2\sigma_i^2\right)}, \qquad p_{ij} = \frac{p_{i|j} + p_{j|i}}{2N} \qquad (1)$$

   $p_{j|i}$ means the probability that x_i would choose x_j as its neighbor if neighbors were picked in proportion to their probability under a Gaussian centered at x_i

2. the value of σ_i is the dispersion of neighbors and approximated by

$$\text{Perplexity} = 2^{-\sum_{j} p_{j|i} \log_2 p_{j|i}} \qquad (2)$$

[1] How exactly UMAP works and why exactly it is better than t-SNE by N. Oskolkov

3. $q_{j|i}$, y_i, y_j are the low-dimensional counterparts of $p_{j|i}$, x_i, x_j and define $q_{j|i}$ as

$$q_{ij} = \frac{\exp\left(-\|y_i - y_j\|^2\right)}{\sum_{k \neq l} \exp\left(-\|y_k - y_l\|^2\right)}$$

but this has problems on crowded regions so authors change it by another distribution (no exponentials :))

$$q_{ij} = \frac{\left(1 + ||y_i - y_j||^2\right)^{-1}}{\sum_{k \neq l} \left(1 + ||y_k - y_l||^2\right)^{-1}} \qquad (3)$$

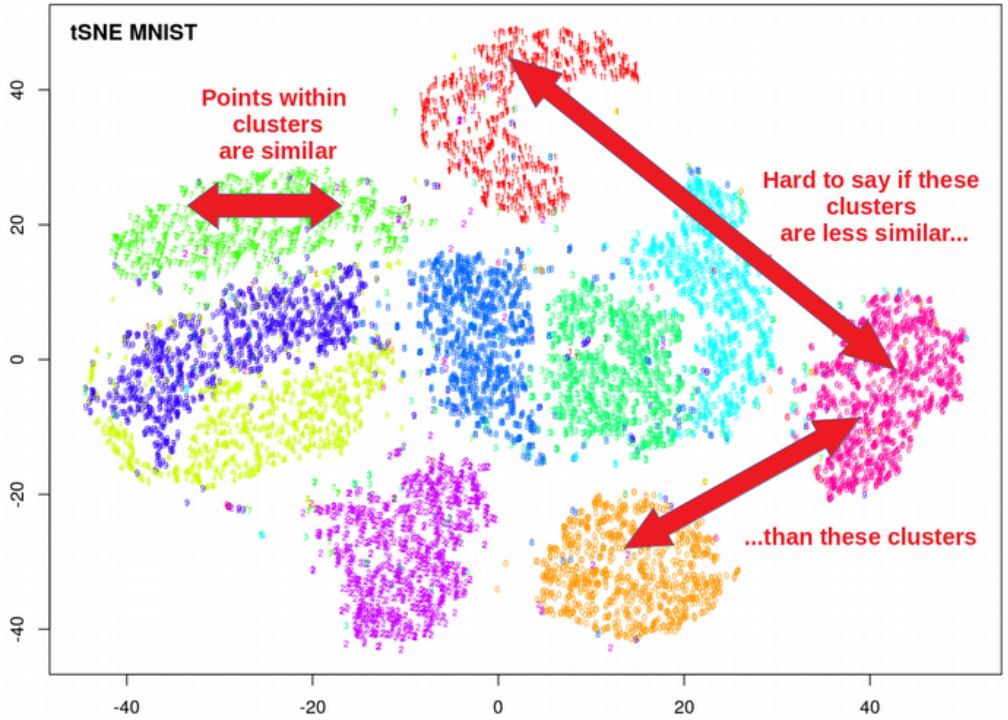
4. If the y_i, y_j model well the distances of x_i, x_j $\forall i, j$ in the high-dimensional space then $p_{j|i}$ and $q_{j|i}$ should be *similar*.

$$KL(P_i||Q_i) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}, \qquad \frac{\partial KL}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)\left(1 + ||y_i - y_j||^2\right)^{-1} \qquad (4)$$

KL = Kullback-Leibler *divergence* $\neq$ distance, between 2 distributions, not simmetric. But can be minimized iteratively to find the y_i's

UMAP

t-SNE has been superseded by UMAP. One key difference: t-SNE keeps local structure well but not so much global structure



Read the post *How exactly...*. Apparently different, it bears several similarities with it. Complete explanation is out of scope. By the same author, read this post : How to program UMAP from scratch

How to run t-SNE and UMAP ?

- Scikit-learn has t-SNE + usage examples (and PCA)
- install U-MAP with `conda` or `pip`, read the docs
- both already included in Tensorboard
- **simplest** but slow : upload your data to https://projector.tensorflow.org/

Cool demos

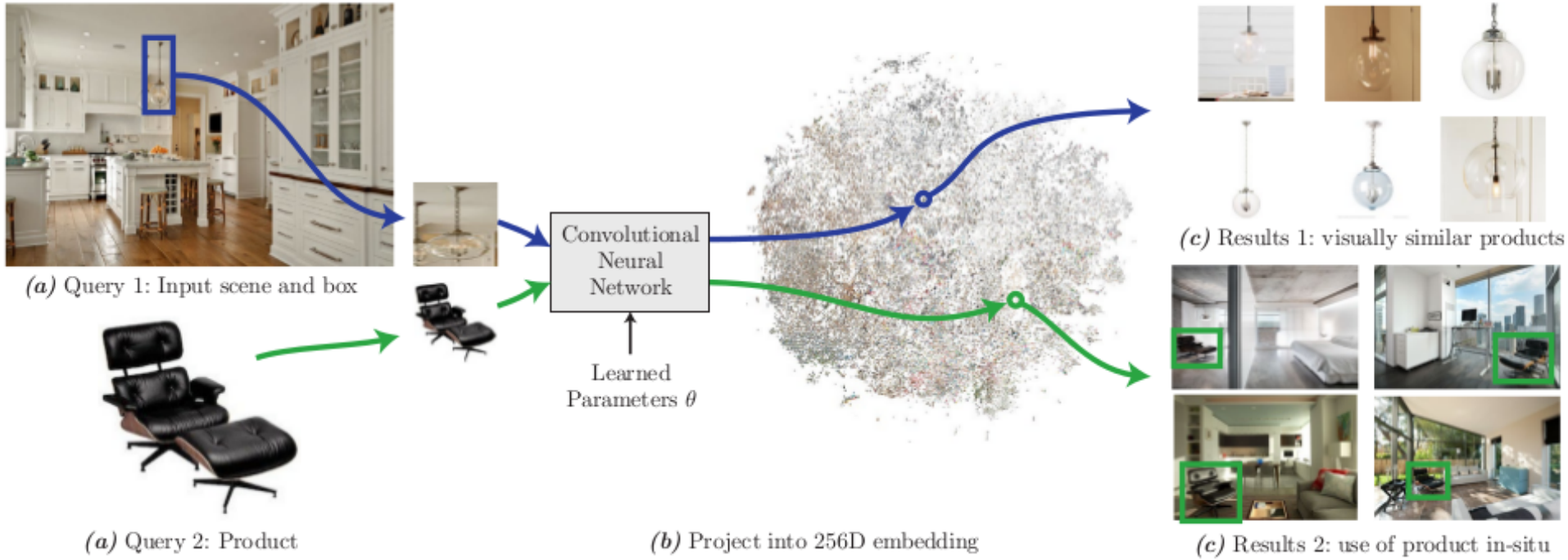
From the UMAP documentation you can interactively explore already computed embeddings of

- MNIST
- Fashion MNIST

6. Applications

Image retrieval

Learning visual similarity for product design with convolutional neural networks. Sean Bell, Kavita Bala (U. Cornell). ACM Trans. on Graphics (SIGGRAPH'15).



(a) Query 1: Input scene and box

(a) Query 2: Product

(b) Project into 256D embedding

(c) Results 1: visually similar products

(c) Results 2: use of product in-situ

Interest

- retrieval $\neq$ classification
- two image domains
- compares learning features with metric learning and with classifier learning
- contrastive loss combined with cross-entropy
- nice visualiation of the learned embedding

Goal

- learn an embedding for visual search of products in **interior design**
- **identify products on catalogs** from pictures of scenes or catalogs: *where can I find this ?*, *find me chairs similar in style to this one*
- search for **matching products** : *find me products combining with this armchair*
- learn an embedding (distance) for images from **two domains**: crops from photos of scenes and product catalogs
- **difficult** because in-situ object images have varying backgrounds, scale, lighting, size, orientation . . . compared to catalogs

Dataset

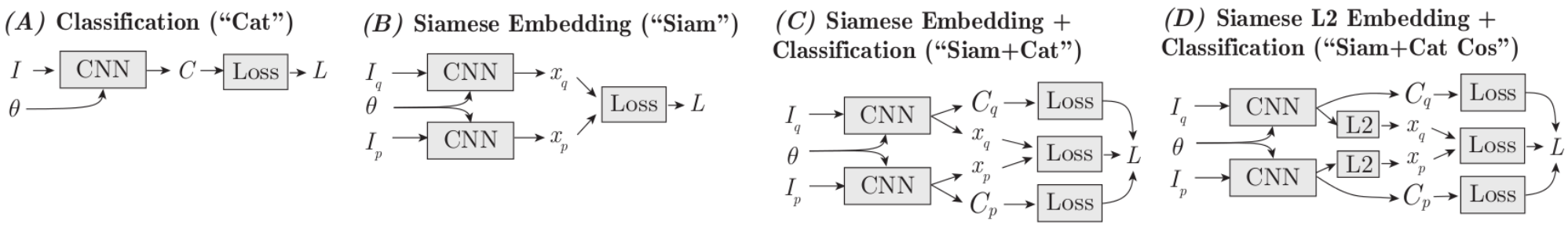
- 7M product photos and 6.5M room photos from Houzz.com

- duplicate and near-duplicate detection by clustering fc7 layer of AlexNet $\rightarrow$ 3.4M product and 6.1M room images
- out of 3.4M, only 178,712 product photos are product tags
- Mechanical Turk with quality control $\rightarrow$ final **102K pairs (in-situ bounding box, product image)** for $2500, $1.5/hour
- for each of the 102 K positive pairs, 20/80 random images of products different/same category $\rightarrow$ **64M training pairs**
- from **unseen** 6400 room pictures, **10K test pairs**

Retrieval task

- it's not about classification $=$ given a test image find the most similar images **in the training set**
- train with similar/dissimilar pairs to learn a distance
- at test time we have **new pairs of images**, possibly of **different classes**
- for each pair, one image is the **query** and the other the **target** or tagged
- for every query ask for k most similar images and count how many times the target is in the answer, on average : **mean recall @ k**

Method



CNN = GoogLeNet, AlexNet

A : product category classification and then features for embedding

B : Siamese, plain contrastive loss

C : category prediction + contrastive loss

D : same as C with L2 normalization, **best**

Evaluation

On a test set of 10K pairs

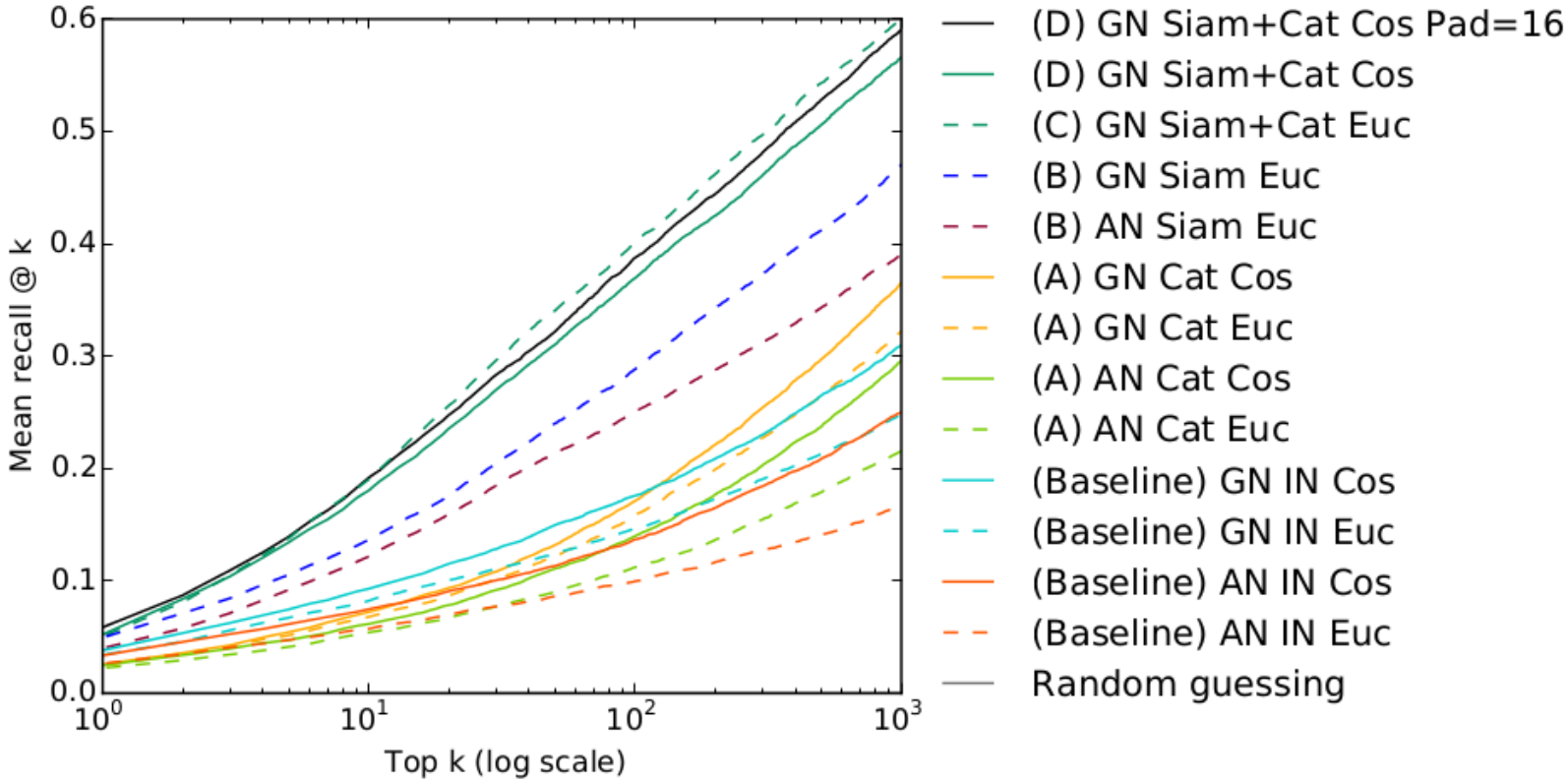


Figure 12: Quantitative evaluation (log scale). *Recall (whether or not the single tagged item was returned) as a function of the number of items returned (k). Recall for each query is either 0 or 1, and is averaged across 10,000 items. "GN": GoogLeNet, "AN": AlexNet,*

Random queries

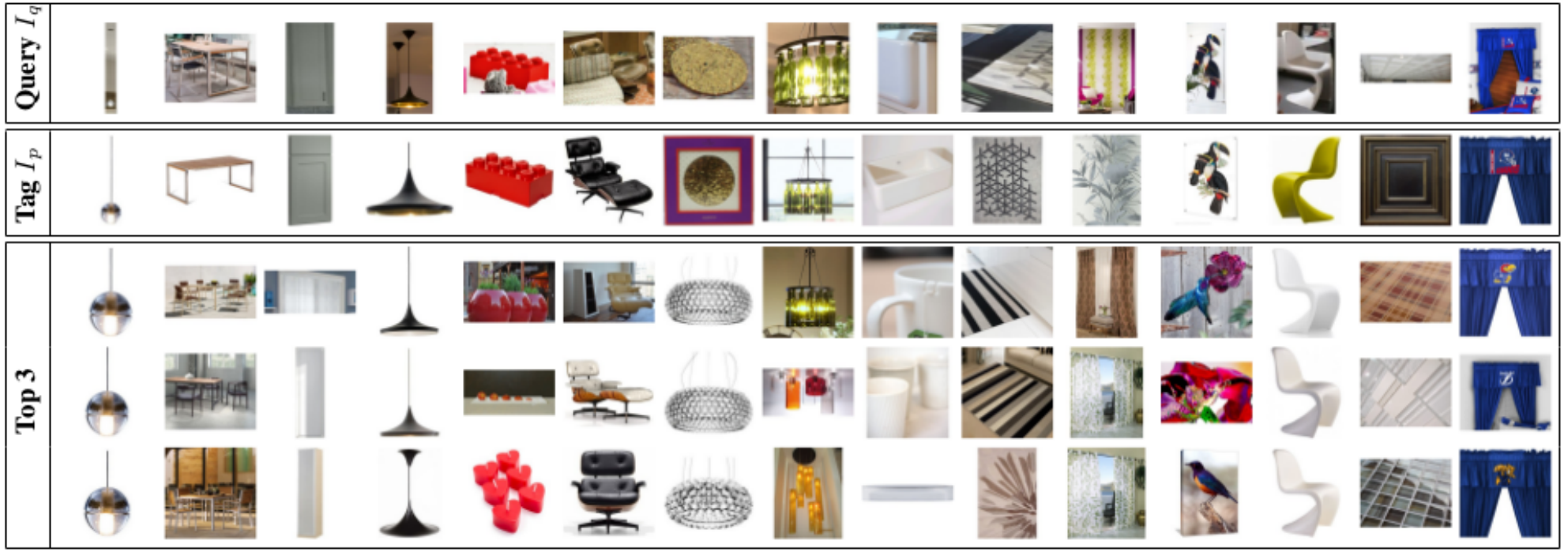
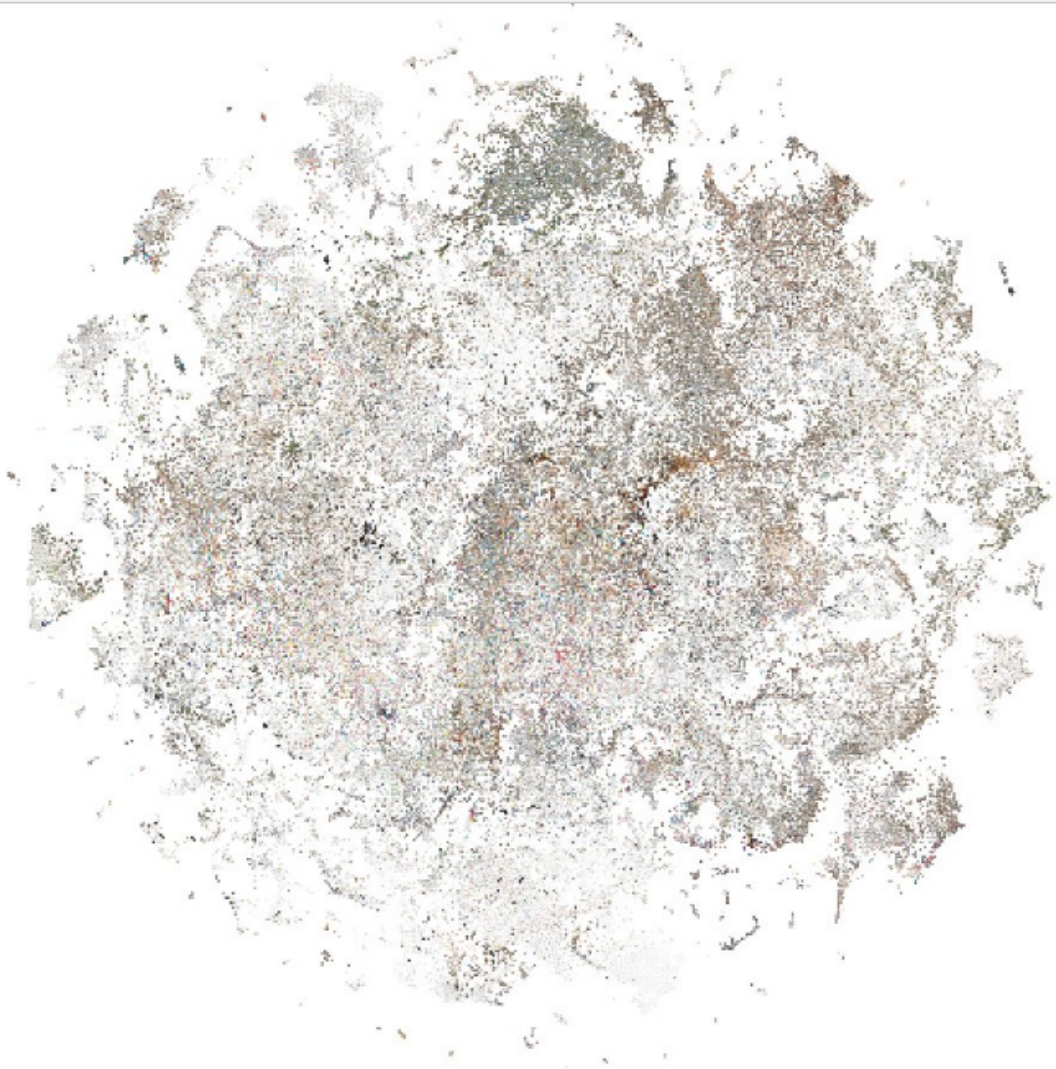


Figure 11: Product search: uncurated random queries from the test set. *For each query I_q, we show the top 3 retrievals using our method as well as the tagged canonical image I_p from Houzz.com. Object categories are not known at test time. Note that sometimes the retrieved results are closer to the query than I_p.*

Embedding

256 dimensions projected to 2D with t-SNE



Verification and clustering

Facenet: a unified embedding for face recognition and clustering. Florian Schroff, Dmitry Kalenichenko, James Phiblin (Google). 2015.

Interest

- large scale learning, HUGE amounts of data
- landmark in face verification
- triplet loss with negatives batch hard mining
- results

Goal

- verification : given two pictures, predict whether the two images are from the same person or not
- check effect of large scale : train with millions, 10s and 100s millions of images

Method

- Inception type network
- triplet loss with batch **semihard** negative mining : "*selecting the hardest negatives can lead to bad local minima early on training*"

$$d(a_i, p_i) < d(a_i, n_i) < m$$

- embedding dimension 128
- batches of 1800 triplets
- trained during 1000 - 2000 hours = 2.7 **months**

Method

- train with undisclosed dataset of **200 million images** of **8 million identities**
- then take another dataset (personal, LFW), split it in 10 folds
- from first 9 folds, learn the threshold *on the distance* for same/different id classification
- test with the last fold, repeat for every fold

"*Compared to only millions images, tens of millions of exemplars result in a clear boost of accuracy (60% less error). Hundreds of millions still gives a small boost but the improvement tappers off.*"

Results

- Labeled Faces in the Wild (2009) is a benchmark dataset for face recognition and verification
- 13K images, 1680 identities with 2+ images (but MS-Celeb-1M has 8.4M images of 100K identities)
- **99.63%** verification accuracy, 3 times less error than best previous work

The **only** errors

Re-Id

Deep Relative Distance Learning: Tell the Difference Between Similar Vehicles. Hongye Liu, Yonghong Tian, Yaowei Wang *et al* (Beijing U.). CVPR'16.

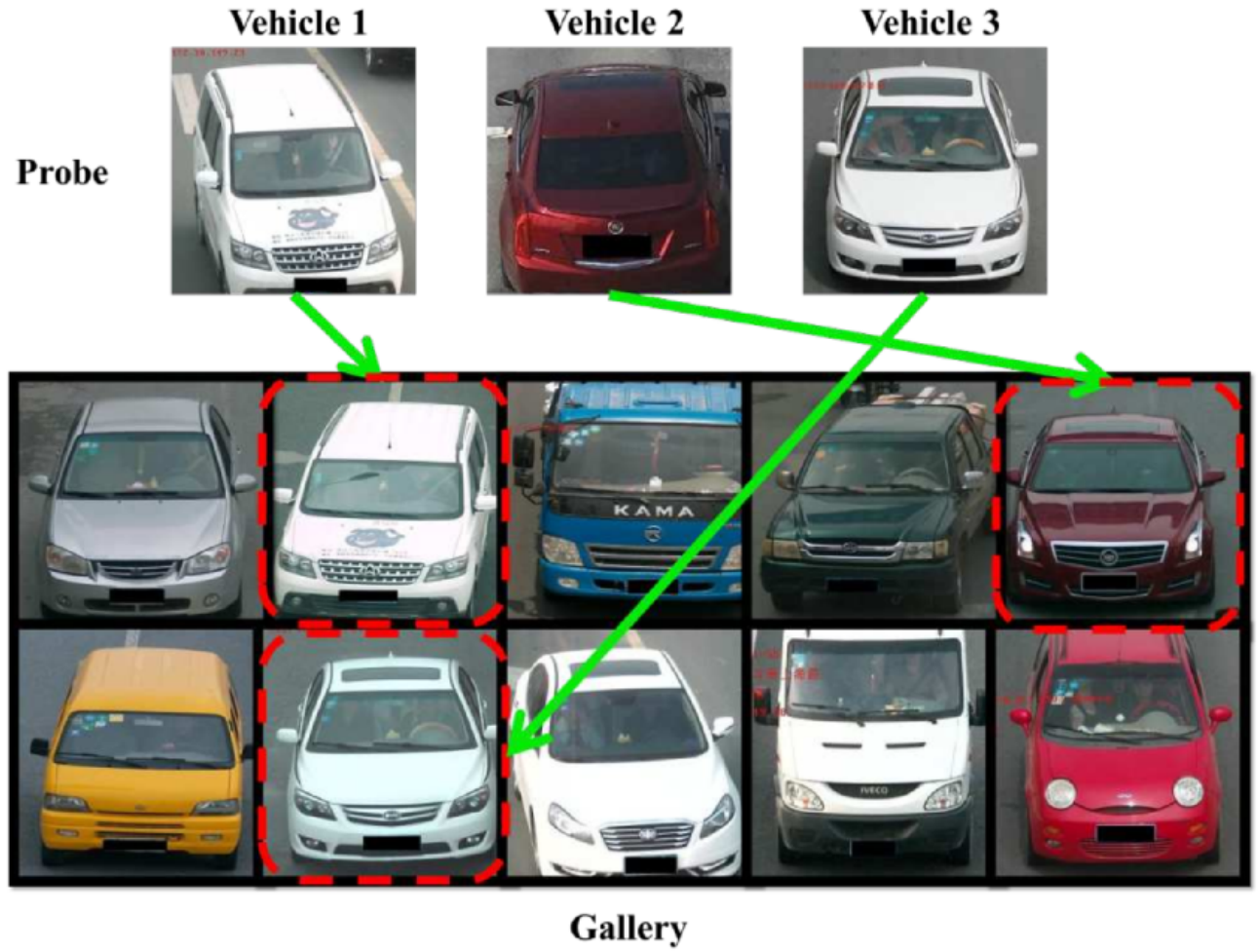
Interest

- (almost) new task
- adapted triplet loss
- adapted Siamese architecture
- dataset

Person re-identification

- In video surveillance, when being presented with a person-of interest (query), person re-ID tells **who is**: returns **images of this same person** previously observed in another place/time/camera , **or tells it's unknown**
- retrieve image(s) of the same person from a **gallery**
- challenging : match two images of the same person under **intensive appearance changes** such as lighting, person pose, camera and viewpoint
- **in-between of classification and retrieval** : in re-id training and testing identities may not overlap (but in the vast majority of papers they do) $\rightarrow$ closed world re-id $\approx$ classification
- however, very **few samples/class** and **many ids**

Vehicle re-identification



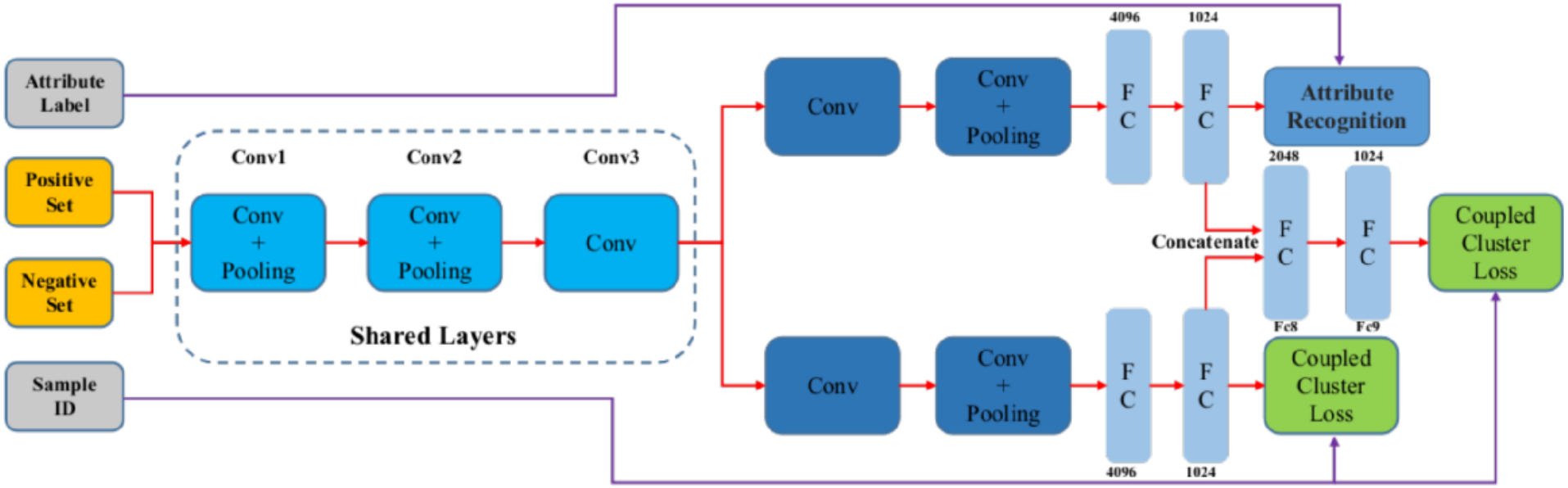
Vehicle re-identification

Differently from person re-id, two different vehicles of same maker-model-year could be equal, but in practice they normally are not because of **characteristic details**.

Special marks which can be used for identification task

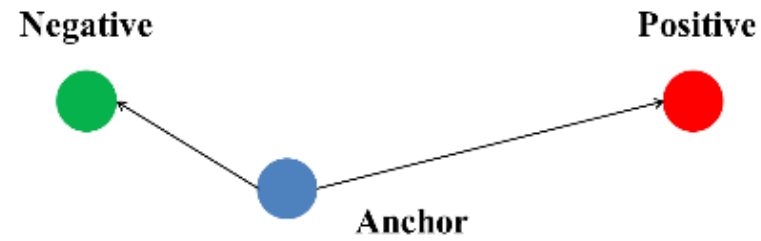
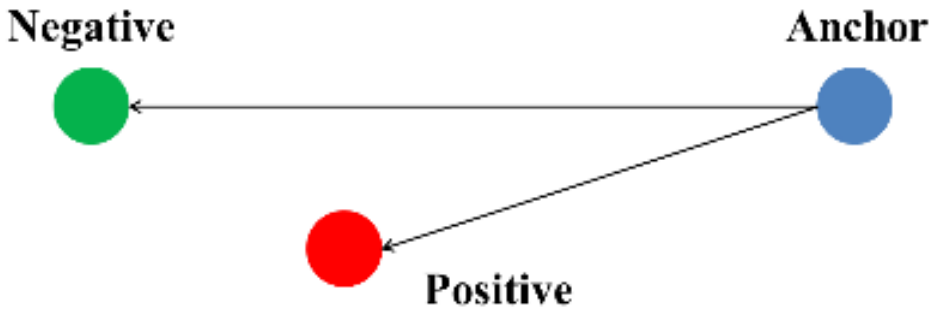
Mehod : architecture

- two vehicles have definitely different id if from different models
- if not, then take into account the characteristic details
- special Siamese network : one branch for same/different model detection, another branch for same/different id



attribute = maker + model + year, "Audi A4L 2009"

Mehod : modified triplet loss

	$a \longleftrightarrow p$	
with standard triplet loss, loss $>$ 0		loss $=$ 0 but p, n **alineated**

How to deal with wrong anchor choices ? For each id, replace their anchors by an *estimation* of the class center.

Mehod : modified triplet loss

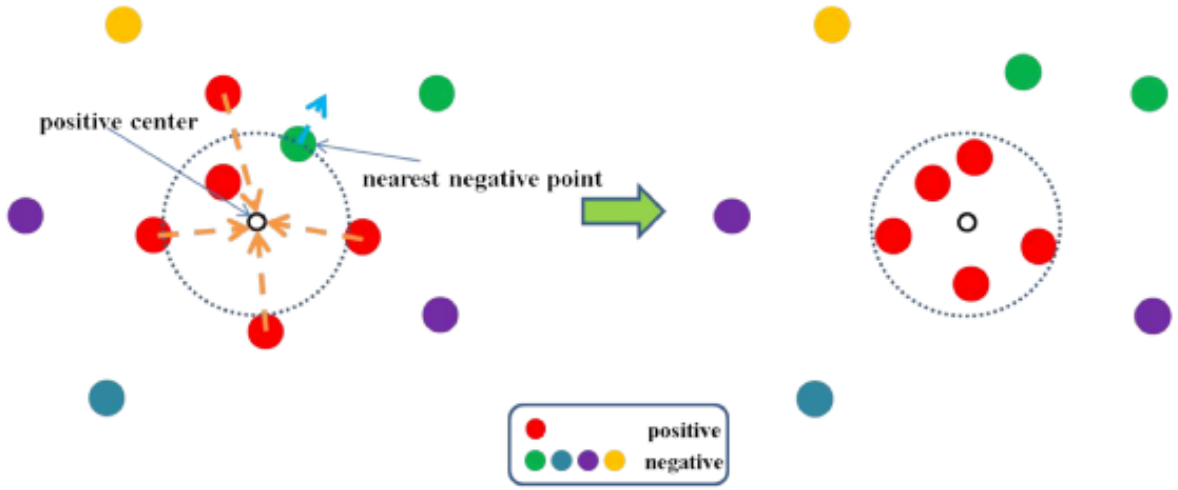
$X^p = \{x_1^p \dots x_{N^p}^p\}$ samples of **one** same id

$X^n = \{x_1^n \dots x_{N^n}^n\}$ samples of other ids

$c^p = \frac{1}{N^p} \sum_i x_i^p$ estimated id center at present

$x_\star^n = \arg\min_j ||c^p - x_j^n||_2$ hardest negative, nearest to c^p

$L(X^p, X^n) = \max\{ 0,\ m + d(x_i^p, c^p) - d(x_\star^n, c^p) \}$



Coupled clusters loss.

VehicleID dataset

- 220K images of 26K vehicles, 8.4 images/vehicle
- both frontal and rear views
- multiple cameras

Results

- 50/50 train/test
- test samples further divided into 3 test splits, small/medium/large with **800/1600/2400 query** vehicle images (1 per id), to ressemble person re-id datasets
- recall at top-5 : **0.73, 0.67, 0.62** small/medium/large test split
- recall at top-1 : **0.49, 0.43, 0.38**

Few-shot learning

Prototypical networks for few-shot learning. Jake Snell, Kevin Swersky, Richard S. Zemel (U. Toronto, Twitter). arXiv:1703.05175v2.

Interest

- new difficult task
- realistic, open recognition
- new training algorithm, *episodes*

What's few-shot learning

- a classifier must generalize to new classes not included in the training set
- given only a small number of examples for each new class
- can't retrain the model with the new data because it would severely overfit the new classes
- but humans can recognize new classes, even from one exemplar (one-shot)

Method

- learn an embedding in which samples from one class cluster around a single prototype representing that class
- this will happen also for new classes to come
- prototype is the mean
- classification is performed by nearest prototype = nearest class mean
- the network computing the embedding is trained trying to mimic the arrival of new few-shot classes
- so that at test time it will perform better

$S = \{(x_1, y_1) \dots (x_N, y_N)\}, x_i \in R^D, y_i \in \{1 \dots K\}$, dataset of N samples from K classes

S_k, samples of class k

$f_\phi(x)$, embedding of sample x when network has parameters ϕ

$c_k = \frac{1}{|S_k|} \sum_{x_i \in S_k} f_\phi(x_i)$, prototype of class k

$p_\phi(y = k|x) = \frac{e^{-d(f_\phi(x), c_k)}}{\sum_{l=1}^{K} e^{-d(f_\phi(x), c_l)}}$, distance $\rightarrow$ probability

$J(\phi) = -\log p_\phi(y = k|x)$ with k true class, loss function

Algorithm 1 Training episode loss computation for prototypical networks. N is the number of examples in the training set, K is the number of classes in the training set, $N_C \leq K$ is the number of classes per episode, N_S is the number of support examples per class, N_Q is the number of query examples per class. RANDOMSAMPLE(S, N) denotes a set of N elements chosen uniformly at random from set S, without replacement.

Input: Training set $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$, where each $y_i \in \{1, \ldots, K\}$. $\mathcal{D}_k$ denotes the subset of $\mathcal{D}$ containing all elements $(\mathbf{x}_i, y_i)$ such that $y_i = k$.

Output: The loss J for a randomly generated training episode.

$V \leftarrow$ RANDOMSAMPLE$(\{1, \ldots, K\}, N_C)$ ▷ Select class indices for episode

for k in $\{1, \ldots, N_C\}$ **do**

$S_k \leftarrow$ RANDOMSAMPLE$(\mathcal{D}_{V_k}, N_S)$ ▷ Select support examples

$Q_k \leftarrow$ RANDOMSAMPLE$(\mathcal{D}_{V_k} \setminus S_k, N_Q)$ ▷ Select query examples

$$\mathbf{c}_k \leftarrow \frac{1}{N_C} \sum_{(\mathbf{x}_i, y_i) \in S_k} f_{\boldsymbol{\phi}}(\mathbf{x}_i)$$ ▷ Compute prototype from support examples

end for

$J \leftarrow 0$ ▷ Initialize loss

for k in $\{1, \ldots, N_C\}$ **do**

for $(\mathbf{x}, y)$ in Q_k **do**

$$J \leftarrow J + \frac{1}{N_C N_Q} \left[d(f_{\boldsymbol{\phi}}(\mathbf{x}), \mathbf{c}_k)) + \log \sum_{k'} \exp(-d(f_{\boldsymbol{\phi}}(\mathbf{x}), \mathbf{c}_k)) \right]$$ ▷ Update loss

end for

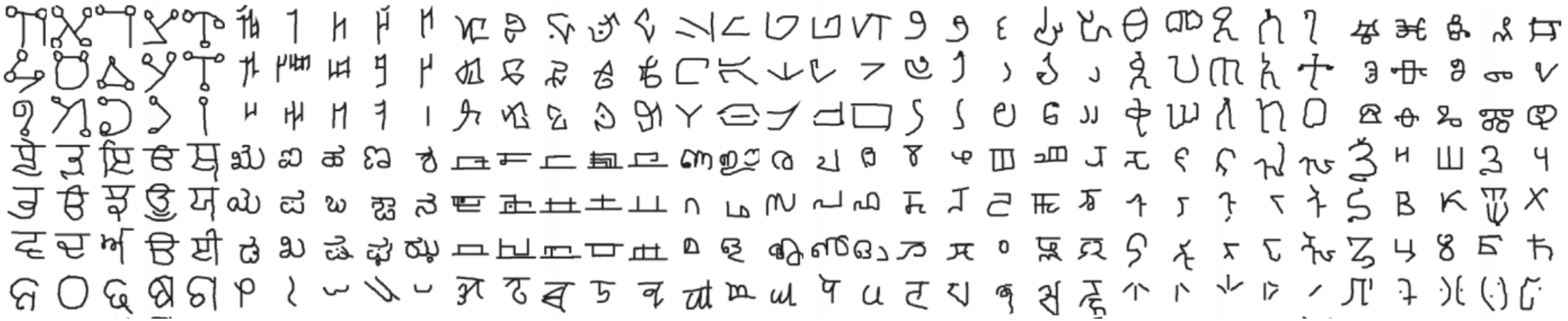
end for

Want samples of Q_k to be close to $\mathbf{c}_k$, computed with samples S_k

Each time only a subset of N_C classes is used (indices in V)

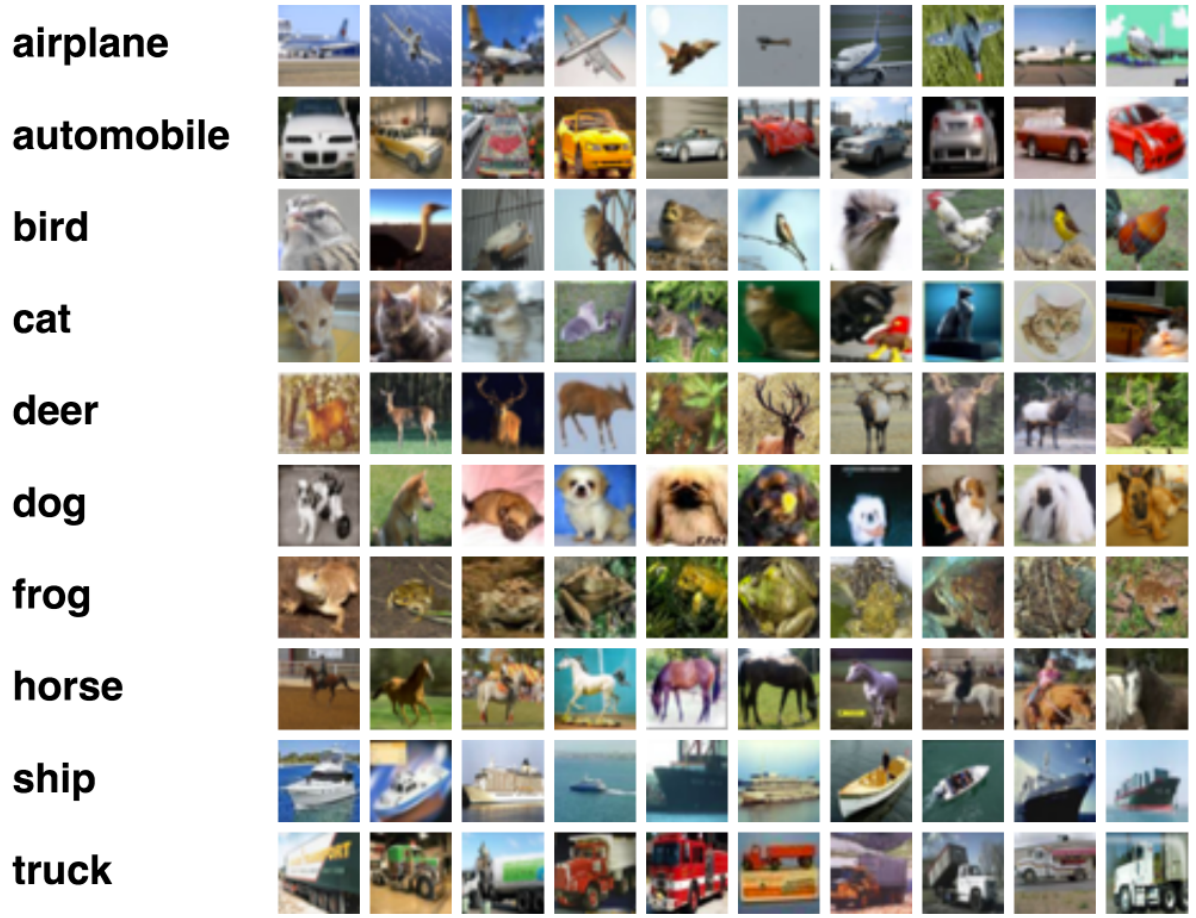
Results

- Omniglot dataset : 1623 handwritten characters from 50 alphabets.



- 1200 train, 463 test
- 4 blocks of conv $3 \times 3 \times 64$ + BN + ReLU + maxpool $2 \times 2 \rightarrow$ for 28×28 images, 64 dimensions
- 2000 episodes of $N_C = 60$ classes, $N_Q = 5$ query points per class
- 1-shot 98.8%, 5-shot 99.7% (+0.7%, +3% than previous works)

- Tiny ImageNet dataset : 60K images 84 $\times$ 84 of 100 classes, 600 images/class



- 6 splits of 100 classes, 64/16/20 train/validation/test
- 1-shot 49.42%, 5-shot 68.2% (+6% and +8% than previous works)

Patch correspondence

Learning to compare image patches via convolutional neural networks. Sergey Zagoruyko, Nikos Komodakis. CVPR'15.

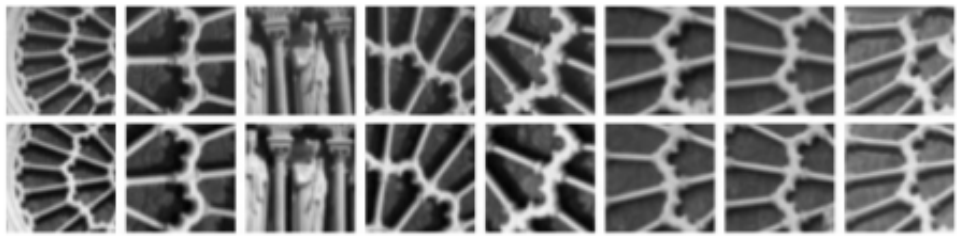
Interest

- again, task very well suited to metric learning
- seminal work
- tries several Siamese architectures
- at the basis of stereo reconstruction, but stereo papers involve more steps too long to explain

Goal

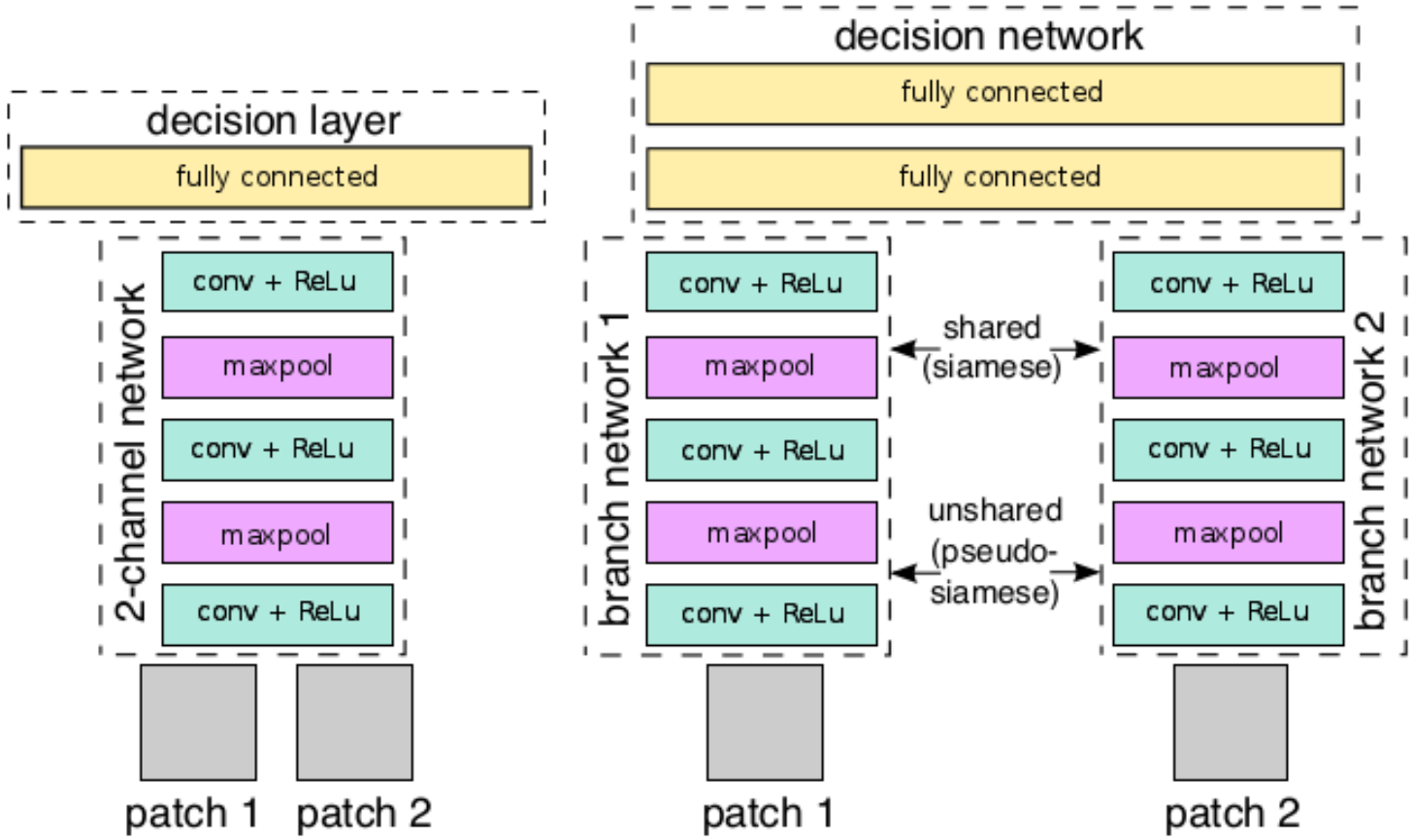
- learn directly from data a general **similarity function to compare image patches**, instead of handcrafted descriptors + euclidean distance
- use it to decide if **two patches are corresponding** to each other or not
- despite changes in **viewpoint, illumination, occlusions, shading, camera** etc.
- why : basis of dense **stereo, structure from motion**, building **panoramas**, **optical flow**...
- investigate what network **architecture** is the best

Dataset : Notre Dame, 3d reconstructions

	
positive pairs	negative pairs

Method

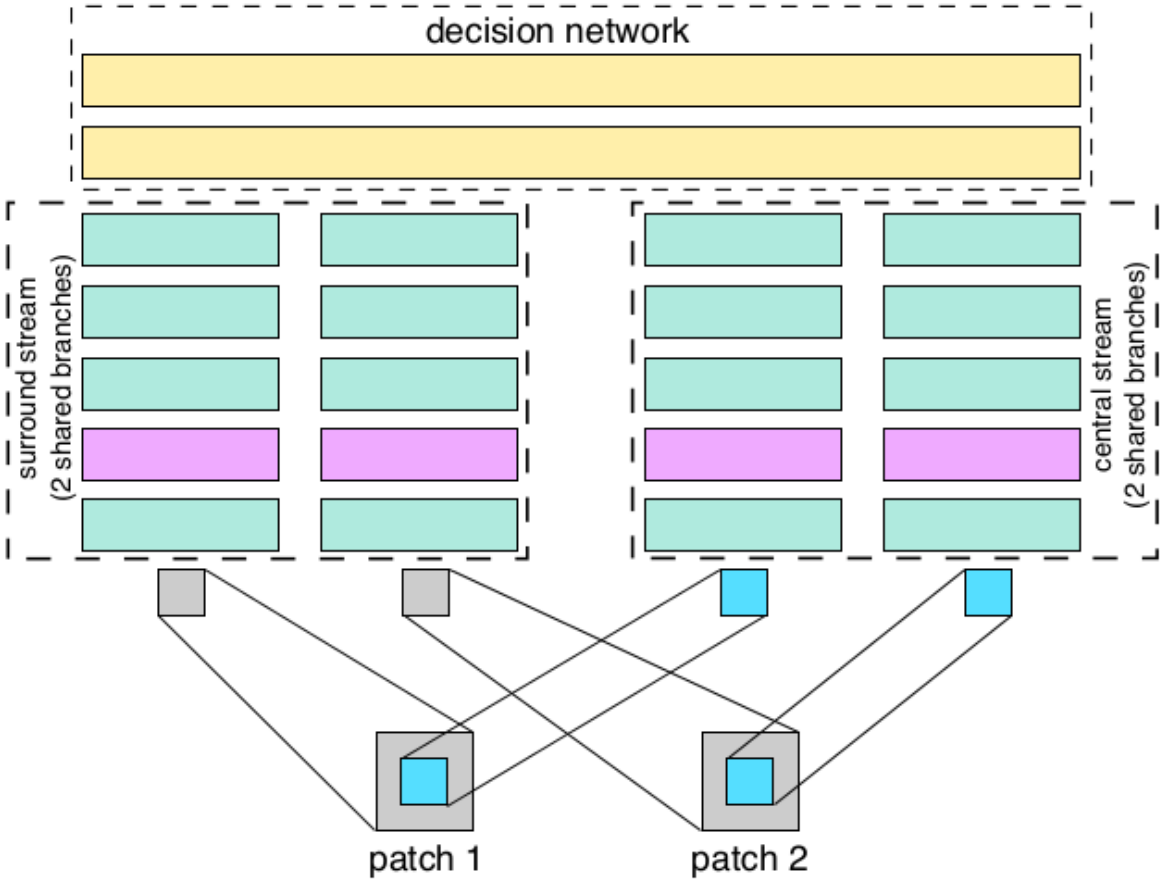
Four architectures are compared to learn the similarity function



1. 2-channel 2. pseudo-siamese (non-shared weights) 3. siamese

Method

Four architectures are compared to learn the similarity function



4. central-surround two stream network

Method

- output of decision network is scalar in $[0, 1]$
- output of branches are concatenated before entering fc layer
- Siamese : learns descriptors
- Pseudo-Siamese : branches don't share weights $\rightarrow$ more flexibiliy, double parameters
- 2-channel : no descriptors, the two patches are concatenated into a 2×3 channels image, at test time $O(n^2)$ inputs
- Central-surround : multiresolution should improve performance

Method : Loss function

N number of pairs in minibatch (batch size)

$y_i \in \{+1, -1\}$ groundtruth for pair i, corresponding / not

$o_i \in [-1, 1]$ output of decision network

w network weights

$$L(y_i, o_i) = \min_w \lambda ||w||_2 + \sum_i \max\,(0,\ 1 - y_i\, o_i)$$

Training:

- each of the three subsets has 500K pairs, half positive half negative
- train with one subset, test with another

Results

Metric : False Positive Rate at 95% recall (FPR = FP / TN+FN = probability of false positive, Recall = TP / TP+FN = probability of detection of positives)

Train	Test	2ch-2stream	2ch-deep	2ch	siam	siam-l_2	pseudo-siam	pseudo-siam-l_2	siam-2stream	siam-2stream-l_2	[19]
Yos	ND	**2.11**	2.52	3.05	5.75	8.38	5.44	8.95	5.29	5.58	6.82
Yos	Lib	**7.2**	7.4	8.59	13.48	17.25	10.35	18.37	11.51	12.84	14.58
ND	Yos	**4.1**	4.38	6.04	13.23	15.89	12.64	15.62	10.44	13.02	10.08
ND	Lib	4.85	**4.55**	6.05	8.77	13.24	12.87	16.58	6.45	8.79	12.42
Lib	Yos	5	**4.75**	7	14.89	19.91	12.5	17.83	9.02	13.24	11.18
Lib	ND	**1.9**	2.01	3.03	4.33	6.01	3.93	6.58	3.05	4.54	7.22
mean		**4.19**	4.27	5.63	10.07	13.45	9.62	13.99	7.63	9.67	10.38
mean(1,4)		**4.56**	4.71	5.93	10.31	13.69	10.33	14.88	8.42	10.06	10.98

- 2-channel best architecture : joint use information of the two patches right from the first layer is better
- Siamese architectures and specially central-surround beat previous best work : importance of multiresolution

Tracking

Fully-Convolutional Siamese Networks for Object Tracking. Luca Bertinetto, Jack Valmadre, Joao F. Henriques, Andrea Vedaldi, Philip H.S. Torr (U. Cambridge). *arXiv*:1606.09549

Interest

- new task, unrelated to classification
- metric learning well suited to it
- VOT'17 challenge winner
- simple !

Goal

Tracking **one arbitrary object** in a video, where the object is identified solely by a **bounding box in the first frame**

Previous approaches

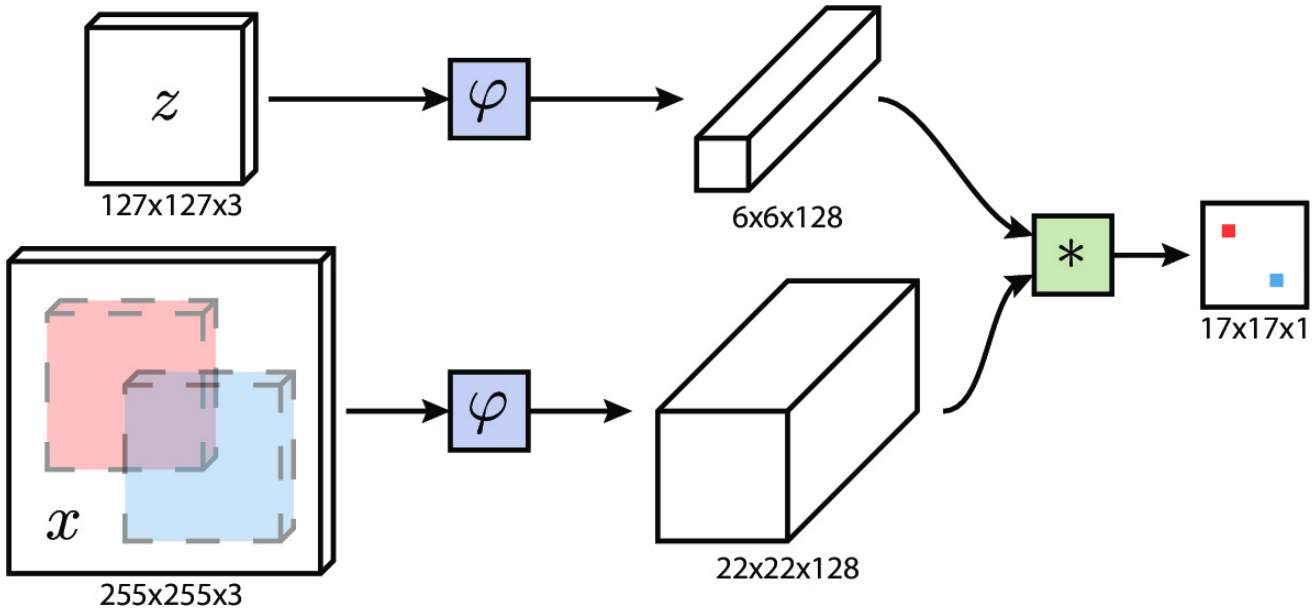
- learn a model of the object's appearance online, from scratch, using examples from the video
- scarce data $\rightarrow$ only simple models can be learnt
- fast or real-time constraint $\rightarrow$ no time to train a deep network to learn better representations

Method

- train a Siamese network to learn a **general** similarity function $f(z, x)$, **offline**
- z is exemplar or target, x candidate image (sliding window), of same size
- $f(z, x)$, returns a high score if z and x depict the same object, low otherwise
- f is learned from ImageNet Video dataset: 4500 videos of 30 classes of animals and vehicles, +1M annotated frames (bounding box of object)

Method

- Siamese φ is **fully convolutional**, no fully connected layers $\Rightarrow$ conmutes with translation, no need of sliding windows



- $z =$ **bbox of object** in the **first frame always**, $x =$ **search image**, region around previous detection and $\times 4$ its size
- output of network is not a plain vector but a **feature map**
- $f(x, z) = \varphi(z) * \varphi(x)$ is a **score map**
- φ is a **very simple CNN** : 5 conv-maxpool-relu layers

Method

- training is done with positive and negative pairs $(x_1, x_2, y) = (\text{exemplar, candidate, } \{+1, -1\})$
- **loss function** : $l(y, x_1, x_2) = \log(1 + \exp(-y\, f(x_1, x_2)))$
- each training frame produces a large number of positive and negative pairs
- positive pairs are those at a distance less than a threshold to the real object center
- minibatches include exemplar at 5 scales
- at test time, **86 frames per second**

Results

videos

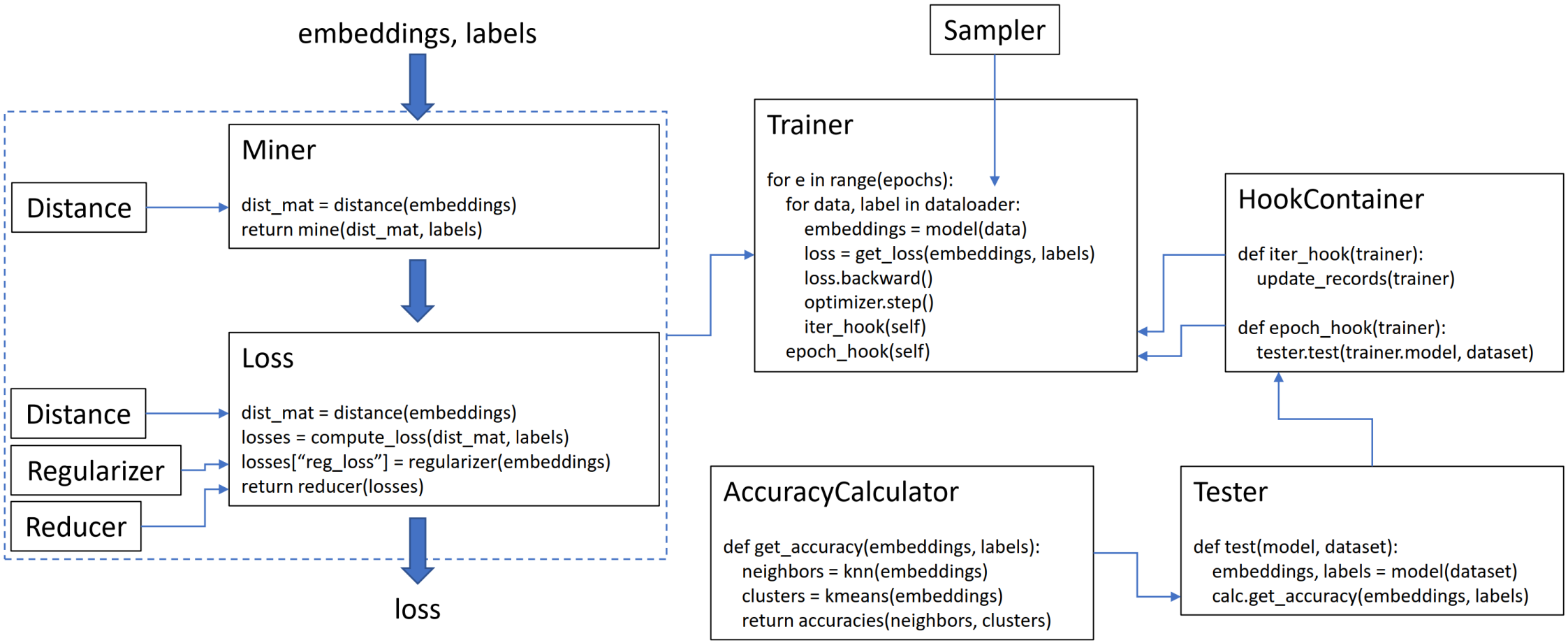
7. I want to try

But don't have much time...

- Siamese/Triplet networks, off and online sampling, mining etc. from Adam Bielski's Github : simple, easy, has the basics, +1,600 stars

I have time to learn

- Pytorch Metric Learning by Kevin Musgrove
  - very well documented, examples and Colabs
  - very well structured
  - many losses, mining methods
  - customizable, extendable
  - fast (FAISS library for kNN)


embeddings, labels
Sampler
Miner
dist_mat = distance(embeddings)
return mine(dist_mat, labels)
Distance
Trainer
for e in range(epochs):
for data, label in dataloader:
embeddings = model(data)
loss = get_loss(embeddings, labels)
loss.backward()
optimizer.step()
iter_hook(self)
epoch_hook(self)
HookContainer
def iter_hook(trainer):
update_records(trainer)
def epoch_hook(trainer):
tester.test(trainer.model, dataset)
Loss
dist_mat = distance(embeddings)
losses = compute_loss(dist_mat, labels)
losses["reg_loss"] = regularizer(embeddings)
return reducer(losses)
Distance
Regularizer
Reducer
AccuracyCalculator
def get_accuracy(embeddings, labels):
neighbors = knn(embeddings)
clusters = kmeans(embeddings)
return accuracies(neighbors, clusters)
Tester
def test(model, dataset):
embeddings, labels = model(dataset)
calc.get_accuracy(embeddings, labels)
loss

Wrap up

- Siamese and triplets are useful for **learning semantic similarity descriptors**
- Metric learning as a number of **applications**
- **Easy to implement** if you know how to make a classification network
- Just need to define a **loss** and sample **pairs or triplets**
- **Versatile** tool : define your own loss function, architecture, miner ...
- In practice, **mining matters** to training time and results
- Active field in deep learning that you should know

Sample exam questions

Q1. Consider a Siamese network implemented as two identical branches with shared weights. Let be

- $f(x)$ the d-dimensional vector produced by a branch of the network
- x_1, x_2 inputs to branches
- $D = ||f(x_1) - f(x_2)||_2$
- $y = 0$ if x_1 is considered similar to x_2 and 1 if not
- m a certain given value

a) What is the meaning of the following loss ? Explain it. How is it called ?

$$L(x_1, x_2, y) = (1 - y)\, D^2 + y\, (\max(0, m - D))^2$$

b) Draw a figure showing cases $(L > 0, y = 0)$, $(L > 0, y = 1)$, $(L = 0, y = 1)$

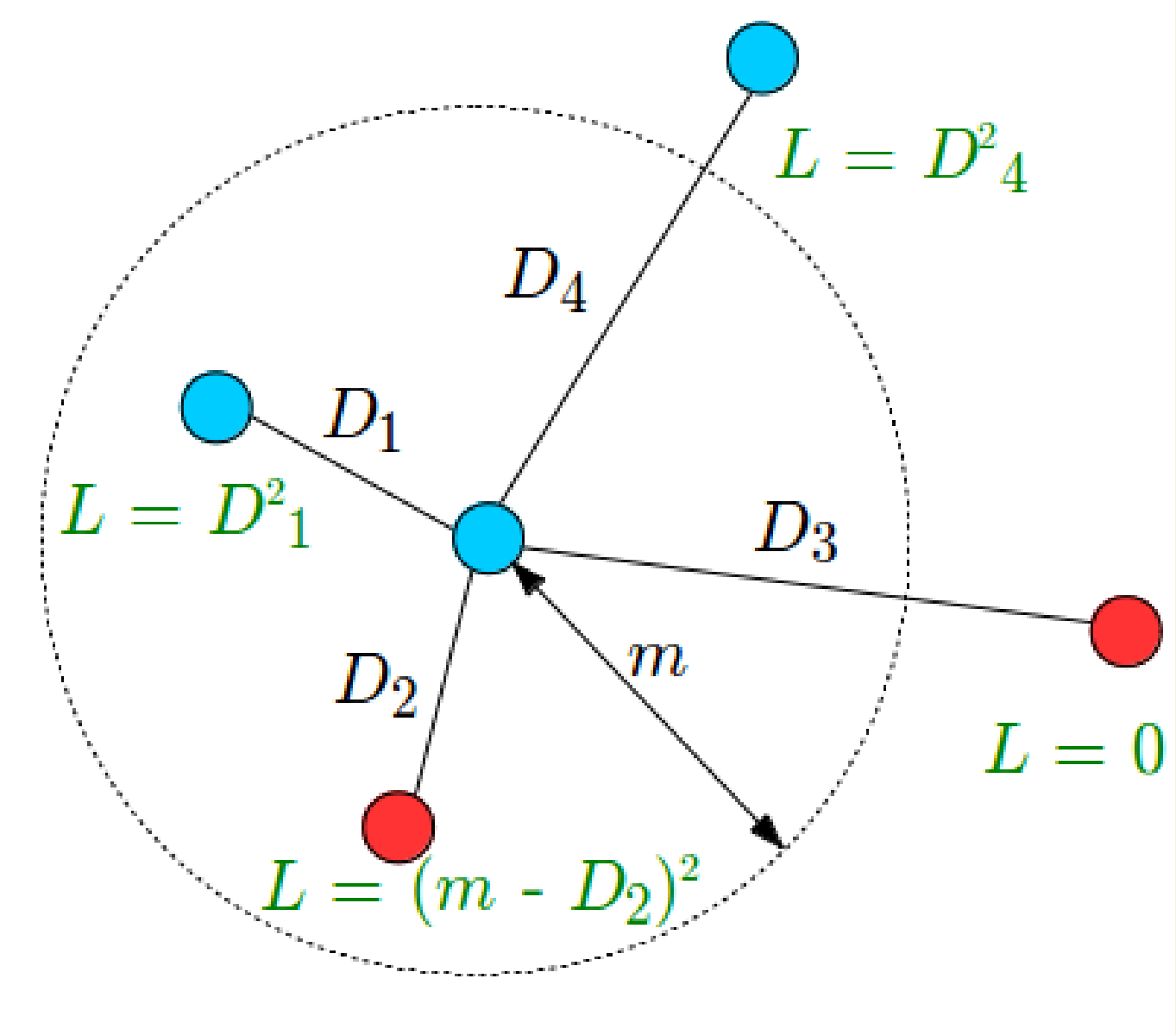
Answer

a) Contrastive loss.

If x_1 and x_2 are considered similar (for instance, belong to the same class) then the loss is the squared Euclidean distance of their representation: the more apart they are, the higher the loss, so this loss tends to pull together samples which are considered similar.

If x_1 and x_2 are dissimilar and Euclidean distance of represetations is less than m, the margin, then the loss is the positive difference between the margin and the distance. If distance is larger than the margin the loss is zero. Therefore this loss pushes away pairs of samples of different class.

b)



Q2. Consider again a Siamese network made of two branches with shared weights. For a minibatch of n samples we obtain $n/2$ pairs, being each one either positive or negative ($y = 0$ or 1). Each pair is one term of the loss. Is there a way to obtain more pairs from the same minibatch ? How ? Why is it better to do it this way ?

Answer

Instead of two branches we have just one. Pairs are made in the embedding space :

1. process all the n samples (pass them through the network) to obtain $f(x_1) \dots f(x_n)$
2. make *all* the possible pairs of transformed samples, $n(n-1)/2$ pairs in total

Each pair is a term of the loss. More terms is better, in principle, to estimate the gradient. This is called *online contrastive loss*.

Q3. What is a hard positive for a Siamese network with contrastive loss ? And a hard negative ?

Answer

Hard positive is a pair of representations $(f(x_1), f(x_2))$ which are far way (or farther than other positive pairs) and x_1, x_2 belong to the same class or are similar according to the groundtruth.

A hard negative is the opposite: $f(x_1), f(x_2)$ are close (or closer than other positive pairs) and x_1, x_2 belong to different classes.

Q4. To train a Siamese network suppose we draw a batch of pairs and denote by P and N the set of similar (positive) and dissimilar (negative) pairs, respectively. Interpret the following loss :

$$J = \sum_{(i,j)\in P} \max\left(0, J_{ij}\right)$$

$$J_{ij} = \max\left(\max_{(i,k)\,\in\, N} (m - d_{ik}), \max_{(j,l)\,\in\, N} (m - d_{jl}) \right) + d_{ij}$$

Why do it like this, instead of simply optimize the regular contrastive loss ?
What are we trying to do in this particular case (meaning of the inner and outer $\max$ terms) ?

Answer

We are doing mining. That is, looking for difficult samples to train, because this tends to yield better results (convergence to better minima), and/or faster convergence.

The way to do it in this case is to maximize the distance to hardest negative wrt $f(x_i)$ and hardest negative wrt $f(x_j)$ and at the same time minimize distance between positives $f(x_i), f(x_j)$.