NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

Masters Thesis

Metric Learning: A Deep Dive

Bill M. Psomas

ATHENS

SEPTEMBER 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

Διπλωματική Εργασία

Μάθηση Μετρικής: Μια Εμβάθυνση

Βασίλειος Μ. Ψωμάς

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΙΟΣ 2020

Masters Thesis

Metric Learning: A Deep Dive

Bill M. Psomas
A.M.:DS1180014

SUPERVISOR: Yannis Avrithis, Research Scientist, INRIA Rennes-Bretagne Atlantique

EXAMINATION COMMITTEE:

Yannis Avrithis, Research Scientist, INRIA Rennes-Bretagne Atlantique

Ioannis Emiris, Professor, National and Kapodistrian University of Athens

Vasileios Katsouros, Research Director, ATHENA Research and Innovation Center

SEPTEMBER 2020

Διπλωματική Εργασία

Μάθηση Μετρικής: Μια Εμβάθυνση

Βασίλειος Μ. Ψωμάς
Α.Μ.:DS1180014

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Γιάννης Αβρίθης, Ερευνητής, INRIA Rennes-Bretagne Atlantique

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ:

Γιάννης Αβρίθης, Ερευνητής, INRIA Rennes-Bretagne Atlantique

Ιωάννης Εμίρης, Καθηγητής, Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Βασίλειος Κατσούρος, Διευθυντής Έρευνας, Ερευνητικό Κέντρο ”Αθηνά”

ΣΕΠΤΕΜΒΡΙΟΣ 2020

ABSTRACT

Metric Learning is an important task in Machine Learning. The objective of Metric Learning is to learn a distance metric that reduces the distance between similar objects and increases the distance between dissimilar ones. Similarity and dissimilarity can be somehow subjective and thus some kind of supervision is needed in order to define the ground-truth. Learning such a distance metric can be proven to be really useful for many tasks, such as classification, retrieval and clustering. The classification and retrieval tasks can be simply reduced to class-level and instance-level nearest neighbor tasks respectively, while the clustering task can be made easier given the similarity matrix.

Traditionally, before Deep Learning, Metric Learning approaches were based either on linear transformations using the Mahalanobis or/and Euclidean distance, or on non-linear transformations using kernel-based methods. Both of them, however, had drawbacks. Linear transformations had a limited ability to capture nonlinear feature structure and thus could not achieve high performance over the new representation of the data. Non-linear transformations that carried the problem to a non-linear space could achieve optimum performance, but often suffered from overfitting. Apart from that, both methods were limited by their ability to process raw data and thus feature engineering was often needed.

With the remarkable success of Convolutional Neural Networks, Deep Metric Learning was introduced. In this context, Neural Networks are discriminatively trained to learn the non-linear mapping from input raw data to a lower dimensional and semantic embedding. This is usually done in a supervised way, in which the label annotations are given and thus these embeddings are optimized to pull samples with the same class label closer and push samples with different class labels apart. The whole training process is done by minimizing a loss function that should have exactly these properties. The great advantage of Deep Metric Learning is that it jointly extracts the features and learns the embedding.

The contribution of this work is threefold. First, we conduct extensive experiments using the most commonly used architectures (GoogLeNet, BNInception, ResNet50) on the most commonly used datasets (CUB200-2011, CARS196, Stanford Online Products) using 10 different loss functions (Contrastive, Triplet, LiftedStructure, NPair, ProxyNCA, ArcFace, Margin, MultiSimilarity, SoftTriple, ProxyAnchor) and four different embedding sizes (64, 128, 512, 1024). We make an ablation study and draw important conclusions using the results. Second, we introduce and propose a new setup for training using a fixed validation set. We conduct experiments using this and a 10-fold cross validation. Our setup seems to balance perfectly between the computational complexity and retrieval quality trade-off. Finally, we design, implement and experiment with a new loss function that is on a par with the state-of-the-art.

SUBJECT AREA: Computer Vision, Deep Learning

KEYWORDS: Metric Learning, Neural Networks

ΠΕΡΙΛΗΨΗ

Η Μάθηση Μετρικής είναι ένα σημαντικό πρόβλημα της Μηχανικής Μάθησης. Ο σκοπός της είναι η εκμάθηση μιας μετρικής, η οποία έχει την ιδιότητα να μειώνει την απόσταση μεταξύ όμοιων αντικειμένων και να αυξάνει την απόσταση μεταξύ ανόμοιων. Το τι είναι όμοιο και τι ανόμοιο μπορεί να είναι κάπως υποκειμενικό και ως εκ τούτου κάποια μορφή επίβλεψης είναι αναγκαία για να οριστούν. Η εκμάθηση μιας τέτοιας μετρικής μπορεί να αποδειχθεί πραγματικά χρήσιμη και σε πολλά άλλα προβλήματα, όπως είναι η ταξινόμηση, η ανάκτηση και η ομαδοποίηση. Τα πρώτα δύο προβλήματα μπορούν να αναχθούν σε προβλήματα κοντινού γείτονα σε επίπεδο κλάσης και οντότητας αντίστοιχα, ενώ το πρόβλημα της ομαδοποίησης μπορεί να γίνει ευκολότερο δοθέντος του πίνακα ομοιότητας.

Πριν τη Βαθιά Μάθηση, οι μέθοδοι στη Μάθηση Μετρικής βασίζονταν είτε σε γραμμικούς μετασχηματισμούς που χρησιμοποιούσαν την Mahalanobis ή/και την Ευκλίδεια απόσταση, είτε σε μη γραμμικούς μετασχηματισμούς που χρησιμοποιούσαν μεθόδους πυρήνα. Και οι δύο, ωστόσο, είχαν μειονεκτήματα. Οι γραμμικοί μετασχηματισμοί είχαν περιορισμένη ικανότητα σύλληψης μη γραμμικών δομών και έτσι δε μπορούσαν να πετύχουν υψηλή απόδοση, ενώ οι μη γραμμικοί μετασχηματισμοί που μετέφεραν το πρόβλημα σε ένα μη γραμμικό χώρο μπορούσαν να πετύχουν βέλτιση απόδοση, αλλά εμφάνιζαν το πρόβλημα της υπερ-προσαρμογής. Ακόμα, και οι δύο μέθοδοι είχαν περιορισμένη ικανότητα να επεξεργαστούν πρωτογενή δεδομένα και έτσι συχνά χρειαζόταν εξαγωγή χαρακτηριστικών.

Με την αξιοσημείωτη επιτυχία των Συνελικτικών Νευρωνικών Δικτύων, εμφανίστηκε η Βαθιά Μάθηση Μετρικής. Στο πλαίσιο αυτής, τα Νευρωνικά Δίκτυα εκπαιδεύονται να μάθουν τον μη γραμμικό μετασχηματισμό που συνδέει τα δεδομένα εκπαίδευσης με τις τελικές εμβαπτίσεις, οι οποίες έχουν μικρότερη διαστησιμότητα και περισσότερη σημασιολογία. Αυτό συνήθως συμβαίνει σε μια διαδικασία επιβλοπόμενης μάθησης, στην οποία οι κλάσεις κάθε δείγματος είναι γνωστές, και έτσι οι εμβαπτίσεις βελτιστοποιούνται ώστε δείγματα της ίδιας κλάσης να έρχονται κοντά και δείγματα διαφορετικής κλάσης να απωθούνται. Η όλη διαδικασία γίνεται ελαχιστοποιώντας μια συνάρτηση κόστους με αυτές τις ιδιότητες. Το σημαντικό πλεονέκτημα της Βαθιάς Μετρικής Μάθησης είναι ότι πραγματοποιεί από κοινού την εξαγωγή των χαρακτηριστικών και την εκμάθηση των εμβαπτίσεων.

Η συνεισφορά αυτής της διπλωματικής εργασίας είναι τριπλή. Πρώτον, πραγματοποιούνται εκτεταμμένα πειράματα χρησιμοποιώντας τις πιο διαδεδόμενες αρχιτεκτονικές (GoogLeNet, BNInception, ResNet50) στα πιο διαδεδομένα σετ δεδομένων (CUB200-2011, CARS196, SOP) χρησιμοποιώντας δέκα διαφορετικές συναρτήσεις κόστους (Contrastive, Triplet, LiftedStructure, NPair, ProxyNCA, ArcFace, Margin, MultiSimilarity, SoftTriple, ProxyAnchor) και τέσσερα διαφορετικά μεγέθη για τις εμβαπτίσεις (64, 128, 512, 1024). Πραγματοποιείται μελέτη των αποτελεσμάτων και εξάγονται σημαντικά συμπεράσματα. Δεύτερον, παρουσιάζεται και προτείνεται μια νέα διαδικασία εκπαίδευσης που χρησιμοποιεί σταθερό σετ δεδομένων επικύρωσης. Πραγματοποιούνται πειράματα χρησιμοποιώντας αυτή και μια δεκαπλή διασταυρωμένη επικύρωση. Διαπιστώνεται ότι η πρώτη ισορροπεί εξαιρετικά ανάμεσα στην υπολογιστική πολυπλοκότητα και την ποιότητα ανάκτησης. Τέλος, σχεδιάζεται και παρουσιάζεται μια νέα συνάρτηση κόστους, η οποία είναι ισότιμη με τις σύγχρονες μεθόδους.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Όραση Υπολογιστών, Βαθιά Μάθηση

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Μετρική Μάθηση, Νευρωνικά Δίκτυα

ACKNOWLEDGEMENTS

Half of this work was conducted in the Linkmedia Team of INRIA Rennes-Bretagne Atlantique and the other half in Athens. I would like to thank my supervisor Yannis Avrithis from the bottom of my heart for his valuable help, guidance and support, his endless motivation and the trust he showed me, as also the postdoctoral student Mateusz Budnik for the technical knowledge he shared with me and his willingness to help.

Additionally, I would like to thank all the people of Linkmedia Team for creating a really pleasant and warm work environment, especially given the fact that my stay in INRIA coincided with the coronavirus quarantine. Being welcome and included as a new member of the group meant a lot to me.

Finally, I would like to thank my family and friends for the love and support they have provided me throughout this project.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Motivation

The concepts of similarity and dissimilarity are fundamental for both the human cognitive process, as also for artificial systems. They can be very useful for tasks as classification, clustering, recognition, information retrieval etc. Similarity and dissimilarity, though, can be somehow subjective and thus the questions that arise are what exactly is meant by the term similarity, where exactly it refers and how to assess it. To make it more clear, when comparing two faces with the goal of finding whether they are matching or not, then the term similarity refers to a similarity function that should emphasize on appropriate features, like the hair color, the shape of eyes, the size of nose etc. But when the goal is to determine just the pose, then a function that captures pose similarity is required and the term similarity refers only to this pose. This means that the notion of good similarity function is problem-dependent, or in other words, each problem has its own semantic notion of similarity, which often cannot be captured by standard functions or metrics (e.g. Euclidean distance), and thus the problem arising is how exactly to chose this function or metric.

A naive solution would be a handcrafted one, in which one would attempt to determine by hand this function, by the combination of appropriate features. This solution would be expensive in terms of effort and probably not so robust to changes in data. An alternative would be to learn task-specific similarity functions and automate this process. The term “learn” refers to Machine Learning, a definition of which is “a field of Computer Science that aims to teach computers how to learn and act without being explicitly programmed”. Machine Learning algorithms make use of data in order to build models that can later predict or make decisions. Learning a similarity function or equivalently learning a distance function (or metric) that reduces the distance between similar objects and increases the distance between dissimilar ones is the scope of Metric Learning.

Before Deep Learning, Metric Learning approaches are either linear using distances as the Euclidean, Mahalanobis etc., either non-linear using kernel-based methods. The linear ones have difficulties in capturing non-linear feature structures and thus cannot achieve high performance, while the non-linear ones are often suffering from overfitting. In 2012, Alexnet [9], a deep Convolutional Neural Netowrk that combines a lot of existing ideas like the utilization of GPUs during training and of the non-saturating ReLU [21] as an activation function, is the winner of the ILSVRC [8] (ImageNet Large Scale Visual Recognition Competition) achieving remarkably lower error rate than its competitors. This is considered to be the starting point of the Deep Learning era, as after this the scientific community gradually turns to Neural Networks in order to find better solutions to various tasks.

In this context, in 2015, authors of [1] introduce the setup for Deep Metric Learning, using the GoogLeNet [10] architecture on 2 existing datasets (CUB200-2011 [19] and CARS196 [20]) and on 1 new they collect (SOP). All datsets are split in half, using half of the classes for training and the other half for testing. This paper also introduces a new loss function, LiftedStructure, with greater task-specific properties than the existing Contrastive [22, 23] and Triplet [13, 24]. This setup achieves significant improvement over existing feature embedding methods and opens a new path full of challenges for Metric Learning.

1.2 Challenges and Related Work

In order to understand the challenges of Deep Metric Learning, it is firstly essential to describe the whole pipeline of it. A CNN initialized with the ImageNet weights takes as input images from the dataset, extracts features through its layers and outputs a n-dimensional embedding. This CNN is trained having available label annotations for each image and using a loss function that should have the property to pull images with the same class label (positives) closer and push images with different class labels (negatives) apart in the embedding space. This means that the objective of the whole training of the Network is to output embeddings that are dimensionally lower and semantically more meaningful than the original images. For example, given a dataset of car images like the CARS196, the embeddings of SUVs should live close each other and far away from the embeddings of the roadsters, as shown in Fig. 1. Half of the classes of the dataset are used for training and the other half for testing. This means that during test time the query image is an image of a class that the Network has never seen before and this makes the task different from classification.

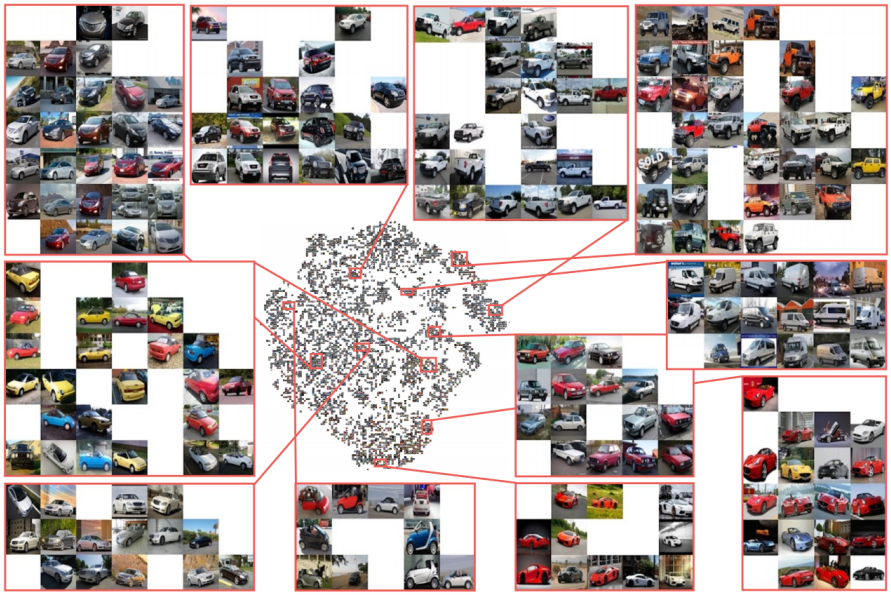


Figure 1: Visualization of the embedding space on the test split of CARS196 using the LiftedStructure loss [1].

So, how exactly should this loss function be? Can it be a classification one like the CrossEntropy or should it be different? If CrossEntropy can really be used, then are there any other changes to the rest of the setup needed? Losses like Contrastive and Triplet introduce a new setup, in which the samples are examined in pairs and triplets respectively. A pair consists of either two positives or two negatives, while a triplet consists of an anchor, a positive and a negative. Contrastive attempts to make the distance between positive pairs smaller than some threshold, and the distance between negative pairs larger than this threshold. Triplet attempts to make the anchor-positive distance smaller than the anchor-negative distance. The rationale of those two loss functions seems to be closer to the Metric Learning objective, but do they truly give better results? And if they do, is there enough space for improvements or is there enough space for something new at all? LiftedStructure introduces the idea of exploiting the whole batch during training, rather than individual pairs or triplets of it and this gives a new direction that should be investigated. To sum up, the design of the loss function is truly a challenge. In the next chapters, more

state-of-the-art loss functions will be examined and many of the aforementioned questions will be answered.

The loss function can be computed using pairs, triplets or even the whole batch. Mining is the process of finding the most informative samples and use those for training. So, instead of having, for example, two easy positives (images of the same class that really have great similarity) in a pair, which will not contribute so much to the loss function, one can have two hard positives (images of the same class that should have great similarity, but do not). This is called hard mining and a typical implementation of it can be found in [25]. While hard mining contributes in consuming less memory and avoiding plateaus in performance, it can produce noisy gradients and converge to bad local minima [26]. That is why other types of mining have been introduced, as the semi-hard negative mining [13], the distance-weighted mining [26] etc. All these approaches are in the direction of mining within each batch, but the batch itself remains randomly selected. Loss functions like LiftedStructure or MultiSimilarity [14] that make use of the whole batch during training cannot really take advantage of it. For loss functions of this kind, it would be useful to mine within the whole dataset in order to construct the batch using the most informative images. Harwood et al. [27] propose doing a nearest neighbor search before each epoch in order to mine semi-hard samples from the dataset, but they make use of a generalized Triplet that probably cannot be fully benefited from it.

The Neural Network itself is surely another challenge. Which Network with which embedding size is the most suitable? Having a ResNet50 [12] 512-dimensional embedding is for sure richer in terms of information than having a GoogLeNet 64-dimensional embedding, but is this extra information really needed? Chao-Yuan Wu et al. [26] make experiments using the ResNet50 with an embedding size of 128 and compare their Margin loss with other losses that are using other Networks and other embedding sizes. And this is something to be seen in a lot of other papers too. Are these comparisons fair? Fehervari et al. [28] recently addressed this problem, but they mainly focused on making experiments using former methods and did not achieve to create a real benchmark.

The last challenge to be mentioned is the setup of training. In almost all the Machine Learning tasks the setup follows the pipeline of training, validation and testing. The data are split in 3 different sets respectively and the significance of each is great. Hyun Oh Song et al. [1] introduced a setup without a validation set and this makes the hyperparameter tuning probably more difficult, unstable and not a good tactic. How sure can we be sure that we have found the optimal hyperparameters, when we fine-tune using the test set? How sure can we be that our model is able to generalize, when the selection of it is done with direct feedback from the test set? And how correct can this be? This is something that should also be investigated.

1.3 Contributions

Having outlined some of the most important challenges of Deep Metric Learning, the contributions, which are greatly related with the challenges, are presented below:

- We conduct extensive experiments using the most common CNN architectures (GoogLeNet, BNInception, ResNet50) on the most common used datasets (CUB200-2011, CARS196, SOP) using 10 different loss functions (Contrastive, Triplet, LiftedStructure, NPair, ProxyNCA, ArcFace, Margin, MultiSimilarity, SoftTriple, ProxyAnchor) and 4 differ-

ent embedding sizes (64, 128, 512, 1024). This work can be considered as a benchmark for fair comparisons and a great ablation study.

- We propose a new setup for training, in which a fixed validation set is used. We make experiments using this and a 10-fold cross validation and draw conclusions about the performance vs. computational complexity trade-off.
- We design, implement and experiment with a new loss function that is proxy-based, but it is in between the rationale of classification and embedding losses. It treats proxies of different class as negatives that should be pushed away and samples of the same class as positives that should be pulled by the corresponding proxy.

1.4 Structure

In this work we attempt to make extensive experiments under equal terms in order to create a benchmark that will help us gaining insight into Deep Metric Learning. The chapters of this work are organized as follows:

- **Chapter 1** makes an introduction and definition of Metric Learning while also presenting the challenges, related work and contributions of this work.
- **Chapter 2** presents the story of Metric Learning in detail, starting from former methods and ending with Deep Metric Learning. It highlights the most important loss functions.
- **Chapter 3** presents details about the experimental setup, as also some issues that made us conduct the experiments.
- **Chapter 4** presents the experimental results along with a discussion about the findings
- **Chapter 5** proposes a new setup using a fixed validation set, while also presenting a cross validation alternative
- **Chapter 6** proposes a new proxy-based loss function that performs on a par with the state-of-the-art
- **Chapter 7** outlines the findings and future research directions.

2. BACKGROUND

Metric Learning aims to learn a distance metric or a similarity function that reduces the distance between similar samples and increases the distance between dissimilar ones. In this chapter, we will study different approaches of Metric Learning that existed over the years starting from the former ones and ending with the state-of-the-art. The remarkable success of Convolutional Neural Networks that brought changes to almost all the domains of Machine Learning and Computer Vision could not but have affected Metric Learning too. In the first section we will have a quick look on the approaches before Deep Learning, in the second one we will study the essentials of Neural Networks that will finally lead us to Deep Metric Learning of the third section.

2.1 Metric Learning

Let us start by formulating the Metric Learning problem. Given an input distance function $d(x, y)$ or a similarity function $s(x, y)$ between two objects x and y, along with supervised information regarding the ideal distance or similarity, construct a new distance function $d'(x, y)$ or a similarity function $s'(x, y)$ which is "better" than the original one. This new distance function can be of the form $d(f(x), f(y))$, which means that we learn a mapping f over data but utilize the original distance or similarity function. In this Chapter we will have a look on this formulation of the problem, where the point is to learn this mapping f having some kind of supervision, as this formulation is really close to the formulation of the Deep Learning era that will be discussed afterwards.

The mapping f can be either linear or nonlinear. In both cases, the input distance function can be the Euclidean, i.e. $d(x, y) = \|x - y\|_2$. In the linear case we aim to learn a linear mapping based on supervision, which can be encoded as a matrix G such that the learned distance is $\|Gx - Gy\|_2$, while in the nonlinear case the distance function has the general form of $\|f(x) - f(y)\|_2$. This can be done by extending linear methods via kernelazation. The basic idea is to learn a linear mapping in the feature space of some potentially non-linear function ϕ; that is; the distance function may be written $d(x, y) = \|G\phi(x) - G\phi(y)\|_2$, where ϕ may be a nonlinear function. Assuming that the kernel function $(x, y) = \phi(x)^T\phi(y)$ can be computed, it turns out that we may efficiently learn G in the input space using extensions of linear techniques. A quick overview of linear and nonlinear Metric Learning is presented below.

2.1.1 Linear Metric Learning

Let us suppose that we have a set of data points $x_1, ..., x_n$. Let $X = [x_1, ..., x_n]$ be the matrix of all the data points. We will use the Euclidean distance $\|x_i - x_j\|_2 = \sqrt{(x_i - x_j)^T(x_i - x_j)}$ as the canonical distance measure and the inner product $x_i^T x_j$ as the canonical inner product. The Mahalanobis distance is defined as $d_M(x_i, x_j) = \sqrt{(x_i - x_j)^T\Sigma^{-1}(x_i - x_j)}$ where Σ is the covariance matrix of the data. The Mahalanobis distance is closely related to data whitening, as it can be easily proven that the Euclidean distance between two whitened variables is simply the Mahalanobis distance. A whitening transformation is a decorrelation transformation that transforms a set of random variables into a set of new random variables with identity covariance (uncorrelated with unit variances). This is very useful in cases where there are outliers in data that would dominate the computation of the

Euclidean distance. Using Mahalanobis would avoid this, as the data would be implicitly whitened.

In the Metric Learning literature, the term "Mahalanobis distance" is often used to denote the distance function of the form: $d_A(x, y) = (x - y)^T A(x - y)$, where A is some positive semi-definite matrix. We can view this distance simply as applying a linear transformation of the input data: since A is positive semi-definite, we factorize it as $A = G^T G$ and simple algebraic manipulations show that $d_A(x, y) = \|G_x - G_y\|_2^2$ and thus this generalized notion of Mahalanobis distance exactly captures the idea of learning a global linear transformation.

Having said this, we will now form a generalized model for linear Metric Learning, which is proposed by [2] and captures most of the techniques. Through this we will gain a good insight of how most of the linear Metric Learning approaches were working before Deep Learning, without having to delve deep into them. Recall that the aim of Metric Learning is to learn a new distance function using supervision that is a function of the learned distance. Given the squared Euclidean as the original distance, the learned distance would be again the squared Euclidean distance after applying the transformation G globally, as already mentioned. To encode the supervision , we will assume that we are given a collection of m loss functions that we will denote as $c_1, ..., c_m$ and we will make the assumption that each loss function depends on the data only through the mapped inner product matrix $X^T G^T G X = X^T A X$. For example, one loss function might encode the squared loss between the target distance between x_i and x_j and the squared Euclidean distance between x_i and x_j using the Mahalanobis distance with A. This forms the second part of Eq. 2.1, while the first part is a regularizer on the model, which will be a function of A.

$$\mathcal{L}(A) = r(A) + \lambda \sum_{i=1}^{m} c_i(X^T A X). \tag{2.1}$$

The λ term behaves as a trade-off between the regularizer and the loss. The goal is to find the minimum of $\mathcal{L}(A)$ over the domain A, which we will denote as $dom(A)$. Sometimes this model will be specified as a constrained optimization problem. The two most popular forms of constraints are the (a) similarity/dissimilarity constraints, and (b) relative distance constraints. For similarity constraints, we are given a set of pairs $(i, j) \in \mathcal{S}$ of objects that should be similar, and a set of pairs $(i, j) \in \mathcal{D}$ of objects that should be dissimilar and we want to ensure that:

$$d_A(x_i, x_j) \leq u, (i, j) \in \mathcal{S} \tag{2.2}$$

$$d_A(x_i, x_j) \geq l, (i, j) \in \mathcal{D} \tag{2.3}$$

We can encode these as loss functions in various ways; for example, the Hinge loss would encode the desired constraints as:

$$c(X^T A Y) = \max(0, d_A(x_i, x_j) - u), (i, j) \in \mathcal{S} \tag{2.4}$$

$$c(X^T A Y) = \max(0, l - d_A(x_i, x_j)), (i, j) \in \mathcal{D} \tag{2.5}$$

Note that the above losses rely on appropriate choice of u and l. The other popular form of constraints is utilizing relative distances. These are specified via a triple $(i, j, k) \in \mathbb{R}$

which denotes that x_i should have a smaller distance to x_j than x_j to x_k:

$$d_A(x_i, x_j) < d_A(x_i, x_k) \tag{2.6}$$

Note that, unlike similarity and dissimilarity constraints, the relative distances do not require one to specify any parameters. However, typically one adds a margin to the above constraint:

$$d_A(x_i, x_j) < d_A(x_i, x_k) - m \tag{2.7}$$

These can be encoded into loss functions analogously to the similarity and dissimilarity constraints. Regarding regularizers, some of the most popular are:

$$r(A) = \frac{1}{2}\|A\|_F^2 \tag{2.8}$$

$$r(A) = \mathrm{tr}(AC) \tag{2.9}$$

The regularizer of Eq. 2.8 is based on the squared Frobenius norm and can be found in various models, as the Schultz and Joachims [29], the Kwok and Tsang [30], the Pseudo-Metric Online Learning Algorithm (POLA) [31], etc. The regularizer of Eq. 2.9 is a linear function with respect to the Mahalanobis matrix A and can be found in models like the Large-Margin Nearest Neighbors (LMNN) [32], Neighborhood Component Analysis (NCA) [33], Maximally Collapsing Metric Learning (MCML) [34], etc. Most of the times $C = \sum_{x_i, x_j \in \mathcal{S}} (x_i - x_j)(x_i - x_j)^T$, while $C = I$, where I is the identity matrix, can also be seen.

2.1.2 Nonlinear Metric Learning

Before discussing the nonlinear Metric Learning approaches, it is considered important to mention the naive XOR example (Fig. 2), in order to demonstrate the limitations of linear methods and motivate the need of nonlinear metrics.

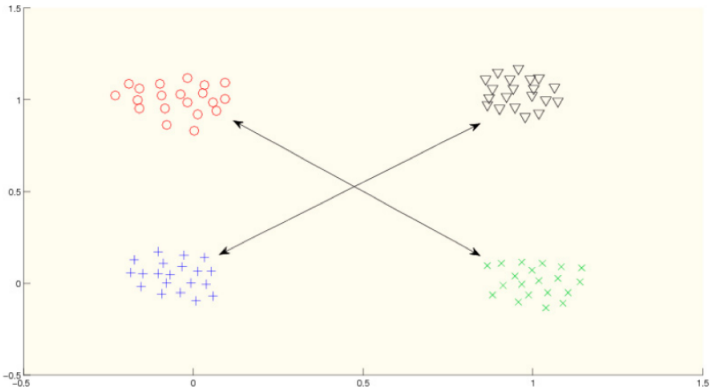


Figure 2: Suppose that we would like red and green points to belong in the same class, while black and blue to belong in another one. There is no line or no linear transformation in this plane that could successfully satisfy these constraints. Note that a linear transformation is equivalent to stretching the axes and rotating the data. [2].

The most common way to go from linear to nonlinear methods is via kernelazation. Recall the linear model discussed previously:

$$\min_{A \succcurlyeq 0} r(A) + \lambda \sum_{i=1}^{m} c_i(X^T A X), \tag{2.10}$$

where r is the regularizer and c_i are the loss functions. The algorithms discussed for optimizing this model generally require updating A iteratively, using the data points from X directly. But we can also design algorithms that do not access the data directly; instead, they only require access to inner products $x_i^T x_j$ between data points. Then we can generalize the resulting algorithms to utilize kernel functions instead of inner products. Such kernel functions represent inner products in a high or infinite dimensional space, and applying such algorithms will correspond to learning nonlinear transformations in the input space.

More specifically, a kernel function can be written as $\kappa(x, y) = \phi(x)^T \phi(y)$ for some nonlinear function ϕ. Applying linear metric learning algorithms with this inner product corresponds to learning a linear transformation in the space of ϕ function (i.e., the feature space). Therefore, we can write these methods as learning distances of the form $\|G\phi(x) - G\phi(y)\|_2$, which is a nonlinear transformation over the input data in general. In most cases, using kernelization, algorithms for the linear case can easily be generated to algorithms for the nonlinear case.

Let us have a quick look on an example of kernelization. Consider the task of Euclidean nearest neighbor classification. For each query x_q, we compute the distance to each point x in the database via $\|x_q - x\|_2$ and then classify x_q based on the labels of the nearest neighbors. Using the fact that $\|x - y\|_2^2 = (x - y)^T (x - y)$, we can write $\|x_q - x\|_2 = \sqrt{x_q^T x_q - 2x^T x_q + x^T x}$. Replacing the inner products with a kernel function $\kappa(x_q, x) = \phi(x_q)^T \phi(x)$, the distance function is expressed as $\sqrt{\kappa(x_q, x_q) - 2\kappa(x_q, x) + \kappa(x, x)}$. The resulting distance is a "kernelized" distance between the query and a datapoint.

How do we choose the kernel function, though? In order for a kernel function to be valid, it must represent an inner product between data points in a Hilbert space induced by the mapping ϕ. One way to express this requirement is that any matrix K of kernel function values defined over a set of datapoints $x_1, ..., x_n$ must always be positive semi-definite. In practise, one chooses a kernel from a set of known kernel functions. Two popular choices are the polynomial kernel and the Gaussian kernel. The polynomial kernel is defined as $\kappa(x, y) = (x^T y + c)^d$ for positive reals c and d, while the Gaussian kernel is defined as $\kappa(x, y) = e^{\frac{-\|x-y\|_2^2}{2\sigma^2}}$.

2.2 The story of Neural Networks in fast forward

When telling a story, it is always better to start from the beginning. This beginning can go back to 17^{th} century, when the chain rule that underlies the back-propagation algorithm of Neural Networks was invented [35], [36], or can go back to 19^{th} century, when the gradient descent was invented as an algorithm that can iteratively approximate the solution of optimization problems. Both inventions proved to be essential for Neural Networks and are highly used until now, but probably the most accurate beginning of this story is in 1962 and Frank Rosenblatt's book "Principles of Neurodynamics" [37], in which the Perceptron

is for the first time introduced.

2.2.1 Perceptron

While the Perceptron originally refers to a wide range of Network architectures, learning algorithms and hardware implementations, and can be seen as a great introduction to Neural Networks, it was due to Marvin Minsky that Perceptron was considered as a binary linear classifier and an algorithm.

Marvin Minsky with Seymour Papert wrote the book “Perceptrons” [38] in 1972 criticising the work of Rosenblatt by highly focusing to the flaws of linear models, such as their inability to learn the XOR function. While the Perceptron as a single layer Network with only one node is only capable of learning linearly separable patterns (and thus truly cannot learn the XOR function), the same does not hold for a Multilayer Perceptron. This book somehow misguided the research community and is probably the center of controversy in the history of AI, as it is often claimed that it had a great impact in discouraging research on Neural Networks in $1970s$ and contributing to the so-called “AI Winter”.

But let us formulate mathematically the Perceptron as an algorithm. Given an input $x \in \mathbb{R}^d$, the Perceptron is a generalized linear model

$$y = f(x; w) = \text{sgn}(w^\top x) \tag{2.11}$$

where $w \in \mathbb{R}^d$ is a weight vector to be learned, and

$$\text{sgn}(x) = \begin{cases} +1, & x \geq 0 \\ -1, & x < 0 \end{cases} \tag{2.12}$$

is the step function. An input x with output $y = f(x; w) = \text{sgn}(w^\top x)$ is classified to class C_1 if $y = 1$ and to C_2 if $y = -1$. Given a training sample $x \in \mathbb{R}^d$ and a target variable $s \in \{-1, 1\}$, x is correctly classified if the output $y = f(x; w)$ equals s, i.e. $sy > 0$.

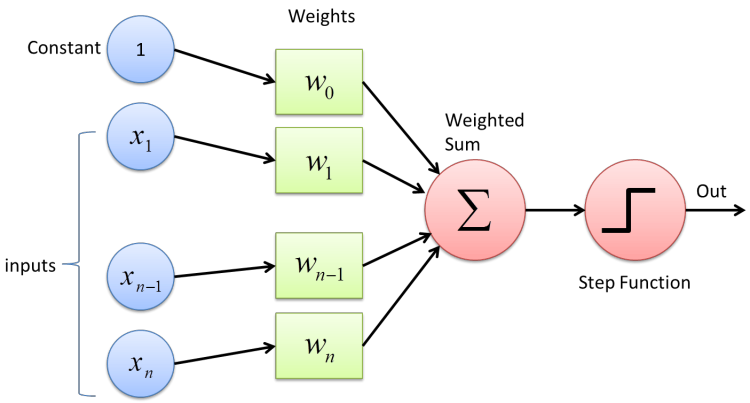


Figure 3: The Perceptron architecture. Apart from the $x_1, ..., x_n$, there is also a $x_0 = 1$ constant term, which is added, so that the weight w_0 plays the role of bias and thus the model takes the generalized form of $y = f(x; w, b) = \text{sgn}(w^\top x + b)$ [3].

In Fig. 3 training samples $x_1, ..., x_n$ are given to a Perceptron, along with target variables $s_1, ..., s_n \in \{-1, 1\}$. Starting from an initial weight vector $w^{(0)}$, the algorithm will learn by iteratively choosing a random sample x_i that is misclassified and updating the weight vector following the rule $w^{(t+1)} \leftarrow w^{(t)} + \epsilon s_i x_i$, where ϵ is a learning rate. This will finally lead to a weight vector (decision boundary) that correctly classifies all data points, as shown in Fig. 4.

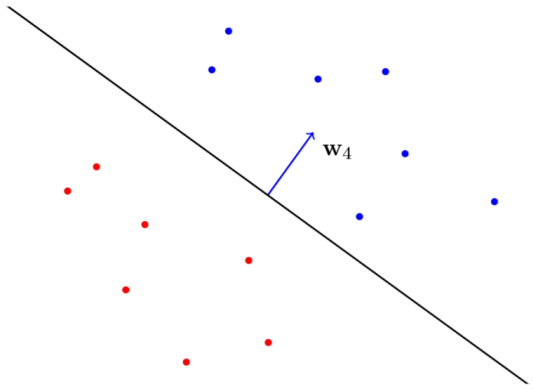


Figure 4: The Perceptron algorithm correctly classifies all data points. The decision boundary w_4 is the weight vector occurred after 4 iterations of the algorithm [4].

2.2.2 Multilayer Perceptrons

While the Perceptron, as already mentioned, can be seen as a generalized linear model, Multilayer Perceptrons or Feedforward Networks can be seen as efficient nonlinear function approximators. Let us denote the function to be approximated as f^* and then take as an example a classifier $y = f^*(x)$ which maps an input x to a category y, as it was happening in Perceptron. A Multilayer Perceptron defines a mapping $y = f(x;\theta)$ and learns the value of the parameters θ that result in the best function approximation. The function f is typically composed of many different functions and the model is associated with a directed acyclic graph describing how the functions are computed together. This can also be called the architecture of the Network. Each application of a different function can be seen as a new representation of the input. A naive example would be a function $f(x) = f^{(2)}(f^{(1)}(x))$, in which functions $f^{(1)}$ and $f^{(2)}$ are connected in chain and represent respectively the first and the second layer of this Multilayer Perceptron. The overall length of this chain gives the depth of the model. The first layer is called the input layer, while the final layer is called the output layer. The intermediate layers are called hidden, and that is because of the fact that the training data does not show the desired output for each of these layers, but only for the output layer. This general architecture can be seen in Fig. 5.

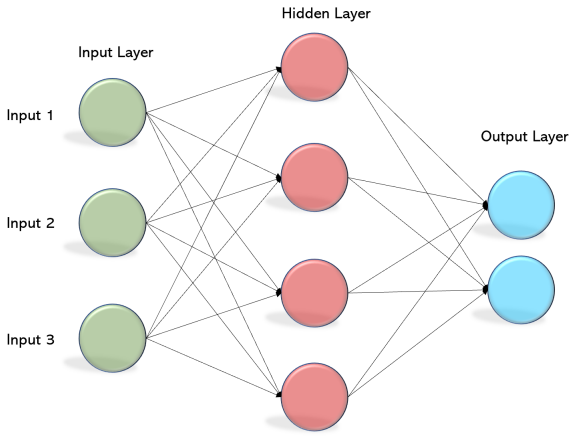


Figure 5: The general architecture of a Multilayer Perceptron. The hidden layers can be more than one and thus changing the overall depth of the model. Each of the nodes is a Perceptron, exactly as described in Fig. 3 [5].

It is probably because of this terminology using the words ”layer” and ”depth” that the name ”Deep Learning” arose. The objective of training is to drive $f(x)$ to match $f^*(x)$. The training data provides us with noisy, approximate examples of $f^*(x)$ evaluated at different training points. Each example x is accompanied by a label $y \approx f^*(x)$. The training examples specify directly what the output layer must do at each point x, but do not specify what the hidden layers should do. Instead, the learning algorithm must decide decide how to use these layers to best implement an approximation of f^*.

When dealing with MLPs, one will probably have to make some design decisions concerning their width and depth. The width has to do with how many neurons there will be in each hidden layer, while the depth has to do with how many hidden layers there will be in the Network. Apart from that, in each neuron there is an activation function that introduces the concept of nonlinearity in the Network and that should also be chosen wisely. Examples of activation functions are: the step function that was used in Perceptron, the sigmoid $\sigma(x) = \frac{1}{1+\exp^{-x}}$, the hyperbolic tangent $tahh(x) = \frac{\exp^{x}-\exp^{-x}}{\exp^{x}+\exp^{-x}} = 2\sigma(x) - 1$, the rectified linear unit $relu(x) = [x]_+ = \max(0, x)$, etc. The last design decisions have to do with the optimizer, the scheduler, the loss function, the form of the output units, etc. The whole training process could not be done without the back-propagation algorithm and its modern generalizations, as this algorithm is used to efficiently compute the gradients and back-propagate them, so that all the weights of the Network are updated. Apart from the loss functions, this work will not explicitly present any other design options, as they are not crucial for its scope. The subsections that follow introduce some of the most popular Network architectures that share something in common: convolutions.

2.2.3 LeNet

Before delving into the LeNet [6] architecture, we will firstly introduce the idea of convolution. In its most general form convolution is an operation on two functions of a real-valued argument. Let us make an example to motivate the definition of convolution. Suppose we are tracking a car using a sensor. The sensor provides a single output $x(t)$, which is the position of the car at time t. Both x and t are real-valued, as we are getting different readings from the sensor at any instant time. Supposing that the laser is noisy and we want to obtain less noisy estimates of car's position, we could try to average together several measurements. More recent measurements are probably more relevant, so we would also like to introduce weights to this average. We can do this with a weighted function $w(a)$, where a is the age of measurement. Eq. 2.13 defines a smoothed estimate of the position of the car and is called convolution.

$$s(t) = \int x(a)w(t-a)da \tag{2.13}$$

Convolution is typically denoted with an asterisk and is used in discretized data, so Eq. 2.14 can be written as:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2.14}$$

While LeNet is not the first Convolutional Neural Network to be introduced (Fukushima in 1980 introduced neocognitron [39], a biologically-inspired Network that uses convolutions), is the first one to combine different ideas in a learning scheme that was working, and

was implemented for a Computer Vision task as the document recognition. Convolutions leverage three important ideas that can improve the performance of a Network: sparse interactions, parameter sharing and equivariant representations. In order to understand these, we first have to mention that for Computer Vision tasks (or in general tasks that deal with images) the first argument x of Eq. 2.14 is the input image, the second argument w is the kernel and the output is the feature map. Convolution can be seen as the procedure of sliding a kernel (or equivalently a filter) over an image. Sliding different kernels produces different feature maps that contain different type of information and exploit different underlying correlations/relations among the pixels.

Sparse connections are accomplished by making the kernel smaller than the input image. For example, when processing an image, the input might have hundreds of millions of pixels, but we can detect small, meaningful features such as edges that occupy only tens or hundreds of pixels. This means that we store fewer parameters, as also that computing the output requires fewer operations. Parameter sharing refers to using the same parameter for more than one function in a model. In MLPs that we saw before, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. In the convolutional part of CNNs, each member of the kernel is used at every position of the input. The parameter sharing used by the convolution operation means that rather than learning a separate set of parameter for every location, we learn the same one. As we will later see, CNNs are using both convolutional layers and MLPs (that in this concept are commonly named fully connected layers). Finally, the particular form of parameter sharing in convolution causes the layer to have the property of equivariance to translation. This means that if the input changes, the output changes in the same way.

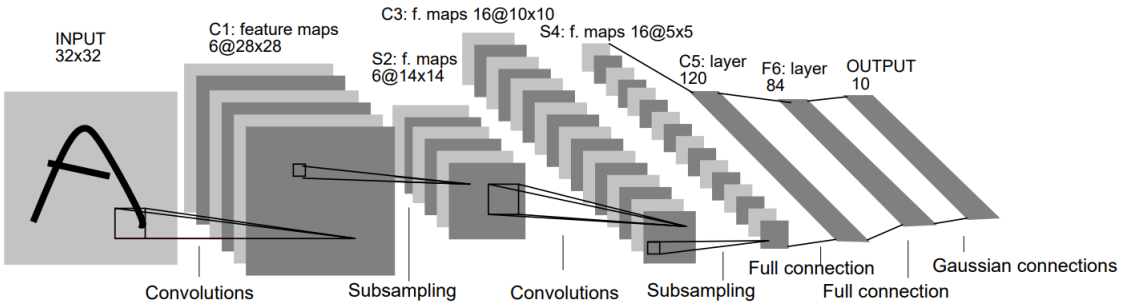


Figure 6: The LeNet architecture [6].

Let us have a look on the LeNet architecture now in Fig. 6. LeNet has 2 convolutional and 3 fully-connected layers. The convolutional layer consists of three stages. In the first one, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear unit that we saw before. In the third stage, a pooling function that replaces the output at a certain location with a summary statistic of the nearby outputs is used.

2.2.4 AlexNet

AlexNet [9] is the first deep Convolutional Neural Network. It achieved great performance (16.5% top-5 error) on ImageNet Large Scale Visual Recognition Challenge (ILSVRC) of 2012, outperforming all its competitors by more than 10% as can be seen in Fig. 7.

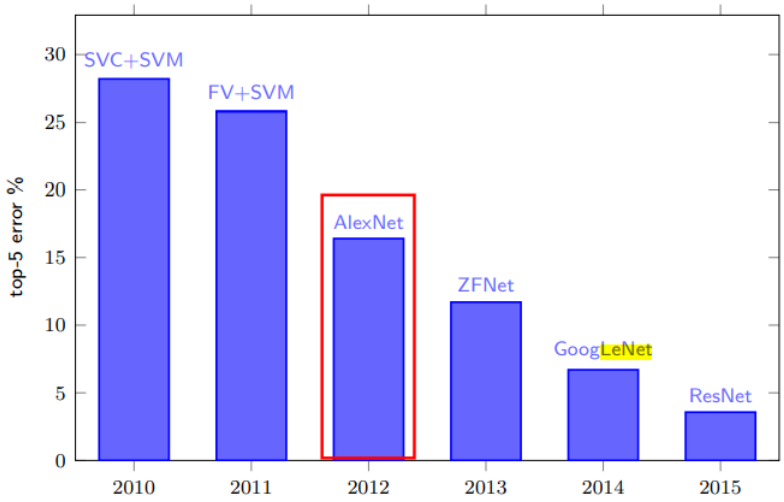


Figure 7: ILSVRC top-5 error% per year in classification task [7].

ImageNet is a dataset of over 15 millions labeled high-resolution images of around 22000 classes. ILSVRC uses a subset of ImageNet of 1.2 million training images, 50 thousands validation images and 150 thousands test images belonging to 1000 classes. Example images of ILSVRC are shown in Fig. 8.

Figure 8: Example images of ILSVRC [8].

AlexNet has 8 layers; 5 convolutional and 3 fully connected. It is the first Network to use the rectified linear unit as an activation function, as also the first to be implemented on two GPUs running in parallel. Its architecture can be seen in Fig. 9.

2.2.5 GoogLeNet (Inception v1)

Two years after AlexNet, GoogLeNet [10] was the winner of ILSVRC 2014 scoring 6.7% top-5 error. GoogLeNet motivated the need of going deeper, e.g. using more layers and thus it increased the depth to 22 layers. It achieved that while having 25 times less parameters than AlexNet because of the inception module that it used. In its naive version of Fig. 10a, the inception module is simply a feature-wise concatenation of three different convolutions and one max pooling. But this would be really expensive in terms of computation

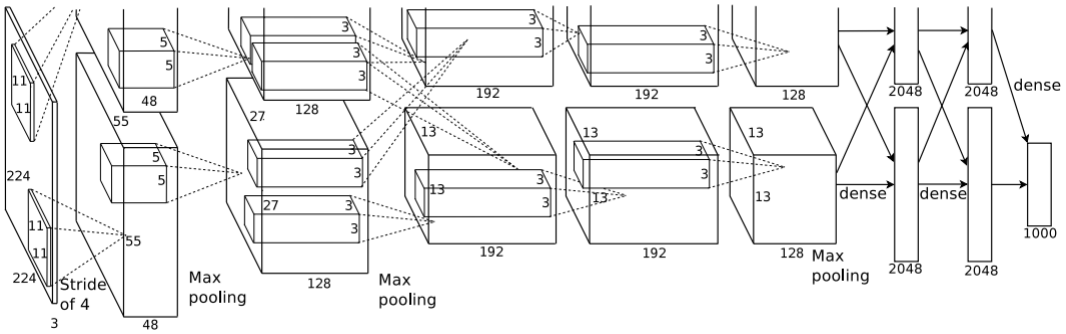
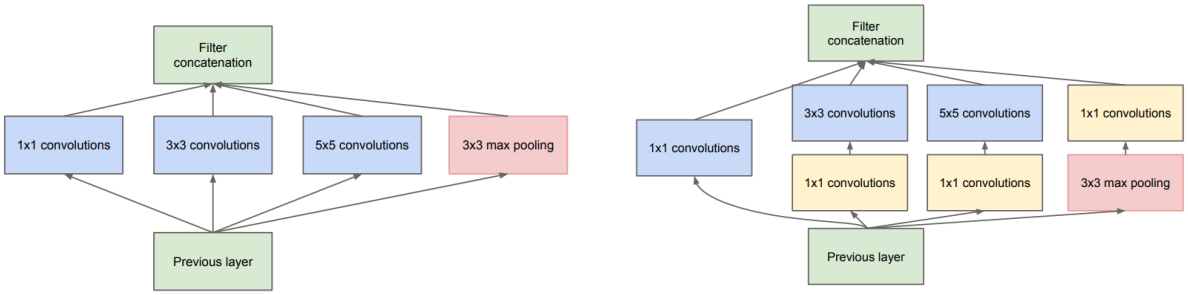


Figure 9: The AlexNet architecture [9].



(a) The naive Inception module.

(b) The Inception module with dimensionality reduction.

Figure 10: The Inception modules [10].

and would keep increasing the dimension. That is why 1 x 1 kernels were used as bottlenecks for dimensionality reduction. The inception module with dimensionality reduction can be seen in Fig. 10b.

2.2.6 BNInception (Inception v2)

The ILSVRC of 2015 was intense. Microsoft proposed PReLU-Net [40] which had an error rate of 4.94% that surpasses the human error rate of 5.1%, while few days later Google proposed a second version of the Inception (BNInception) [11] that was utilizing the batch normalization transform previously introduced from Ioffe et al. [41]. BNInception scored a top-5 error rate of 4.8%.

What exactly is batch normalization though and why do we need it? During training we estimate the mean μ and variance σ^2 of the batch as shown in Fig. 11. The input is then normalized by subtracting the mean μ and dividing it by the standard deviation σ (the epsilon ϵ is used in order to prevent denominator for being zero). The parameters γ and β are used for scale and shift in order to have a better shape and position after normalization. During testing the mean and the variance are calculated using the population. The batch normalization transform can be added to any Network to manipulate any set of activation functions. Ioffe et al. [41] propose that it enables higher learning rates, it prevents the training from getting stuck in poor local minima, it makes training more resilient to the parameter scale and regularizes the model.

BNInception was proposed in [11] along with the next version of it, Inception v3. BNInception also adopts another interesting idea, which is the factorization of 5×5 and 7×7 convolutions to two and three 3×3 convolutions respectively. This setup reduces the

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure 11: Batch Normalization applied to activation x over a batch [11].

parameter count by sharing the weight between adjacent tiles.

2.2.7 ResNet

The last CNN to be presented is ResNet [12] or even better ResNets, as it is a family of Networks. Before ResNets, the trend was to stack more and more layers going deeper and deeper, as we saw in the examples of Alexnet, GoogLeNet and BNInception. However, increasing Network depth does not work by simply stacking layers together, as there is the notorious problem of vanishing gradients. As the gradient is back-propagated from the last layers to the earlier ones, repeated multiplications may make the gradient infinitively small. This problem may not be visible in a 20-layer Network, but it is in an 56-layer Network as Fig. 12 shows. As a result, as the Network goes deeper, its performance gets saturated or even starts degrading rapidly.

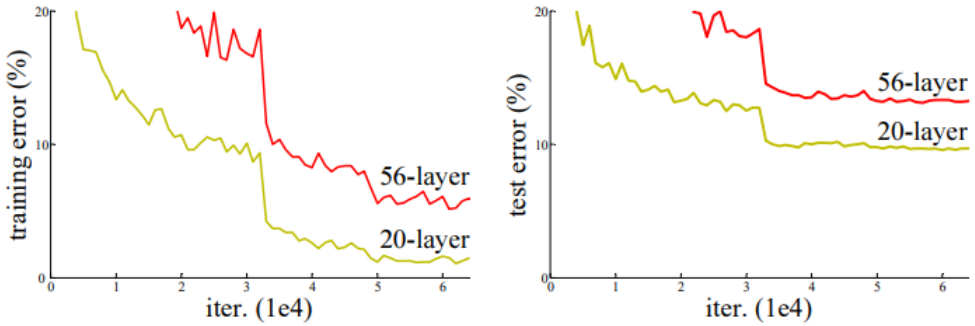


Figure 12: Training and test error of a 20-layer and 56-layer Network. Increasing Network depth leads to worse performance [12].

The core idea or ResNet is introducing the so called “identity shortcut connection” that skips one or more layers, as shown in Fig. 13.

Kaiming He et al. argue that stacking layers should not degrade the Network performance, because we could simply stack identity mappings (layers that do not do anything) upon the current Network, and the resulting architecture would perform the same. This indicates that the deeper model should not produce a training error higher than its shallower

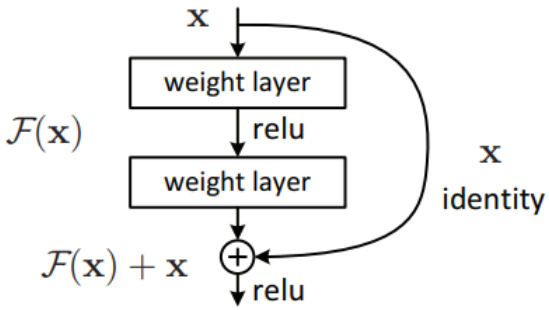


Figure 13: The residual block [12].

counterparts. They hypothesize that letting the stacked layers fit a residual mapping is easier than letting them directly fit the desired underlying mapping. And the residual block above explicitly allows it to do precisely that. An ensemble of ResNets was the winner of ILSVRC 2015 achieving 3.57% top-5 error.

2.3 Deep Metric Learning

Following the outstanding performance of CNNs in other tasks, such as classification [9] and detection [10], Song et al. [1] introduce a new setup for Metric Learning. Previous works from Bell et al. [42] and Schroff et al. [13] had already used Neural Networks to learn the nonlinear mapping from the input image to a lower dimensional and semantically meaningful embedding, but it was Song et al. that introduced the Deep Metric Learning setup that became the standard till this day.

But which are the main contributions of this paper? First of all, a new loss function is introduced that can take full advantage of every sample in the batch. Bell et al. [42] use a Siamese Network [22, 43] trained on pairs of images using Contrastive loss function [22, 23], while Schroff et al. [13] use a custom CNN named FaceNet trained on triplets using the Triplet loss function [24]. These approaches sample random pairs or triplets in order to construct the batch and compute the loss on these individuals pairs or triplets, which means that some samples within the batch are never used. Apart from that, Song et al. [1] collect a new dataset (Stanford Online Products) that together with CUB200-2011 [19] and CARS196 [20] are the datasets that are being still used in Deep Metric Learning. They also introduce a splitting setup, in which each dataset is cut in the middle for training and testing respectively. Half of the classes are used for training and the other half for testing. Finally, they propose an evaluation protocol using the F_1 and NMI metrics for clustering quality and the Recall@k scores for retrieval quality.

In this Section we will discuss about the evolution of loss functions, starting from Contrastive and ending with state-of-the-art ProxyAnchor [18], as well as about the sampling and mining methods that go with some of them.

Let us first have a quick look at the Deep Metric Learning setup. The training set of the dataset is cut into batches and fed into a CNN. The CNN learns the nonlinear mapping from each input image to a lower dimensional and semantically meaningful embedding, as shown in Fig. 14. This is done by minimizing a loss function that takes as input the embeddings and iteratively tries to push embeddings that correspond to images of the same class together and pull embeddings that correspond to images of different classes apart.

Figure 14: The general Deep Metric Learning setup [13].

Loss functions can be split into two categories: embedding losses and classification losses. Embedding losses usually work using pairs, triplets or tuples, while classification losses are based on the inclusion of a weight matrix, where each column correspond to a particular class. Of course, this separation corresponds to the implementation of the respective paper of each loss function. For example, we consider MultiSimilarity as an embedding loss function because of the fact that authors introduced it in this context. However, MultiSimilarity can be considered as a classification loss once it uses proxies. Actually, ProxyAnchor is using MultiSimilarity's equation combined with proxies and is considered a classification loss.

Typical examples of embedding losses are: Contrastive [22, 23], Triplet [13, 24], LiftedStructure [1], MultiSimilarity [14] etc, while typical examples of classification losses are: ArcFace [16], ProxyNCA [15], SoftTriple [17], etc.

2.3.1 Embedding Loss Functions

Let us formulate mathematically the aforementioned in order to take an exhaustive look at the most commonly used loss functions. Let $x_i \in \mathbb{R}^d$ be a real-value instance vector, $X \in \mathbb{R}^{m \times d}$ the corresponding instance matrix and $y \in \{1, 2, ..., C\}^m$ a label vector for the m training samples respectively. Then an input x_i is projected onto a unit sphere in a l-dimensional space by $f(\cdot; \theta) : \mathbb{R}^d \rightarrow S^l$, where f is a Neural Network parameterized by θ. We define the similarity of two samples as $S_{ij} = \langle f(x_i; \theta), f(x_j; \theta) \rangle$, where $\langle \cdot, \cdot \rangle$ denotes the dot product, resulting in an $m \times m$ similarity matrix S whose element at (i, j) is S_{ij}. We also mention that $[\cdot]_+$ indicates the Hinge function $\max(0, \cdot)$. Having said these, let us present some common embedding loss functions.

2.3.1.1 Contrastive

Hadsell et al. [23] proposed a Siamese Network, where Contrastive loss function was designed to encourage positive pairs to be as close as possible, and negative pairs to be apart from each other over a given margin λ:

$$\mathcal{L}_{\text{Contrastive}} = (1 - \mathcal{I}_{ij})[S_{ij} - \lambda]_+ - \mathcal{I}_{ij} S_{ij} \tag{2.15}$$

where $\mathcal{I}_{ij} = 1$ indicates a positive pair, while $\mathcal{I}_{ij} = 0$ indicates a negative pair.

The Siamese Network was firstly introduced by Bromley et al. in [43], but without using Contrastive as a loss function. That was done later in [22, 23]. Following the formulation and notation previously mentioned, the Siamese architecture can be seen in Fig. 15.

Let x_1, x_2 be a pair of input vectors shown to the system. Let $\mathcal{I}_{ij}$ be a binary label for the pair, as mentioned before. Let θ be the shared parameter vector that is subject to learning,

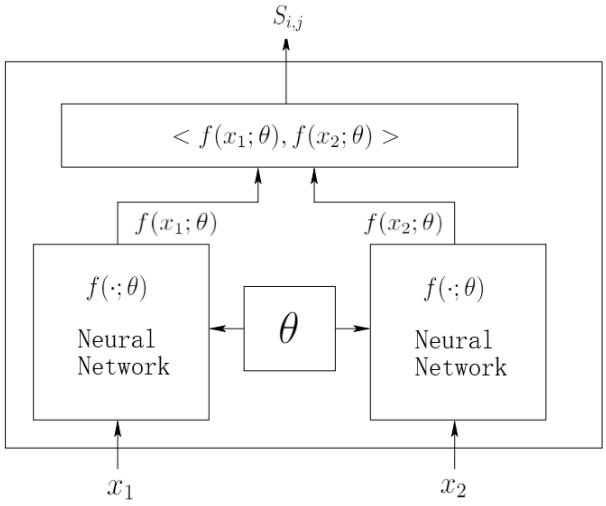


Figure 15: The Siamese Architecture.

and let $f(x_i;\theta)$ and $f(x_j;\theta)$ be two points in the l-dimensional space that are generated by mapping x_1 and x_2. In [22], the system can be viewed as a scalar "energy function" $E(x_1, x_2)$ that measures the compatibility between x_1, x_2 using the Mahalanobis distance, while in [23] the Euclidean distance is used. Here, following our formulation, we use the cosine similarity S_{ij}. The parameter θ can updated using an optimizer like the Stochastic Gradient Descent (SGD) [44] or Adam [45, 46]. The gradients can be computed by back-propagation through the loss and the two instances of $f(\cdot;\theta)$. The total gradient is the sum of the contributions from the two instances.

2.3.1.2 Triplet

In [13, 24, 47], Triplet loss was proposed to learn a deep embedding that ensures that an input vector x_i^a, which is called an anchor, is more similar to all other vectors x_i^p of the same class (positives) than it is to any other vector x_i^n of different class (negatives).

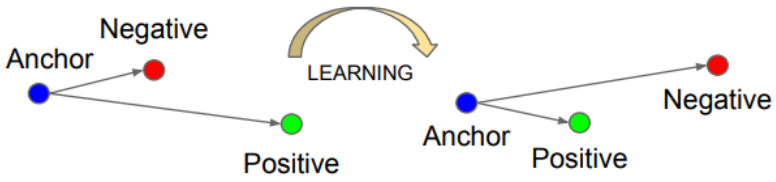


Figure 16: Triplet loss function makes the distance between an anchor and a positive smaller than the distance between this anchor and a negative [13].

Thus, we want:

$$S_{ap} > S_{an} + \lambda, \quad \forall (x_i^a, x_i^p, x_i^n) \in \mathcal{T} \tag{2.16}$$

where S_{ap} and S_{an} denote the similarity of a positive pair x_i^p and a negative pair x_i^n with an anchor x_i^a respectively. λ is a margin that is enforced between positives and negatives, and $\mathcal{T}$ is the set of all possible triplets in the training set. The corresponding loss is:

$$\mathcal{L} = [S_{an} - S_{ap} + \lambda]_+ \tag{2.17}$$

Generating all the possible triplets would result in many triplets that easily fulfil the constraint of Eq. 2.16. These triplets would not contribute to training, as their gradients would be really small or even zero, and thus they would result in slower convergence. Choosing triplets that violate the triplet constraint of Eq. 2.16 is crucial for improving the model.

How exactly do we select these triplets, though? Given an anchor x_i^a we would want to select a positive x_i^p that is called a hard positive, such that:

$$\arg\min_{x_i^p} \langle f(x_i^a), f(x_i^p) \rangle \tag{2.18}$$

and similarly a hard negative x_i^n, such that:

$$\arg\max_{x_i^n} \langle f(x_i^a), f(x_i^n) \rangle \tag{2.19}$$

This is called hard mining and there are two ways to generate these hard triplets. The first way is to generate them offline every n steps, using the most recent Network checkpoint. The second way is to generate them online by selecting them from within the batch. Selecting the hardest samples can lead to bad local minima early on during training or even a collapsed model, i.e. all images have the same embedding. In order to mitigate this, Scroff et al. propose a semi-hard negative mining such that:

$$n_{ap} = \arg\max_{n: S_{ap} > S_{an}} S_{an}, \tag{2.20}$$

where n_{ap} is the mined semi-hard negative. This yields a violating sample that is fairly hard but not too hard.

2.3.1.3 LiftedStructure

Song et al. propose a loss function that takes full advantage of each sample within the batch by "lifting the vector of pairwise distances to the matrix of pairwise distances". This is probably the greatest advantage of this loss function and is explicitly explained in Fig. 17.

Following our notation and formulation, the LiftedStructure loss function can be written as:

$$\mathcal{L}_{\text{LiftedStructure}} = \sum_{i=1}^{m} \left[\log \sum_{y_k = y_i} e^{\lambda - S_{ik}} + \log \sum_{y_k \neq y_i} e^{S_{ik}} \right]_+ \tag{2.21}$$

where λ is a fixed margin. Taking full advantage of the training batch might have a drawback: randomly sampled negative pairs might carry limited information. In order to fight this, Song et al. mine a few positive pairs at random and then actively add their difficult neighbors to the batch. This is further explained in Fig. 18.

2.3.1.4 NPair

Highlighting the drawback of triplet embedding that was illustrated in Fig. 17, Sohn [48] proposes a loss function that recruits multiple negatives for each anchor. He points out

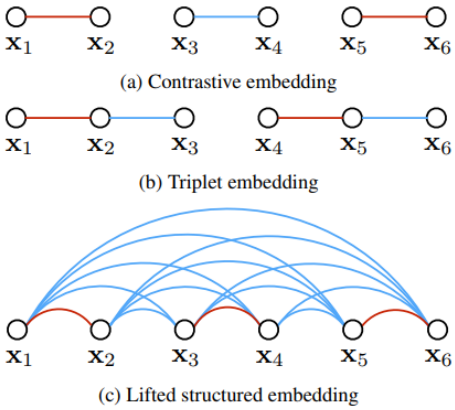


Figure 17: Illustration of a training batch with six samples $x_1, ..., x_6$. Red edges represent positives, while blue edges represent negatives. Contrastive loss function works using separate pairs, thus x_1 and x_2 represent a positive pair, x_3 and x_4 represent a negative pair, etc. Triplet loss function works using separate triplets, thus x_2 represents an anchor that has a positive x_1 and a negative x_3, etc. In general, a sample x_i of the training batch cannot be used more than once when computing both the Contrastive and Triplet loss function. In contrast, LiftedStructure takes into account all pair wise samples within the batch, thus x_1 and x_2 represent a positive pair, while at the same time x_1 and x_3 represent a negative pair [1].

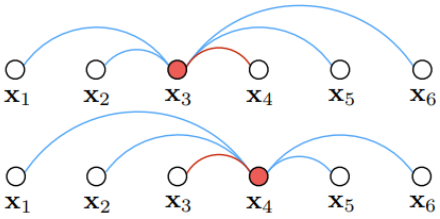


Figure 18: In this illustration with 6 examples, x_3 and x_4 represent a randomly sampled positive pair that independently compares against all other negative pairs in order to mine the hardest one [1].

that during an update Triplet loss function compares one anchor with one negative while ignoring negatives from other classes. As a consequence, the embedding vector for a sample is only guaranteed to be far from the selected negative class but not necessarily the others. The hope is that after looping over sufficiently many randomly sampled triplets, the final similarity will be balanced correctly. Of course, a way to fight this is using hard mining in order to select samples that violate the triplet constraint, but this can be proven to be expensive. Sohn proposes NPair loss function, which compares an anchor with multiple negatives in order to make sure that it is distinguishable from all of them at the same time. Following our formulation and notation, NPair loss function can be written as:

$$\mathcal{L}_{\text{NPair}} = \frac{1}{m}\sum_{i=1}^{m}\log\left(1 + \sum_{y_k \neq y_i, y_j = y_i} e^{S_{ik} - S_{ij}}\right) \tag{2.22}$$

2.3.1.5 Margin

Margin loss function is introduced in [26] as a simple extension to Contrastive loss. Wu et al. highlight the significance of selecting training examples and propose a distance weighted sampling that selects informative and stable examples. They show that sampling might play equal or even more import role than the loss function, as different sampling

strategies can lead to drastically different solutions for the same loss function. Margin loss function is defined as:

$$\mathcal{L}_{\text{Margin}} = [\lambda + \mathcal{I}_{ij}(\beta - S_{ij})]_+, \tag{2.23}$$

where β is a variable that determines the boundary between positive and negative pairs, λ controls the margin of separation, $I_{ij} = 1$ indicates a positive pair, while $I_{ij} = 0$ indicates a negative one.

2.3.1.6 MultiSimilarity

Wang et al. [14] motivate the design of their loss function by establishing a General Pair Weighting (GPW) framework, which casts the sampling task into a unified view of pair weighting through gradient analysis. They express various existing embedding loss functions using the GPW, compare them and make significant assumptions about their differences and limitations. For example, by computing the partial derivative with respect to S_{ij} of Eq. 2.15 they find that all positive pairs and hard negative pairs with $S_{ij} > \lambda$ are assigned with an equal weight. Equal weights are also assigned by Triplet loss function on valid triplets that fulfil the triplet constraint 2.7. They observe that most of the embedding loss functions weight the pairs based on either self cosine similarities or relative similarities compared with other pairs.

They define three different types of similarity:

- S: Self-similarity, which is the similarity computed from the pair itself. Contrastive loss function is based on this similarity.
- N: Negative relative similarity, which is computed by considering the relationship from neighboring negative pairs. LiftedStructure loss function is based on this relative similarity.
- P: Positive relative similarity, which is computed by considering the relationship from neighboring positive pairs. Triplet loss function is based on this relative similarity.

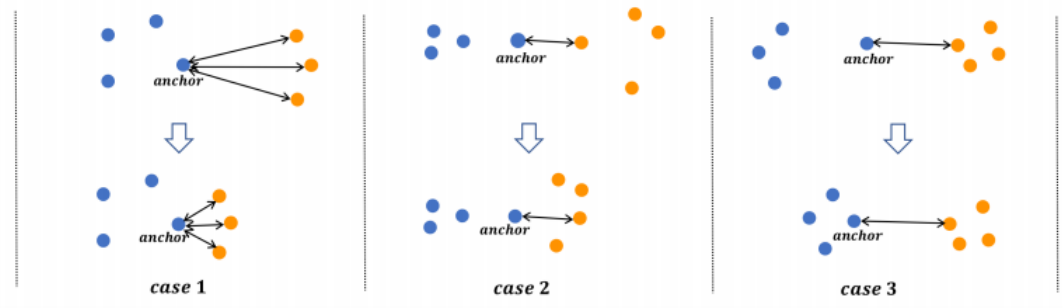


Figure 19: The three types of similarity [14].

Illustration of these three different types of similarity can be seen in Fig. 19. A negative pair is taken as an example to explain them. Blue and yellow dots represent two different classes. The anchor is always the same. Case 1 represents the S:Self-similarity. We examine the similarity between the anchor and the three negatives. As these three negatives come closer to the anchor, the cosine similarity is increasing. Case 2 represents

Table 1: The types of similarity each embedding loss function utilizes. MultiSimilarity takes advantage of all the similarities.

	Contrastive	Triplet	LiftedStructure	NPair	MultiSimilarity
S	✓	✗	✗	✗	✓
N	✗	✗	✓	✓	✓
P	✗	✓	✗	✗	✓

the N:Negative relative similarity. We examine the similarity between the anchor and the negative, but this time the negative itself is not moving. The samples with the same class as the negative are moving and especially they are coming closer to the negative and closer to each other. Thus the cosine similarities of the yellow class is increasing and thus the relative similarity of the anchor with them is decreasing. Case 3 represents the opposite of Case 2: Positive relative similarity. We examine again the similarity between the anchor and the negative. The negative itself is not moving, but this time the positives of the anchor are moving and especially they are coming closer to the anchor and closer to each other. Thus the cosine similarity of the blue class is increasing and thus the relative similarity of the anchor with the negatives is decreasing.

Having said these, Wang et al. analyze various existing embedding loss functions and propose a new one called MultiSimilarity that takes advantage of all the three types of similarity. Table 1 shows the types of similarity that each loss function we previously presented utilize.

Following our formulation and notation, Eq. 2.24 shows MultiSimilarity loss:

$$\mathcal{L}_{\text{MultiSimilarity}} = \frac{1}{m}\sum_{i=1}^{m}\left\{\frac{1}{\alpha}\log\left[1+\sum_{k\in\mathcal{P}_i}e^{-\alpha(S_{ik}-\lambda)}\right]+\frac{1}{\beta}\log\left[1+\sum_{k\in\mathcal{N}_i}e^{\beta(S_{ik}-\lambda)}\right]\right\}, \quad (2.24)$$

where α, β, λ are hyper-parameters, $\mathcal{P}_i$ and $\mathcal{N}_i$ are the sets of positives and negatives respectively.

2.3.2 Classification Loss Functions

As mentioned in 2.3, loss functions can be split into two categories: embedding losses and classification losses. The ones presented so far belong to the first category. These losses enjoy rich and fine-grained data-to-data relations, but constructing all the possible tuples (pairs or triplets) may be expensive in terms of computational complexity and may lead to slow convergence. In addition, a large amount of tuples are not informative and sometimes even degrade the quality of the learned embedding space. To address this issue, most embedding losses entail some kind of mining in order to select and utilize tuples that will contribute the most to training. However, these techniques involve hyper-parameters that have to be tuned carefully and may increase the risk of overfitting. Classification losses resolve this complexity by adopting some kind of proxies, which enable faster and more reliable convergence. A proxy is a learnable representative of a subset of training data. In general, classification losses also demand less hyper-parameter fine-tuning, enable faster convergence and are more robust against label noises and outliers. We will start our journey on classification losses from the well-known SoftMax loss, as it has somehow inspired all the other ones we are going to present.

2.3.2.1 SoftMax

Extending our notation for classification losses, let us denote the embedding of the i-th sample as x_i and the corresponding label as y_i, then the conditional probability output by a Neural Network can be estimated via SoftMax operator:

$$Pr(Y = y_i|x_i) = \frac{e^{w_{y_i}^T x_i}}{\sum_j^C e^{w_j^T x_i}}, \tag{2.25}$$

where $\{w_1, ..., w_C\} \in \mathbb{R}^{d \times C}$ is the last fully connected layer, C denotes the number of classes and d is the dimension of embeddings. The corresponding SoftMax loss is:

$$\mathcal{L}_{\text{SoftMax}} = -\log \frac{e^{w_{y_i}^T x_i}}{\sum_j e^{w_j^T x_i}} \tag{2.26}$$

2.3.2.2 ProxyNCA

Movshovitz-Attias et al. motivate the use of proxies by highlighting the aforementioned fact that when using embedding losses only a specific subset of all possible tuples is taken into consideration when computing the loss. They propose to learn a small set of data points (the set of proxies) that is way smaller than the whole training set, but still a great approximation of it, so that Triplet loss over the proxies is a tight upper bound of the original loss. To make it more clear, they propose to optimize Triplet loss function, but on a different space of triplets consisting of an anchor data, a learnable positive proxy and a learnable negative proxy. These proxies serve as a concise representation for each semantic concept and by this way less triplets are formed, but ideally without losing any semantic information. This is further explained in Fig. 20.

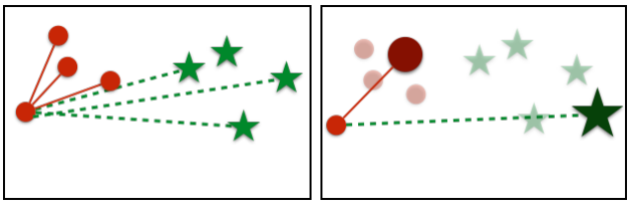


Figure 20: Red circles and green stars represent points of two different classes respectively. There are 48 triplets that can be formed from these instances. Learning a proxy for each class results in only 8 comparisons [15].

ProxyNCA loss assigns a proxy to each class, so that the number of proxies is the same with the number of classes C. Given an input embedding vector $x_i \in \mathbb{R}^d$ with label y_i as an anchor, the proxy of the same class of the input is regarded as positive and the other proxies are regarded as negatives. Let $w_j \in \mathbb{R}^d$ be a real-value weight (proxy) vector. The loss can then be written as:

$$\mathcal{L}_{\text{ProxyNCA}} = \sum_{i=1}^{m} -\log \frac{e^{w_{y_i}^T x_i}}{\sum_{j \neq y_i} e^{w_j^T x_i}}, \tag{2.27}$$

The gradient of ProxyNCA loss with respect to $w^T x$ is given by:

$$\frac{\partial \mathcal{L}}{\partial (w^T x)} = \begin{cases} -1, & if w = w_{y_i} \\ \dfrac{e^{w^T x_i}}{\sum\limits_{j \neq y_i} e^{w_j^T x_i}}, & otherwise \end{cases} \tag{2.28}$$

Eq. 2.28 shows that minimizing the loss encourages the anchor to be close to the positive proxy, but far away from negative proxies. The anchor and the positive proxy are pulled together by a constant force, while negative proxies that are closer to the anchor are more strongly pushed away.

ProxyNCA enables faster convergence thanks to its low training complexity, $O(mC)$, where m is the number of training samples and C the number of classes. This complexity is way lower than the $O(m^2)$ complexity of pair-based embedding losses or $O(m^3)$ complexity of triplet-based ones, since $C \ll m$. Proxies are also more robust against outliers since they are trained to represent groups of data. However, since the loss associates each anchor only with proxies, it can not exploit fine-grained data-to-data relations.

2.3.2.3 ArcFace

Jiankang et al. [16] highlight that the traditional SoftMax loss of Eq. 2.26 does not explicitly optimise the feature embedding to enforce higher similarity for intra-class samples and diversity for inter-class samples, which results in a performance gap. They transform the logit $w_j^T x_i$ to $\|w_j\|\|x_i\| \cos\theta_j$, where θ_j is the angle between the weight w_j and the feature x_i. They fix the individual weight $\|w_j\| = 1$ by l_2 normalization and the embedding feature $\|x_i\|$ by l_2 normalization and rescale it to s. The normalization step on features and weights makes the predictions only depend on the angle between the feature and the weight. The learned embedding features are thus distributed on a hypersphere with a radius of s. They also add an additive angular margin λ between x_i and w_{y_i} to simultaneously enhance the intra-class compactness and inter-class discrepancy. ArcFace loss can then formulated as:

$$\mathcal{L}_{\text{ArcFace}} = \sum_{i=1}^{m} -\log \frac{e^{s\cos(\theta_{y_i}+\lambda)}}{e^{s\cos(\theta_{y_i}+\lambda)} + \sum_{j \neq yi} e^{s\cos\theta_j}} \tag{2.29}$$

ArcFace is proposed to improve the discriminative power of models and to stabilise the training process. Its main idea is the fact that the dot product between the CNN feature and the last fully connected layer is equal to the cosine distance after feature and weight normalisation. This is further illustrated and explained in Fig. 21.

2.3.2.4 SoftTriple

Qian et al. [17] highlight the fact that optimizing using embedding losses that utilize only an anchor and its neighbors (e.g. the active set of pairs or triplets) is sub-optimal, as the samples in a batch may not be able to capture the overall neighborhood well, especially for relatively large datasets. In contrast, SoftMax loss, which does not work this way, seems to perform well on similar distance based tasks. This motivates them to investigate the formulation of SoftMax loss. Their analysis demonstrates that SoftMax loss is equivalent

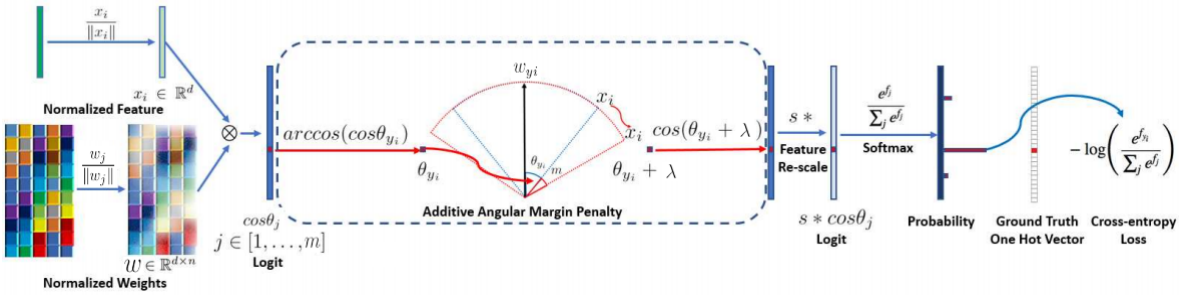


Figure 21: We first normalize the feature x_i and weight w. Then we get the $\cos\theta_j$ logit for each class as $w_i^T x_i$. We calculate the $\arccos\theta_{y_i}$ and get the angle between the feature x_i and the ground truth weight w_{y_i}. w_j can be seen as a proxy for each class. Then, we add an angular margin penalty λ on the target angle θ_{y_i}. We calculate $\cos(\theta_{y_i}+\lambda)$ and multiply all logits by the feature scale s. The logits finally go through the SoftMax and CrossEntropy. [16]

to a smooth Triplet loss: by providing a single proxy for each class in the last fully connected layer, the triplet constraint derived by SoftMax loss can be defined on an anchor, its corresponding proxy and a proxy from a different class. However, since a class in a real-world data can consist of multiple local clusters, a single proxy might not be able to capture the inherent structure of data and thus they propose the use of multiple proxies as shown in Fig. 22.

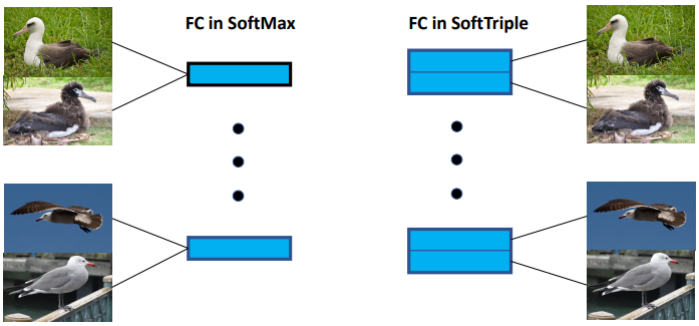


Figure 22: In SoftMax loss, each class has only one corresponding proxy. Samples of the same class will be collapsed to the same proxy no matter their possible variance (pose, color, viewpoint, etc.). In contrast, SoftTriple keeps multiple proxies and thus is more capable of modeling the intra-class variability, as these samples will be assigned to different proxies. [17]

SoftTriple loss can be written as:

$$\mathcal{L}_{\text{SoftTriple}} = \sum_{i=1}^{m} -\log \frac{e^{\alpha(w_{y_i}^T x_i - \lambda)}}{e^{\alpha(w_{y_i}^T x_i - \lambda)} + \sum_{j \neq y_i} e^{\alpha w_j^T x_i}}, \tag{2.30}$$

where α is a scaling factor and λ a fixed margin. Compared with other losses, the number of triplets in SoftTriple is linear in the number of anchors and SoftTriple can learn the embeddings without any sampling phase to be needed; just a mild increase in the size of the last fully connected layer which corresponds to proxies. Apparently, SoftTriple has to determine the number of proxies for each class. The strategy authors propose is to set a sufficiently large number of proxies at the beginning and then to decrease it applying the $L_{2,1}$ norm.

2.3.2.5 ProxyAnchor

Kim et al. [18] underline the advantages and disadvantages of embedding and classification losses and then propose a novel loss function called ProxyAnchor, which takes good points of both of them while correcting their defects. Unlike the other proxy-based losses, ProxyAnchor utilizes each proxy as an anchor and associates it with all samples in a batch. Specifically, for each proxy, the loss aims to pull samples of the same class closer and to push samples of other classes away. On the one hand, as a classification loss, it demands no hyper-parameter for tuple sampling, it has fast convergence and is more robust against noisy labels and outliers. On the other hand, it can take data-to-data relations into account by associating all data in a batch with each proxy so that the gradients with respect to a sample are weighted by its relative proximity to the proxy affected by other data in the batch. ProxyAnchor loss is formulated as:

$$\mathcal{L}_{\text{ProxyAnchor}} = \frac{1}{|W^+|} \sum_{w \in W^+} \log \left(1 + \sum_{x \in X_w^+} e^{-\alpha(w^T x - \lambda)} \right) + \frac{1}{|W|} \sum_{w \in W} \log \left(1 + \sum_{x \in X_w^-} e^{\alpha(w^T x + \lambda)} \right), \tag{2.31}$$

where $\lambda > 0$ is a margin, $\alpha > 0$ is a scaling factor, W indicates the set of all proxies, W^+ denotes the set of positive proxies in the batch. For each proxy w, a batch of embedding vectors X is divided into two sets: X_w^+, the set of positive embedding vectors of w and X_w^-, the set of negative embedding vectors of w. It is easy to notice that the loss aims to pull w and its most dissimilar positive sample (i.e. hardest positive) closer, and to push w and its most similar negative samples (i.e. hardest negative) apart. The formulation of this loss looks a lot like Eq. 2.24. Indeed, due to the nature of Log-Sum-Exp, it pulls and pushes embedding vectors with different degrees of strength that are determined by their relative hardness. That's something that makes this loss different from ProxyNCA, since the gradient of Eq. 2.32 showed that the anchor and the positive proxy are pulled together by a constant force when using ProxyNCA. Having said these, let us take a look at the gradient of ProxyAnchor with respect to $w^T x$:

$$\frac{\partial \mathcal{L}}{\partial (w^T x)} = \begin{cases} \dfrac{1}{|W^+|} \dfrac{-\alpha(w^T x - \lambda)}{1 + \sum\limits_{x' \in X_w^+} e^{-\alpha(w^T x' - \lambda)}} & for\, x \in X_w^+ \\ \dfrac{1}{|W|} \dfrac{\alpha(w^T x + \lambda)}{1 + \sum\limits_{x' \in X_w^-} e^{\alpha(w^T x' + \lambda)}} & otherwise \end{cases} \tag{2.32}$$

This gradient is not only affected by the anchor, but also other embedding vectors in the batch; the gradient becomes larger when the anchor is harder than the others. In this way, ProxyAnchor loss enables embedding vectors in the batch to interact with each other and reflects their relative hardness through the gradients, which helps enhance the quality of the learned embedding space.

Finally, Kim et al. present a comparison between some popular loss functions, which is considered important to be presented. This is illustrated in Fig. 23.

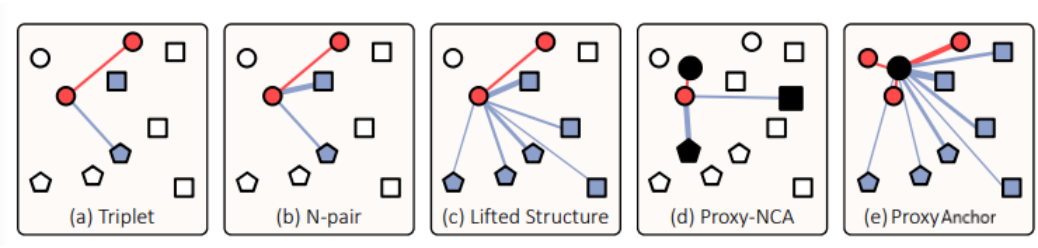


Figure 23: Nodes represent different samples in a batch. Different shapes represent different classes, black nodes represent proxies, red nodes represent positives, blue nodes represent negatives. The associations defined by the losses are expressed by edges and thicker edges get larger gradients. (a) Triplet loss associates each anchor with a positive and a negative without considering their hardness. (b) NPair loss and (c) LiftedStructure loss reflect hardness of data, but do not utilize all data in the batch. (d) ProxyNCA loss cannot exploit data-to-data relations since it associates each data point only with proxies. (e) ProxyAnchor handles entire data in the batch, and associates them with each proxy with consideration of their relative hardness determined by data-to-data relations. See the text for more details [18].

3. EXPERIMENTAL SETUP

This Chapter is dedicated to experimental setup and is organized as follows: we first present the datasets, Networks and evaluation protocol used in Deep Metric Learning, we underline the implementation details of our experiments and highlight the issues that led us conducting them.

3.1 Datasets

Let us have a look at the Deep Metric Learning datasets: CUB200-2011, CARS196 and SOP.

Figure 24: Random images of CUB200-2011 dataset [19].

The CUB200-2011 dataset [19] has 200 classes of birds with 11788 images. Following the setup of [1], the 100 classes (5864 images) are used for training and the rest of classes (5924 images) for testing. Example images of CUB200-2011 dataset can be seen in Fig. 24

Figure 25: Random images of CARS196 dataset [20].

The CARS196 dataset [20] has 196 classes of cars with 16185 images. Following the setup of [1] again, the 96 classes (8054 images) are used for training, while the other 96 classes (8131 images) for testing. Example images of CARS196 dataset can be seen in Fig. 25

The SOP dataset was collected from [1] and has 22634 classes of online products with 120053 images. This means that each product has approximately 5.3 images. Following

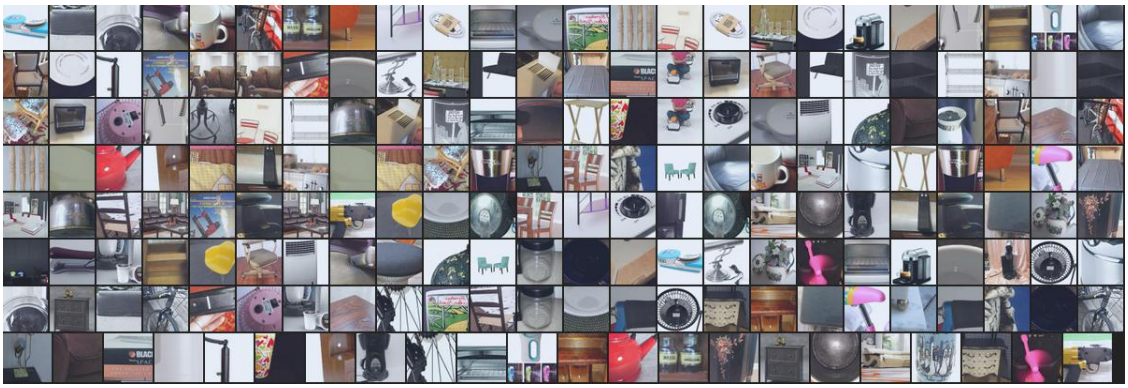


Figure 26: Random images of SOP dataset.

the same setup again, the 11318 classes (59551 images) are used for training, while the other 11316 classes (60502 images) for testing. Example images of SOP dataset can be seen in Fig. 26

3.2 Networks

Before delving into the details of our experiments, let us first have a look at the Networks and the embedding sizes that are used in papers. In [22], Contrastive is using both a 2-layer CNN and five 5-layer CNNs. The exact architecture of the first one is not mentioned, so the embedding size is not known, while the embedding size of the second ones is set to 50. A 2-layer CNN is also used in [23], but the embedding size is set to 2, so that the Contrastive embeddings can be easily visualized. We implement the Triplet loss function with semi-hard negative mining as introduced in [13], where a 22-layer CNN based on [49] and [50] is used, along with some GoogLeNet variations. It is important to mention that both Contrastive and Triplet papers do not report any results on the Deep Metric Learning datasets (CUB200-2011, CARS196, SOP), as these papers were published before the Deep Metric Learning problem formulation. Experiments using Contrastive and Triplet on these datasets were later conducted by papers like [1].

LiftedStructure reports F_1 and NMI scores using embedding sizes of 64, 128, 256, 512 and 1024, but reports Recall@k using only embedding size of 64. Once Recall@k is the metric we use to evaluate our experiments, we only consider the last size. NPair uses GoogLeNet with a 64-dimensional embedding for CUB200-2011 and CARS196 experiments and a 512-dimensional embedding for SOP. ProxyNCA mentions that it uses the Inception architecture with batch normalization, but without citing the corresponding paper of this Network [11]. Instead, it cites the previous papers [10] and [41] and thus is not clear if the BNInception architecture or a variant of it is used. Margin uses ResNet50 with an embedding size of 128, ArcFace uses both ResNet50 and ResNet100 with an embedding size of 512. Finally, MultiSimilarity and SoftTriple use BNInception with both a 64 and a 512-dimensional embedding, while ProxyAnchor uses the same Network but only with a 512-dimensional embedding. The Networks and the embedding sizes used in the papers can be seen in Table 2.

We conduct extensive experiments on all 3 datasets of Deep Metric Learning (CUB200-2011, CARS196, SOP) using the most common architectures (GoogLeNet, BNInception, ResNet50) and 10 different loss functions (Contrastive, Triplet, LiftedStructure, NPair, ProxyNCA, ArcFace, Margin, MultiSimilarity, SoftTriple, ProxyAnchor). These loss func-

Table 2: The Networks and embedding sizes used in respective papers.

Loss Function	Network	Embedding Size
Contrastive	2-layer CNN [22, 23], 5-layer CNN [22]	2 [23], 50 [22]
Triplet	22-layer CNN, GoogLeNet	128
LiftedStructure	GoogLeNet	64
NPair	GoogLeNet	64, 512
ProxyNCA	BNInception	64
Margin	ResNet50	128
ArcFace	ResNet50, ResNet100	512
MultiSimilarity	BNInception	64, 512
SoftTriple	BNInception	64, 512
ProxyAnchor	BNInception	512

tions cover a wide range from older to the state-of-the-art. Some of them are embedding loss functions (i.e. pair-based or triplet-based), while other are classification ones (proxy-based). We also use 4 different embedding sizes (64, 128, 512, 1024) for each experiment.

3.3 Evaluation Protocol

We evaluate our models using the Recall@k metric [51], which shows the retrieval quality. We first compute the embeddings of every image in the test set. Each test image (query) first retrieves k nearest neighbors from the test set and receives score 1 if an image of the same class is retrieved among the k nearest neighbors, otherwise 0. Recall@k averages this score over all the images of the test set. For example, when calculating the Recall@4, we first retrieve the 4 nearest neighbors of the query and then we assign score equal to 1 if one image of the same class is retrieved among these 4 neighbors. This is done iteratively for all the queries and is finally averaged over all the images of the test set.

3.4 Implementation Details

In order to have an unbiased evaluation of Deep Metric Learning methods, we conduct our experiments under the same conditions and following the same pipeline. We want to make sure that no method is favored. We train our models for 100 epochs, which is found to be enough for convergence. We use the AdamW [52] variant of Adam as an optimizer and the StepLR as a scheduler. StepLR decays the learning rate of each parameter group by gamma every step size epochs. The learning rate and the scheduling are taken from papers once they are available. In case they are not, we conduct a small search around the default values. The default values that were used for each experiments are shown in Table 3.

The hyperparameters of loss functions like margins, scales, weights learning rates etc. are taken from papers, as we assume that authors have made an exhaustive search in order to optimize their values. Table 4 presents the hyperparameters that were used for each loss function. We use a batch size of 100 for ResNet50 runs and a batch size of 180 for GoogLeNet and BNInception runs. Concerning sampling, when a method implies using balanced sampling like [14], we sample 5 images per class. Otherwise, we use random

Table 3: The default values that were used for each experiment. The initial learning rate shown in second column is multiplied by gamma after step size epochs. The same procedure is repeated for the resulting learning rate etc.

Experiment	Learning Rate	Step Size	Gamma
CUB200-2011 ResNet50	0.0001	5	0.1
CUB200-2011 BNInception	0.0001	10	0.1
CUB200-2011 GoogLeNet	0.0001	10	0.1
CARS196 ResNet50	0.0001	10	0.1
CARS196 BNInception	0.0001	20	0.1
CARS196 GoogLeNet	0.0001	20	0.1
SOP ResNet50	0.0006	10	0.25
SOP BNInception	0.0006	20	0.25
SOP GoogLeNet	0.0006	20	0.25

sampling. Concerning mining, when a method comes with a specific type of mining like [26] with distance weighted, we use this mining. Otherwise, we use random mining. Table 5 presents the mining methods used in our experiments for each loss function. All the experiments are made using either the NVIDIA V100 or the NVIDIA GeForce RTX 2080 Ti.

3.5 Issues

Why do we make these experiments, though?

3.5.1 Unfair Comparisons

First of all, while delving into the bibliography of Deep Metric Learning, we noticed that a lot of papers seem to introduce new loss functions that outperform the existing ones, but the comparisons are not made fairly. For example, some papers use a better Network than their competitors. As long as the Networks are pretrained on ImageNet and then finetuned on smaller datasets, the choice of the Network is important, as the initial accuracy on the smaller datasets varies depending on the chosen Network. Apart from that, shallower Networks like GoogLeNet will probably have less discriminative power than deeper Networks like ResNet50 and thus their embeddings are more likely to be less powerful or semantically meaningful. Let us highlight some examples of unfair comparisons concerning the choice of the Network:

- Chao-Yuan Wu et al. [26] make experiments using ResNet50 and compare their Margin Loss with other losses that use GoogLeNet.
- Wang et al. [14] use BNInception and claim better performance that the ensemble methods that use GoogLeNet.
- Cakir et al. [53] use ResNet50, while their competitors use either GoogLeNet or BNInception. They also claim better performance than the ensemble methods that use GoogLeNet.
- Qian et al. [17] use BNInception and compare with N-Pair [48] and HDC [54] that use GoogLeNet.

Table 4: The hyperparameters used in our experiments.

Loss Function	Hyperparameter	Value
Contrastive	margin λ	0.5
Triplet	margin λ	0.1
LiftedStructure	margin λ	0.5
NPair	l_2	0.02
ProxyNCA	proxy lr	0.00001
Margin	margin λ beta beta lr	0.5 1.2 0.00005
ArcFace	margin λ scale s weights lr	28.6 64 0.0001
MultiSimilarity	margin λ scale α scale β epsilon	0.5 2 50 0.1
SoftTriple	margin λ scale α weights lr gamma tau	0.1 20 0.0001 10 0.2
ProxyAnchor	margin λ scale α	0.1 32

Table 5: The mining methods used in our experiments.

Loss Function	Mining Method
Contrastive	-
Triplet	semi-hard
LiftedStructure	hard
NPair	-
ProxyNCA	-
Margin	distance weighted
ArcFace	-
MultiSimilarity	hard
SoftTriple	-
ProxyAnchor	-

- Yu et al. [55] use ResNet50, while their competitors use either GoogLeNet or BNInception. They also claim better performance than the ensemble methods that use GoogLeNet.

Using an embedding of higher dimensionality will probably lead to an increased accuracy. Comparing methods that use different embedding sizes cannot be considered fair. Let us also highlight some examples of unfair comparisons concerning the dimensionality of the embedding:

- Chao-Yuan Wu et al. [26] use an embedding size 128, while more than half of their competitors use size 64.
- Wang et al. [56] use size 512, while most of its competitors use size 64.
- Yu et al. [55] use size 512, while the only competing architecture that uses the same architecture, uses size 128.

Moreover, some unfair comparisons have to do with details that are crucial, but are either not highlighted or omitted at all. For example, MultiSimilarity loss freezes batch normalization parameters in their official code, but this is not mentioned in the paper. ProxyAnchor loss, which has a superior performance, uses the sum of Global Average Pooling (GAP) and Global Max Pooling (GMP), but this is also not mentioned in the paper. Additionally, some official implementations of papers tend to use greater learning rate for the fully connected layer than the rest of the Network. Finally, there are some cases that claim to do a simple 256 resize and 227 or 224 random crop, but in their code are actually doing more sophisticated and advanced methods like the RandomResizedCrop method of PyTorch [57]. Having mentioned these, it is somehow clear that the apparent hype of last years is to be questioned. How can someone be sure that Deep Metric Learning has an increasing tendency, when the comparisons are not made fairly?

3.5.2 Lack of Validation Set

Apart from that, the general setup of Deep Metric Learning missing a validation set should also be questioned. As already mentioned, this setup was introduced by [1] and it is still the standard one. This setup implies splitting each dataset so that half of the classes are used for training, while the other half are used for testing. During training, the test accuracy of the model is checked at regular intervals and the model with the best test accuracy is finally chosen. This means that the hyperparameter tuning and the model selection are done with direct feedback from the test set. This breaks one of the most basic rules of Machine Learning. Some papers do not even check performance at regular intervals, but instead report accuracy after training for a predetermined number of iterations, which is unclear how exactly is chosen. Training with test set feedback is a bad practice and thus it is clear that this setup has to be further studied and questioned.

3.5.3 Benchmark and Ablation Study

We make these experiments in order to create a real benchmark. A benchmark including all the Networks, all the datasets and a lot of loss functions starting from the classic

ones and ending with the state-of-the-art. We aim to make an ablation study using this benchmark that will give us some deeper insight into Deep Metric Learning.

Our experiments are expected to answer questions like:

- how difficult or how stable is the training process without a validation set?
- is there a true convergence using this setup or most of the methods just report the best accuracy found on test set, which is just a random peak?
- does this setup gives models that can truly generalize well?
- are the hyperparameters found using this setup the optimal?

The last 2 questions cannot be answered only by these experiments, because of the fact that they also need a comparison with other setups. We will later see that indeed we compare these results with the results of another setup that includes a validation set.

4. EXPERIMENTAL RESULTS AND DISCUSSION

This Chapter is dedicated to experimental results and is organized as follows: we present all the CUB200-2011 results, as also the CARS196 and SOP results using BNInception. We then make a discussion about our findings: Networks, embeddings, datasets, loss functions and finally about the setup.

4.1 Results

4.1.1 CUB200-2011 ResNet50

Table 6: CUB200-2011 ResNet50 experiments.

(a) embedding size = 64.

	R@1	R@2	R@4	R@8
Contrastive	60.28	71.49	80.77	87.07
Triplet	57.56	69.62	80.22	87.44
LiftedStructure	58.36	70.41	79.25	87.20
NPair	57.28	68.54	78.92	87.29
ProxyNCA	60.25	71.51	80.71	87.68
Margin	59.66	71.10	81.06	88.40
ArcFace	58.32	69.23	78.38	85.84
MultiSimilarity	60.84	72.15	81.67	88.86
SoftTriple	61.28	73.11	82.58	89.37
ProxyAnchor	**62.93**	**74.00**	**83.13**	**89.62**

(b) embedding size = 128.

	R@1	R@2	R@4	R@8
Contrastive	62.64	73.66	82.55	89.03
Triplet	60.48	72.13	82.11	89.03
LiftedStructure	60.16	72.35	81.88	88,44
NPair	58.91	70.66	79.98	87.74
ProxyNCA	62.76	73.13	82.17	88.50
Margin	63.00	74.00	83.59	90.41
ArcFace	61.33	71.84	80.13	87.36
MultiSimilarity	63.96	74.85	83.63	90.31
SoftTriple	64.16	75.59	84.01	90.21
ProxyAnchor	**66.71**	**76.79**	**85.18**	**90.63**

(c) embedding size = 512.

	R@1	R@2	R@4	R@8
Contrastive	64.87	75.41	83.27	89.67
Triplet	63.52	75.62	84.38	90.50
LiftedStructure	65.92	75.81	84.50	90.41
NPair	61.36	72.81	82.08	89.01
ProxyNCA	65.22	75.55	83.76	89.60
Margin	64.99	76.15	84.60	90.46
ArcFace	64.40	74.68	83.20	89.60
MultiSimilarity	68.69	78.56	86.75	92.08
SoftTriple	67.27	77.73	86.19	92.00
ProxyAnchor	**69.48**	**79.27**	**86.95**	**92.37**

(d) embedding size = 1024.

	R@1	R@2	R@4	R@8
Contrastive	66.51	76.50	85.15	90.73
Triplet	63.55	75.35	84.03	90.36
LiftedStructure	66.34	76.67	84.47	90.36
NPair	61.83	72.60	82.07	89.01
ProxyNCA	65.12	74.78	83.56	89.60
Margin	65.48	76.54	84.53	91.15
ArcFace	65.82	76.71	84.18	89.70
MultiSimilarity	68.72	79.17	**87.15**	92.29
SoftTriple	67.42	78.16	86.02	91.64
ProxyAnchor	**69.82**	**79.86**	87.12	**92.69**

We first present the CUB200-2011 experiments made using ResNet50. These can be seen in Table 6. The order is chronological, so that it is easy to track the progress (or the lack of it). As already mentioned, these experiments are made using 4 different embedding sizes: 64, 128, 512 and 1024. From a quick look it is easy to see that the worst performance is made by NPair and Triplet, while MultiSimilarity, SoftTriple and ProxyAnchor have consistently the best performance. Contrastive seems to perform way better than it is presented in papers. LiftedStructure performs better than Triplet and NPair and sometimes it even performs better than later losses like ProxyNCA, Margin and ArcFace that seem to be ranked somewhere in the middle concerning their performance. We observe that apart from NPair and Triplet that are consistently ranked in the worst positions and MultiSimilarity, SoftTriple and ProxyAnchor that are consistently ranked in the best

positions, the other losses have significant variation. Of course we now compare their performance in relation to embedding size, but if this behaviour continues when using different Networks, this is something that we should further investigate.

At this point it is crucial to highlight some unfair comparisons that were confirmed. In [26], Margin seems to outperform LiftedStructure by 20% and NPair by more than 12%, but while Margin uses ResNet50, LiftedStructure and NPair use GoogLeNet. In Table 6b it is easy to see that when the the comparison is made under equal conditions the true differences are less than 3% and less than 4% respectively.

Fig. 27 presents the results of Table 6b in a graphical way. Recall@1 has been been highlighted, as it is considered to be the most informative metric. The order is again chronological and thus if the papers really reflected the reality, metrics of this Fig. should be constantly increasing. While Contrastive is the oldest loss function, its performance seems to be really competitive. NPair is introduced in [48] as an improved version of Triplet with multiple negatives, but this is not reflected in our results. The reason for this is probably the fact that while Triplet follows its most sophisticated implementation in our experiments and uses semi-hard mining (Eq. 2.20), NPair does not use any kind of mining, as in [48] is mentioned that when the number of output classes is not too large, hard negative mining is not necessary.

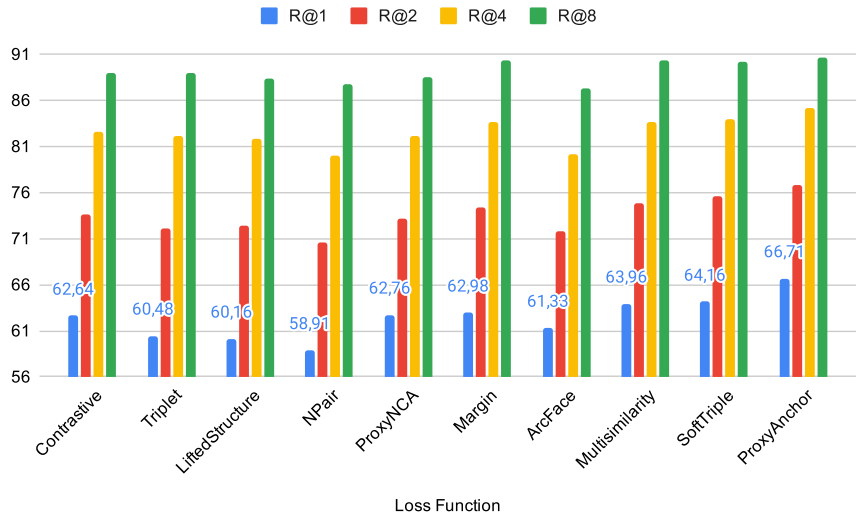


Figure 27: Comparing the retrieval quality of loss functions on CUB200-2011 using ResNet50 with an embedding size of 128.

Except for the results themselves, there are some other aspects concerning training that have to be mentioned. Classification loss functions like ProxyNCA, SoftTriple and ProxyAnchor seem to converge way faster than embedding loss functions. This is really impressive with ProxyAnchor, which most of the times achieves to reach its greatest accuracy in less than 20 epochs. Embedding loss functions seem to be trapped sometimes in local minima and while most of the times achieve to exceed them, this delays the whole training. Of course, there is a trade off here between the ability to capture data-to-data relations and the convergence speed. While classification losses converge faster, their ability to capture the inherent structure of data is limited and this is reflected in their performance. Exception to this are SoftTriple and ProxyAnchor, as the first one uses multiple proxies for each class, while the second one associates all data in a batch with each proxy.

Let us now have a closer look at the retrieval quality of the losses using different embed-

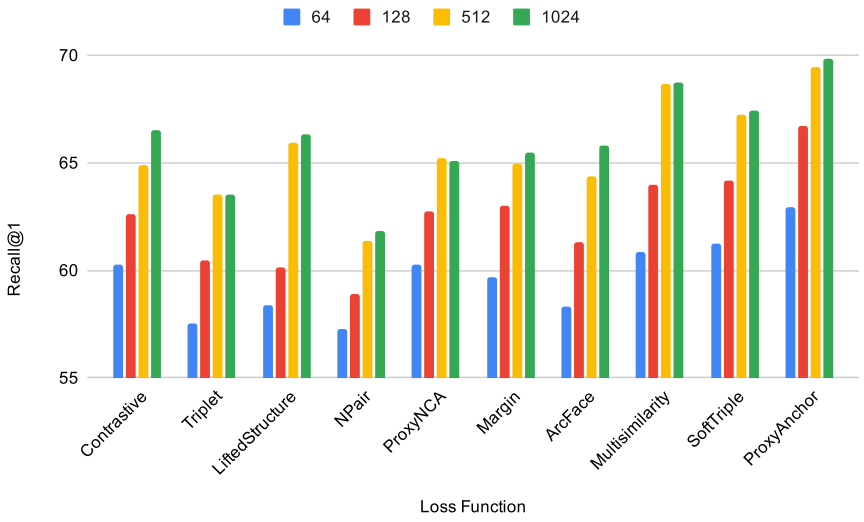


Figure 28: Comparing the retrieval quality of loss functions on CUB200-2011 using ResNet50 with different embedding sizes.

ding sizes. Figure 28 shows that in almost all the cases embeddings of size 1024 have very similar retrieval quality as embeddings of size 512. This means that the size of 512 is the optimal for ResNet50 and there is no need to use 1024, especially when this entails more computational complexity.

4.1.2 CUB200-2011 BNInception

The next experiments we are presenting are the CUB200-2011 experiments made using BNInception. These can be seen in Table 7. All the losses seem to perform a little worse when using BNInception, but this is something to be expected, as BNInception is a shallower Network with less discriminative power compared to ResNet50. BNInception is the Network that has been used in a lot of recent papers as can be seen in Table 2, so this makes the comparison between our results and the official results of papers plausible. SoftTriple reports R@1 equal to 65.40 using an embedding size of 512 in [17], while we report R@1 equal to 66.76. Moreover, SoftTriple seems to be the only loss function approaching the accuracy of ProxyAnchor, while sometimes it even exceeds it. Contrastive's performance is again better than expected and reported on papers. The performance of LiftedStructure is really impressive too. It is ranked on the third and fourth position managing to even pass MultiSimilarity and SoftTriple when using an embedding size of 64. NPair has again the worst performance.

Of course, there are again a lot of unfair comparisons that we can highlight. In [14], MultiSimilarity is compared to ProxyNCA. They are both using a 64-dimensional embedding, but as already mentioned, it is not clear whether ProxyNCA uses the BNInception architecture or a GoogLeNet architecture with batch normalization. Table 7a probably confirms this reflection, as we report a R@1 equal to 56.98% for ProxyNCA, which is more than 7% greater than the one reported in the paper. In turn, in [15], ProxyNCA is compared with Triplet, LiftedStructure and NPair that are all using GoogLeNet without batch normalization. Our results can be seen in Table 7a. When using the BNInception with an embedding of size 64, LiftedStructure performs better than ProxyNCA which in turn really performs better than Triplet and NPair.

Table 7: CUB200-2011 BNInception experiments.

(a) embedding size = 64.

	R@1	R@2	R@4	R@8
Contrastive	58.88	69.70	78.53	86.12
Triplet	55.82	67.13	77.11	83.95
LiftedStructure	58.29	68.96	79.43	87.22
NPair	54.17	65.98	76.87	83.80
ProxyNCA	56.98	67.10	77.08	85.14
Margin	56.80	68.08	78.00	85.24
ArcFace	55.77	67.92	77.92	85.50
MultiSimilarity	57.24	69.31	79.49	86.92
SoftTriple	58.07	69.42	79.42	87.39
ProxyAnchor	**61.06**	**72.67**	**82.05**	**88.67**

(b) embedding size = 128.

	R@1	R@2	R@4	R@8
Contrastive	61.24	71.79	80.40	87.62
Triplet	58.56	70.12	79.10	86.45
LiftedStructure	61.60	73.36	81.97	88.61
NPair	56.90	69.02	78.02	84.98
ProxyNCA	60.15	71.08	81.15	85.80
Margin	60.80	71.45	81.90	86.24
ArcFace	59.94	71.08	80.57	87.63
MultiSimilarity	61.92	73.28	82.99	89.21
SoftTriple	63.44	74.29	83.27	**89.96**
ProxyAnchor	**63.88**	**74.51**	**83.86**	89.92

(c) embedding size = 512.

	R@1	R@2	R@4	R@8
Contrastive	63.28	74.51	82.83	89.50
Triplet	61.98	73.59	83.80	88.87
LiftedStructure	64.28	75.47	83.91	89.89
NPair	59.90	71.98	80.47	87.25
ProxyNCA	63.84	74.02	82.98	89.54
Margin	63.48	75.86	83.90	89.78
ArcFace	62.36	73.48	81.67	88.08
MultiSimilarity	65.24	75.76	84.69	90.48
SoftTriple	66.76	77.09	85.36	**91.21**
ProxyAnchor	**68.11**	**78.63**	**85.77**	91.12

(d) embedding size = 1024.

	R@1	R@2	R@4	R@8
Contrastive	64.43	74.48	82.88	89.47
Triplet	62.45	73.80	83.10	89.20
LiftedStructure	64.86	75.68	84.00	90.01
NPair	60.76	71.89	81.67	88.40
ProxyNCA	64.10	74.40	82.80	89.14
Margin	64.08	75.40	83.01	89.90
ArcFace	63.07	73.94	83.04	88.93
MultiSimilarity	66.22	77.62	85.40	90.94
SoftTriple	67.44	78.11	**85.91**	91.28
ProxyAnchor	**68.47**	**78.41**	85.75	**91.36**

Concerning different embedding sizes, Fig. 29 shows that unlike ResNet50, BNInception seems to improve its retrieval quality by little when using an embedding size of 1024.

4.1.3 CUB200-2011 GoogLeNet

Finally, CUB200-2011 experiments using GoogLeNet are presented in Table 8. GoogLeNet is the shallowest of all the 3 Networks and this is reflected in the retrieval quality of it, as the average Recall@1 is about 3,5% lower than the corresponding one when using BN-Inception and 5% lower than the corresponding one when using ResNet50. ProxyAnchor is again the winner among all the loss functions, while SoftTriple and MultiSimilarity seem to perform worse than in previous experiments. Contrastive, LiftedStructure and ArcFace have an impressive performance and are ranked in the very first positions. In fact, ArcFace seems to exceed the performance of almost all the loss functions except of ProxyAnchor when using an embedding size of 512 and 1024. The performance of Triplet, NPair and ProxyNCA is the worst.

Concerning different embedding sizes, Fig. 30 shows that GoogLeNet seems to improve its retrieval quality by little when using an embedding size of 1024.

4.1.4 CARS196 BNInception

The next dataset on which we conduct experiments is the CARS196. In this Chapter, we present the CARS196 experiments made using BNInception. The rest of them us-

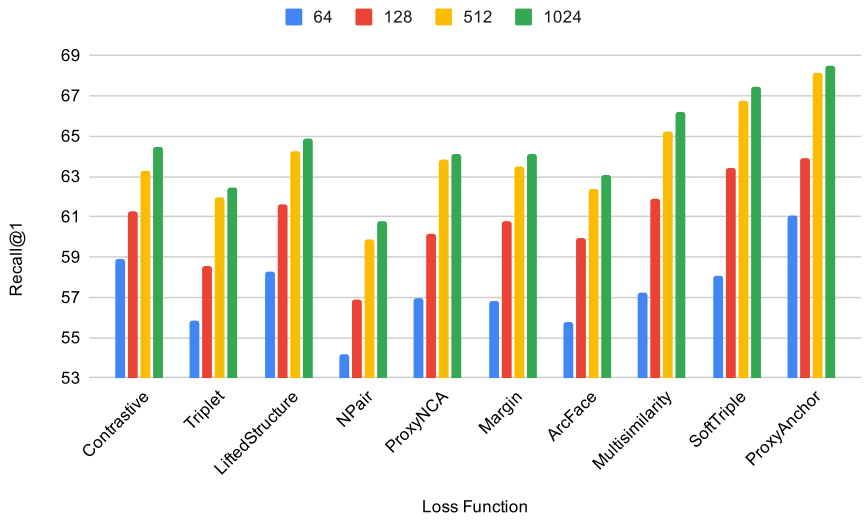


Figure 29: Comparing the retrieval quality of the loss functions on CUB200-2011 using BNInception with different embedding sizes.

Table 8: CUB200-2011 GoogLeNet experiments.

(a) embedding size = 64.

	R@1	R@2	R@4	R@8
Contrastive	56.36	67.94	78.41	86.19
Triplet	52.12	63.69	75.08	84.37
LiftedStructure	55.18	67.76	77.51	86.02
NPair	48.76	60.38	71.78	81.36
ProxyNCA	51.01	61.93	73.07	82.56
Margin	54.27	66.48	77.13	85.69
ArcFace	52.92	63.52	74.31	82.77
MultiSimilarity	53.47	65.69	76.33	85.07
SoftTriple	55.00	67.51	77.95	85.96
ProxyAnchor	**58.07**	**69.23**	**79.37**	**87.05**

(b) embedding size = 128.

	R@1	R@2	R@4	R@8
Contrastive	57.73	69.04	79.51	87.29
Triplet	55.06	67.22	77.80	85.62
LiftedStructure	58.07	69.78	79.56	87.14
NPair	50.30	61.19	72.99	81.79
ProxyNCA	55.77	66.88	76.90	85.16
Margin	57.88	69.54	79.29	86.99
ArcFace	56.94	68.01	77.97	85.67
MultiSimilarity	55.66	68.37	78.87	86.95
SoftTriple	57.00	68.91	79.61	87.61
ProxyAnchor	**60.23**	**71.89**	**82.26**	**88.86**

(c) embedding size = 512.

	R@1	R@2	R@4	R@8
Contrastive	61.01	72.79	82.07	88.20
Triplet	57.60	69.35	79.60	87.56
LiftedStructure	60.89	72.37	81.20	88.67
NPair	54.22	67.10	77.29	85.05
ProxyNCA	57.46	69.09	78.40	86.30
Margin	60.61	71.51	80.77	87.90
ArcFace	61.60	72.67	81.95	88.62
MultiSimilarity	59.57	72.42	82.42	89.76
SoftTriple	60.90	71.62	81.67	88.71
ProxyAnchor	**63.84**	**75.25**	**84.05**	**90.29**

(d) embedding size = 1024.

	R@1	R@2	R@4	R@8
Contrastive	62.05	73.14	82.14	88.78
Triplet	58.54	69.68	80.22	87.84
LiftedStructure	61.77	72.74	82.19	89.08
NPair	55.40	67.58	77.86	85.75
ProxyNCA	57.60	69.02	78.61	86.34
Margin	59.55	71.49	80.96	88.08
ArcFace	62.24	73.57	82.38	88.42
MultiSimilarity	61.16	72.92	82.51	89.18
SoftTriple	61.55	73.09	82.49	89.61
ProxyAnchor	**64.47**	**75.96**	**84.61**	**90.63**

ing ResNet50 and GoogLeNet are presented in Appendix A. We choose to present the CARS196 BNInception experiments here, as BNInception is the Network that has been used the most in recent papers.

NPair and Triplet seem to have the worst performance. ArcFace seems to improve its performance as the embedding size is increased, but still does not manage to get to the first

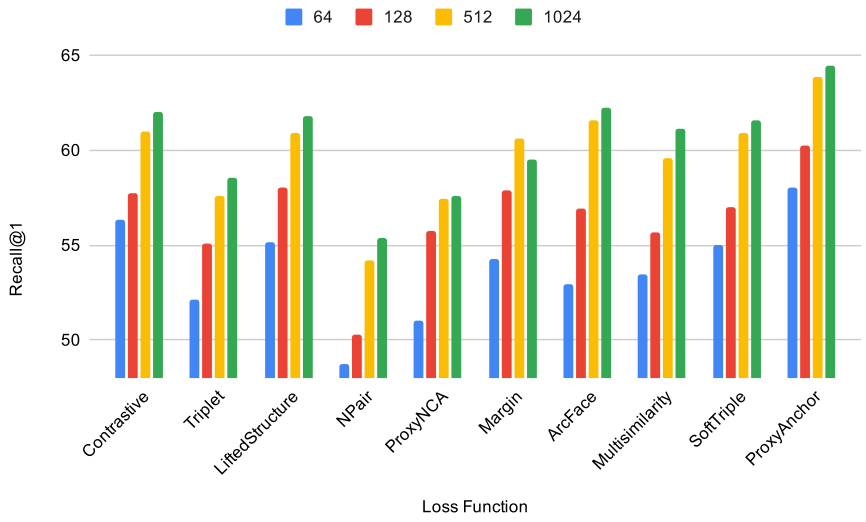


Figure 30: Comparing the retrieval quality of the loss functions on CUB200-2011 using GoogLeNet with different embedding sizes.

Table 9: CARS196 BNInception experiments.

(a) embedding size = 64.

	R@1	R@2	R@4	R@8
Contrastive	73.39	81.98	88.14	92.61
Triplet	70.02	79.12	85.98	91.01
LiftedStructure	73.53	82.51	88.40	92.81
NPair	68.54	78.21	84.90	89.87
ProxyNCA	72.52	81.20	86.05	91.20
Margin	72.94	81.48	87.09	91.68
ArcFace	69.33	78.82	85.62	90.74
MultiSimilarity	76.25	84.60	90.30	94.50
SoftTriple	77.70	86.11	91.33	95.02
ProxyAnchor	**79.79**	**87.27**	**92.44**	**95.52**

(b) embedding size = 128.

	R@1	R@2	R@4	R@8
Contrastive	75.52	84.12	89.35	93.17
Triplet	72.48	81.80	87.90	92.02
LiftedStructure	77.68	85.27	90.47	94.12
NPair	70.56	80.18	86.50	90.46
ProxyNCA	76.10	84.98	90.03	94.24
Margin	78.12	86.03	91.24	94.45
ArcFace	75.19	83.34	88.86	92.71
MultiSimilarity	80.69	87.75	92.29	95.53
SoftTriple	81.44	89.08	**93.68**	**96.35**
ProxyAnchor	**83.11**	**89.53**	93.46	95.99

(c) embedding size = 512.

	R@1	R@2	R@4	R@8
Contrastive	79.09	86.36	91.69	95.06
Triplet	77.02	84.12	89.79	93.56
LiftedStructure	79.82	86.79	91.86	94.92
NPair	73.25	81.86	86.58	90.45
ProxyNCA	81.02	86.97	92.47	95.12
Margin	81.98	87.75	91.75	94.85
ArcFace	79.42	86.77	91.71	94.70
MultiSimilarity	83.75	89.84	93.75	96.53
SoftTriple	85.29	91.10	**94.78**	**97.10**
ProxyAnchor	**86.21**	**91.71**	94.70	96.95

(d) embedding size = 1024.

	R@1	R@2	R@4	R@8
Contrastive	78.86	86.37	91.72	94.93
Triplet	77.40	84.23	89.98	93.47
LiftedStructure	79.46	86.70	91.39	95.00
NPair	74.28	81.98	86.79	90.63
ProxyNCA	81.90	87.70	91.66	94.45
Margin	81.78	87.60	91.78	94.90
ArcFace	79.74	86.57	91.24	94.50
MultiSimilarity	84.38	90.64	94.34	96.64
SoftTriple	86.20	**91.88**	**95.41**	**97.40**
ProxyAnchor	**86.41**	91.70	94.90	97.12

positions. Contrastive performs better than expected, but not as good as in the CUB200-2011 experiments. LiftedStructure, ProxyNCA and Margin are ranked somewhere in the middle. Finally, MultiSimilarity, SoftTriple and ProxyAnchor achieve the best performance once again.

As shown in Table 9, SoftTriple manages to get better R@2, R@4 and R@8 scores than

ProxyAnchor in some experiments. Actually, SoftTriple manages to get better scores than the ones officially reported in the corresponding paper [17], as authors report a R@1 equal to 84.5% using the BNInception with 512-dimensional embedding, while we report 85.29% (7c. Moreover, another unfair comparison is confirmed in this paper, as authors compare SoftTriple with Margin, but Margin uses ResNet50 with a 128-dimensional embedding and thus achieves a R@1 equal to 79.6%. In Table 9c we can see that when the comparison is made fairly with Margin using BNInception with 512-dimensional embedding, its performance is increased by more than 2% reaching a R@1 equal to 81.98%.

In the same paper, SoftTriple using BNInception with an embedding size of 64 is compared with other loss functions like Triplet, LiftedStructure and NPair that use GoogLeNet. In Table 9a we can see that when the comparison is made fairly with Triplet and LiftedStructure using BNInception, their performance is increased by more than 20% and while they again fail to surpass SoftTriple, they achieve to get really closer than presented in the paper. As long as NPair is concerned, it is worth mentioning that this seems to perform worse in our experiments than what it is reported in papers. This is something that will be discussed in the next section.

4.1.5 SOP BNInception

Finally, we present the SOP experiments made using BNInception. The rest of them using ResNet50 and GoogLeNet are presented in Appendix A. We choose to present the SOP BNInception experiments here, as BNInception is the Network that has been used the most in recent papers.

Table 10: SOP BNInception experiments.

(a) embedding size = 64.

	R@1	R@10	R@100	R@1000
Contrastive	73.90	86.97	93.02	96.40
Triplet	70.00	83.74	88.17	91.20
LiftedStructure	74.88	88.22	95.16	98.57
NPair	68.11	81.10	87.13	90.85
ProxyNCA	73.66	86.11	92.44	95.81
Margin	73.80	86.65	92.57	95.63
ArcFace	71.85	83.68	89.10	92.54
MultiSimilarity	74.76	88.27	94.89	98.42
SoftTriple	**76.82**	89.04	95.01	**98.21**
ProxyAnchor	76.56	**89.05**	**95.12**	98.12

(b) embedding size = 128.

	R@1	R@10	R@100	R@1000
Contrastive	76.00	89.01	95.41	97.78
Triplet	72.27	84.79	90.11	93.14
LiftedStructure	76.10	88.85	95.36	98.67
NPair	70.23	82.47	87.98	90.71
ProxyNCA	75.40	87.89	92.78	95.45
Margin	75.74	87.92	93.24	95.28
ArcFace	73.14	64.56	75.04	84.51
MultiSimilarity	76.43	89.04	95.32	98.56
SoftTriple	**78.43**	90.11	95.61	98.38
ProxyAnchor	78.42	**90.22**	**95.64**	**98.42**

(c) embedding size = 512.

	R@1	R@10	R@100	R@1000
Contrastive	77.10	89.01	95.23	97.85
Triplet	73.85	84.93	90.11	92.45
LiftedStructure	77.14	89.62	95.73	98.75
NPair	71.45	82.88	87.99	90.58
ProxyNCA	76.15	88.02	93.14	95.47
Margin	76.54	87.98	92.51	94.89
ArcFace	74.91	85.29	90.81	93.39
MultiSimilarity	77.73	89.88	95.77	98.69
SoftTriple	79.29	**90.70**	95.85	98.53
ProxyAnchor	**79.42**	90.66	**96.05**	**98.62**

(d) embedding size = 1024.

	R@1	R@10	R@100	R@1000
Contrastive	77.98	88.90	95.11	97.90
Triplet	73.47	84.78	90.12	92.81
LiftedStructure	76.94	89.37	95.71	98.70
NPair	70.98	81.57	87.59	90.11
ProxyNCA	76.85	86.78	92.49	94.68
Margin	76.15	87.11	92.89	94.97
ArcFace	75.48	85.48	91.34	34.27
MultiSimilarity	78.20	89.82	95.75	98.69
SoftTriple	**79.51**	90.64	95.96	98.57
ProxyAnchor	79.35	**90.89**	**96.10**	**98.63**

4.2 Discussion

Both while conducting the experiments, as also after they were over, we observed some aspects that should be discussed.

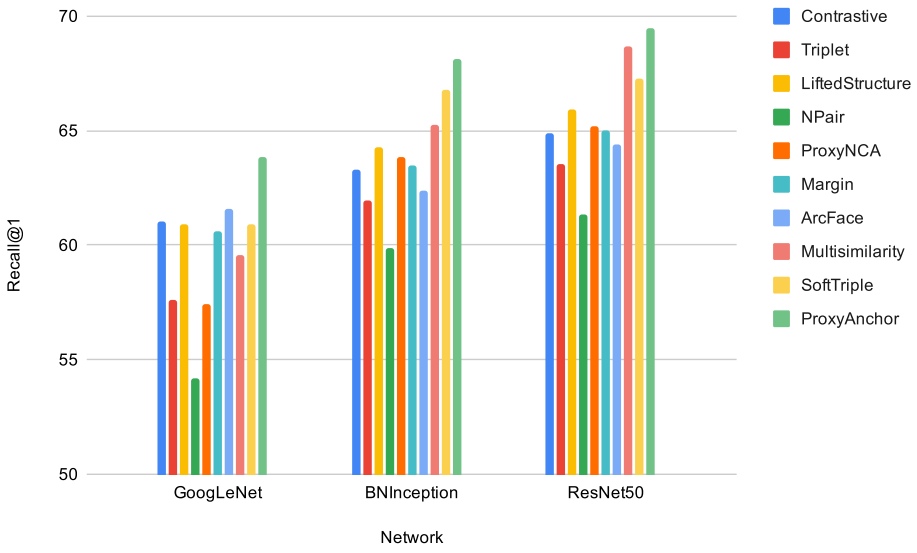


Figure 31: Comparing the discriminative power of Networks. All loss functions use an embedding size of 512.

4.2.1 About Networks

First of all, let us have a comparison between the different Networks that were used in our experiments. Fig. 31 shows the Recall@1 scores of each loss function when trained using each of the 3 Networks on CUB200-2011. It is obvious that ResNet50 is the undisputed winner followed by BNInception and finally GoogLeNet. ResNet50's representations are surely more powerful than those of the other 2 Networks. This means that a loss function that uses ResNet50 cannot be compared with a loss function using another Network, as this comparison is not made under equal terms. When having an equally sized embedding, it is sure that the loss function using ResNet50 will have a better performance. This performance is not due to the superiority of the loss function, but rather due to the superiority of the Network.

4.2.2 About Embeddings

Let us now have a discussion about the retrieval quality of loss functions using different embedding sizes. Figures 28, 29, 30 show that while the increase in retrieval quality is great when going from 64 to 128 and from 128 to 512-dimensional embeddings, this does not happen when going from 512 to 1024. Of course, there are some exceptions to this like the more than 1% increase of MultiSimilarity when using GoogLeNet with 512 and 1024 embeddings on CUB200-2011, as can be seen in Tables 21c and 21d, but again this increase can not be considered so significant. Moreover, taking into consideration the computational cost of a 1024-dimensional embedding, the conclusion to be drawn is that given these Networks, the 512-dimensional embedding is the optimal solution.

4.2.3 About Datasets

CUB200-2011 and CARS196 have almost the same number of classes, but while CUB200-2011 has 11788 images, CARS196 has 16815, which means that while CUB200-2011 has an average of almost 59 images per class, CARS196 has an average of more than 82. Images of CUB200-2011 seem to have more intraclass variance, as birds can be seen in very different angles and poses, but also sometimes there are images of chicks that differ a lot from adult birds of the same class. CUB200-2011 dataset is considered to be the most difficult of the 3 and this is reflected in the retrieval scores of loss functions. It is even more difficult than the SOP, which is a huge dataset of 22634 classes with 120053 images. Let us highlight the fact that SOP has an average of only 5.3 images per class. Concerning complexity it is obvious that SOP is way more expensive and thus makes finetuning a lot more difficult.

4.2.4 About Loss Functions

4.2.4.1 Embedding vs. Classification Loss Functions

Embedding loss functions like Contrastive, Triplet, Margin, etc. are able to capture data-to-data relations, but because of this property they are also sensitive to noisy labels and outliers. Their constraints can often be easily fulfilled and thus mining informative tuples is necessary. The gradients of tuples that fulfil their constraints are really small and thus they do not contribute significantly to training, thus making the convergence slower or even impossible. Mining tuples that violate their constraints is crucial and this is reflected in trivial mining methods like the semi-hard mining of Triplet and the distance weighted mining of Margin. Margin is a very simple loss function that is mostly based on its mining method. In contrast, MultiSimilarity is a powerful loss function that improves even more its performance using hard mining.

Classification loss functions are based on the idea of adopting some kind of proxies, which enable faster and more reliable convergence. A proxy is a learnable representative of a subset of data. Classification loss functions demand less hyperparameter finetuning and are more robust again outliers. SoftTriple adopts the idea of multiple proxies per class in order to capture better the intraclass variance. ProxyAnchor is using the Log-Sum-Exp of MultiSimilarity in order to treat positives and negatives different, but this is done under a classification scheme of having proxies as anchors.

4.2.4.2 Tournament of Loss Functions

Our experiments show that some loss functions perform consistently better than others. This can be easily observed qualitatively, but we need something more than this, we need a quantitatively process that will help us confirm this observation and draw important conclusions. Under this context, we introduce a Tournament of Loss Functions on CUB200-2011. We choose to run this tournament on CUB200-2011, as it is the dataset that we have experimented with the most and we have presented its results in more detail.

We collect the ranking of each loss function in each experiment, where the total number of experiments is 12, as there are 4 different embedding sizes for each of the 3 different Networks. For example, given the experiment of Table 8a, ProxyAnchor is assigned with a ranking equal to 1, while NPair with a ranking equal to 10. We sum these rankings for

Table 11: The Tournament of Loss Functions.

Loss Function	Total Rankings	Average Ranking	Standard Deviation
ProxyAnchor	12	1	0
SoftTriple	38	3.16	1.19
MultiSimilarity	51	4.25	2.05
Contrastive	52	4.33	1.72
LiftedStructure	54	4.5	1.89
Margin	70	5.83	1.27
ArcFace	79	6.58	2.31
ProxyNCA	81	6.75	1.66
Triplet	103	8.85	0.51
NPair	120	10	0

each loss function and this results in total rankings. The smaller this number, the better the performance of the corresponding loss function in our experiments. Then, we divide this number by 12 in order to get the average ranking. We calculate the standard deviation of each loss function taking into account the average ranking and the ranking in each corresponding experiment. A loss function with standard deviation equal to 0 is a loss function that was consistently ranked in the same position across all the CUB200-2011 experiments, while a loss function with great standard deviation is a loss function whose position varied the most.

Gold Medal Table 11 shows that ProxyAnchor is consistently the winner across all the CUB200-2011 experiments. ProxyAnchor is a loss function combining interesting ideas that seem to also work on their own. This combination probably results in its superior performance. Ideas like:

- the Log-Sum-Exp of Eq. 2.31 that pulls and pushes embedding vectors with different degrees of strength that are determined by their relative hardness
- the utilization of proxies that will enable a faster, reliable and stable convergence
- the utilization of proxies as anchors and the association of them with each sample in a batch, so that data-to-data relations can be indirectly taken into account

SoftTriple is ranked in the second position and has a standard deviation of 1.19, which means that in the worst case scenario is ranked a little bit more than one place lower than its average ranking. SoftTriple's performance is sometimes surpassing this of ProxyAnchor and is even better than presented in the official paper.

Silver Medal SoftTriple is also based in proxies, but its main idea is that of using multiple proxies that can capture the inherent structure of data. Each anchor is associated with proxies of the same class and proxies of different classes. Finding the appropriate number of proxies per class is challenging and trades between efficiency and effectiveness. In the extreme case where this number is equal to the number of original examples, it results in the common Triplet formulation of cubic complexity, while in the naive case where this number is equal to one, it results in the common SoftMax formulation of linear complexity. SoftTriple's solution is using the $L_{2,1}$ norm in order to find an optimal and adaptive number of proxies per class.

Bronze Medal MultiSimilarity is ranked third and is really close to the unexpectedly good Contrastive. However, MultiSimilarity ranks lower than expected in GoogLeNet experiments and this is the reason why its average ranking is 4.25. As long as there is no other obvious reason for its bad performance using GoogLeNet, this raises some questions about the hyperparameter finetuning. We remind that in our experiments we only finetune the learning rate and the scheduling; the hyperparameters of loss functions are generally not finetuned and thus there is a chance that MultiSimilarity's hyperparameters were not the optimal for GoogLeNet.

MultiSimilarity is a pure embedding loss function that fully exploits data-to-data relations and similarities. It takes advantage of every sample in the batch. Its formulation has clearly affected ProxyAnchor. Moreover, if we take a look at Eq. 2.24, 2.30, 2.31, we will see that MultiSimilarity, SoftTriple and ProxyAnchor have some aspects in common and once these loss functions perform the best, one should take into account these aspects when formulating a new loss function.

Honors Contrastive's performance is impressive and especially if one considers its formulation simplicity and the fact that is utilizes no mining. LiftedStructure's performance is better than expected and seems to be relatively consistent. Margin is performing as expected. ArcFace is the only loss function that does not come from Deep Metric Learning, but from the face verification task. Its standard deviation is the greatest among all the loss functions and this raises again the same questions about hyperparameter finetuning as MultiSimilarity. ProxyNCA has one of the worst performances among all the loss functions, while NPair has always the worst performance.

This ranking of NPair is something that worries us, as its performance is sometimes even worse than presented in papers. Let us recall the fact that NPair is a Triplet with more than one negatives and thus it is expected to have better performance than Triplet. However, while Triplet uses semi-hard negative mining in our experiments, NPair does not use any kind of mining. Apart from that, authors of NPair apply multiscale transformations in images, while we just apply the singe crop transformation of the official setup.

4.2.5 About Setup

Let us now have a discussion about the setup. As already mentioned, in general, we only finetune the learning rate and the scheduling, not the hyperparameters of loss functions. Exception to this is ArcFace, which is a loss function designed for face verification and thus we finetune its parameters for the Deep Metric Learning task. We notice that minor changes in hyperparameters are able to affect dramatically the performance of methods. This is making the whole finetuning a lot more difficult and somehow uncertain. There are cases in which we are not 100% sure if the hyperparameters found are the optimal, both the ones found by authors and the ones found by us. Concerning convergence, most of the times loss functions achieve to reach a minimum, which is not sure, though, if it is a local or a global.

Concerning the lack of validation set, one cannot really claim that methods might overfit the test set because of this, as the classes of the test set are different from the ones of training set. One can claim, though, that finetuning using the test set is not a good tactic and can also question the generalization ability of this setup to unseen data. All these motivate us to experiment with and design a new setup.

5. OUR SETUP

5.1 Cross Validation

The first idea we experiment with is the utilization of a 10-fold cross validation. We want to keep the classes of the test set the same, so that our results are comparable with bibliography. We use the classes of training set of default setup for cross validation: for the first fold we randomly select the 9/10 of the training classes for training and the rest 1/10 for validation. As default setup is meant the one introduced by [1] and described in the first Section of this Chapter. We choose to make a random selection rather than a deterministic one, as we notice that consecutive classes might be semantically similar, e.g. the first 3 classes of CUB200-2011 are species of the Albatross family. For each next fold we repeat the same process, but each time the random selection is more limited, as the same classes cannot be used twice for validation. This means that by the end of the cross validation all the classes will have been included exactly once in validation set.

The metric we choose to report at each epoch is the Recall@1 on validation set. Once the training of 1 fold is over, we save and load the model that scored the best Recall@1 on validation set for testing. We compute the Recall@1 on test set of this model. By the end of cross validation, we have 10 different models. We compute and report the average and the standard deviation of their respective Recall@1 on test set.

We conduct experiments using the BNInception with a 512-dimensional embedding on CUB200-2011. The choice of Network and embedding size is in line with the respective choice of the state-of-the-art papers. The choice of dataset is purely determined by size. We choose to make experiments using the top-3 loss functions of our Tournament of Loss Functions, i.e. ProxyAnchor, SoftTriple and MultiSimilarity. Table 12 shows the results. We also report the best Recall@1 on validation set of each fold for MultiSimilarity loss, as well as the Recall@1 on test set of each respective model. These results can be seen in Table 13.

These experiments are using the same hyperparameters as the ones conducted with the default setup. We remind that these parameters can be seen in Table 4. We do not conduct any hyperparameter search using the cross validation setup, as this proved to be too expensive computationally. Let us consider the fact that while in default setup we only train 1 model per experiment, in the 10-fold cross validation setup we train 10 models. This led us to design a different setup that is balancing between the computational complexity and the need of a validation set.

Table 12: The cross validation scores of MultiSimilarity, SoftTriple and ProxyAnchor using the BNInception with an embedding size of 512 on CUB200-2011.

Loss Function	R@1
MultiSimilarity	63.61 $\pm$0.59
SoftTriple	64.09 $\pm$0.48
ProxyAnchor	66.32 $\pm$0.44

Table 13: MultiSimilarity R@1 scores on each fold of the 10-fold cross validation. We choose the model scored the best R@1 on each validation set and report its R@1 on test set.

Fold	Best R@1 on Validation Set	R@1 on Test Set
1	94.07	64.25
2	94.39	63.67
3	99.15	63.60
4	98.21	63.76
5	97.63	64.28
6	98.49	64.24
7	99.00	63.35
8	99.50	63.33
9	99.50	63.23
10	97.92	62.34

Table 14: Comparing the different fixed validation splits in order to find the one with the best Recall@1 on test set. The values of R@1 on validation set are greater than the respective ones on test set, as the validation classes are only 10, while the test classes are 100.

Split Ratio (Training Classes/ Validation Classes)	Best R@1 on Validation Set	R@1 on Test Set
70/30	86.13	61.28
80/20	92.53	62.74
90/10	91.49	**64.38**
95/5	93.31	62.92

5.2 Fixed Validation Set

The idea behind the fixed validation set is that we want to train just 1 model, exactly as in the default setup, but we want to split the classes of training set in training and validation classes. We choose to randomly select qualitatively which classes will be included in training set and which in validation set for the same reason as before. Quantitatively, we want to find the split ratio that gives the best R@1 on test set. We conduct experiments using the MultiSimilarity loss in different split schemes on CUB200-2011. For all the fixed validation set experiments we choose to use again the BNInception with a 512-dimensional embedding. Table 14 shows that the optimal split ratio is the 90/10.

We use the 90 classes of the default training set for training and the rest 10 for validation. We conduct extensive hyperparameter search on validation set using MultiSimilarity, SoftTriple and ProxyAnchor. The search is done as follows: we define a range within we search using a specific step for the optimal value of each hyperparameter. For example, we define the range [0,1] with a search step of 0.1 for MultiSimilarity's margin λ. We conduct consecutive experiments starting from $\lambda = 0$ and ending with $\lambda = 1$. We keep the value of λ that gives the best R@1 on validation set. For the sake of time, we do not train each model till full convergence, as the impact of the value can be seen early in the training. Table 15 shows the hyperparameters we finetune, the range within we search, the search step for each hyperparameter, as also the optimal value we find.

Let us highlight the fact that while the optimal values of MultiSimilarity's and SoftTriple's hyperparameters we find differ a lot from the ones authors report in papers, the optimal values of ProxyAnchor's hyperparameters are exactly the same.

Table 15: Hyperparameter search settings on fixed validation set.

Loss Function	Hyperparameter	Range of Search	Search Step	Optimal Value
MultiSimilarity	margin λ	[0,1]	0.1	0.8
	scale α	(0,100]	2	18
	scale β	(0,100]	2	76
	epsilon	[0,1]	0.1	0.4
SoftTriple	margin λ	[0,1]	0.1	0.4
	scale α	(0,100]	2	78
	weights lr	[0.00001, 0.0001]	0.00001	0.00005
	gamma	(0,100]	10	58
	tau	[0,1]	0.1	0.4
ProxyAnchor	margin λ	[0,1]	0.1	0.1
	scale α	(0,100]	2	32

Table 16: The R@1 scores of loss functions using the optimal values found using our fixed validation setup.

Loss Function	R@1
MultiSimilarity	65.61
SoftTriple	66.12
ProxyAnchor	66.56

We report the R@1 on test set that these 3 loss functions score using optimal hyperparameters on Table 16. Concerning MultiSimilarity and SoftTriple, these scores surpass the respective ones using the default setup. Considering the fact that we train using only the 90 out of the 100 classes of the default setup, this is not what we expected to get. Our speculation is that authors avoid to conduct extensive finetuning using the default setup, as they know that finetuning on test set is not a good practice. This speculation becomes even more powerful if one takes a look at the values of hyperparameters proposed by authors. These values can be seen in Table 4. Most of the loss functions use the same ”standard” values, e.g. 0.5 or 0.1 for margin, etc. that can be more easily justified. Concerning ProxyAnchor, the R@1 on test set is about 1,5% lower, but this is something expected, as ProxyAnchor’s official hyperparameters were already the optimal, so it is not benefited from our searching.

The fixed validation set seems to balance perfectly between the computational complexity and retrieval quality trade-off. In contrast with k-fold cross validation, it demands no extra computations, while it also gives the opportunity of an exhaustive hyperparameter search that results in optimal parameters. We thus propose not only as an alternative to default setup, but moreover as the new default setup of Deep Metric Learning.

6. OUR METHOD

Having studied thoroughly the most important loss functions of Deep Metric Learning, we design, implement and experiment with a new loss function that is proxy-based, but is in between the rationale of classification and embedding losses. It is crucial to highlight the fact that this loss function was designed before the ProxyAnchor paper publication. The fact that these two loss functions share a lot of common aspects show that we move in the right direction.

How exactly is this function working, though? There are two different variations of it. In the naive one, we assign one proxy to each class. We take advantage of the full batch during training and associate each sample with its corresponding positive proxy (proxy of the same class). We make use of the Log-Sum-Exp of MultiSimilarity in order to ensure that the relative hardness between each proxy and sample is taken into consideration. The samples of the batch are not associated with negative proxies. However, the proxies themselves are treated as negatives that should be pushed away. Iteratively, our loss function aims to pull samples of the same class close to their corresponding proxy, while also pushes proxies away. This can be seen in Fig. 32

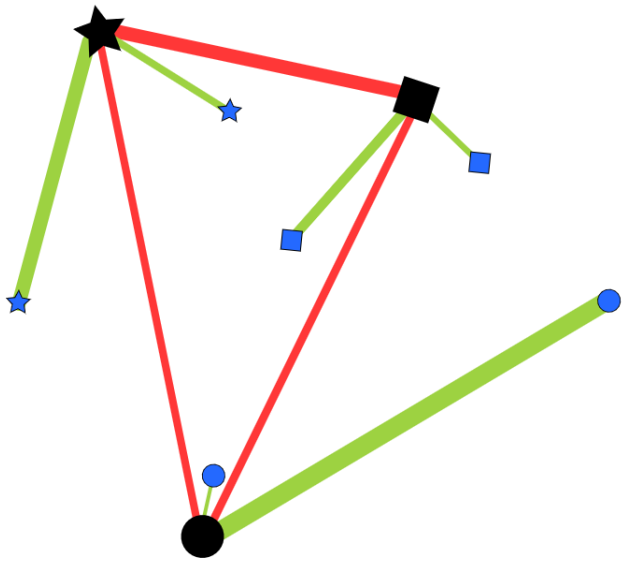


Figure 32: Visualization of the way our loss function works. Different shapes correspond to different classes. Black nodes represent proxies, while blue nodes represent samples. Green edges represent positive associations, while red nodes represent negative associations. Thickness is analogous to gradients that samples or proxies get. For example, let us examine the case of the star class that has two positive samples and two negative proxies. Concerning positives, star class is pulling closer both samples, but with different degrees of strength determined by their relative hardness. The star sample that is further away gets larger gradient than the other one. Exactly the opposite is happening concerning negatives, as the square proxy that is closer gets larger gradient than the circle proxy.

Let us formulate our loss function as:

$$\mathcal{L}_{\text{OurLoss}_1} = \frac{1}{|W^+|} \sum_{w \in W^+} \log \left(1 + \sum_{x \in X_w^+} e^{-\alpha(w^T x - \lambda)} \right) + \frac{1}{|W|} \sum_{w \in W} \log \left(1 + \sum_{w^- \in W^-} e^{\alpha(w^T w^- + \lambda)} \right), \tag{6.1}$$

Table 17: Comparing the retrieval quality of loss functions on CUB200-2011.

	R@1	R@2	R@4	R@8
Contrastive	63.28	74.51	82.83	89.50
Triplet	61.98	73.59	83.80	88.87
LiftedStructure	64.28	75.47	83.91	89.89
NPair	59.90	71.98	80.47	87.25
ProxyNCA	63.84	74.02	82.98	89.54
Margin	63.48	75.86	83.90	89.78
ArcFace	62.36	73.48	81.67	88.08
MultiSimilarity	65.24	75.76	84.69	90.48
SoftTriple	66.76	77.09	85.36	**91.21**
OurLoss	65.42	75.89	84.99	90.52
ProxyAnchor	**68.11**	**78.63**	**85.77**	91.12

Table 18: Comparing the retrieval quality of loss functions on CARS196.

	R@1	R@2	R@4	R@8
Contrastive	79.09	86.36	91.69	95.06
Triplet	77.02	84.12	89.79	93.56
LiftedStructure	79.82	86.79	91.86	94.92
NPair	73.25	81.86	86.58	90.45
ProxyNCA	81.02	86.97	92.47	95.12
Margin	81.98	87.75	91.75	94.85
ArcFace	79.42	86.77	91.71	94.70
MultiSimilarity	83.75	89.84	93.75	96.53
SoftTriple	85.29	91.10	**94.78**	**97.10**
OurLoss	84.12	90.12	94.00	96.97
ProxyAnchor	**86.21**	**91.71**	94.70	96.95

where $\lambda > 0$ is a margin, $\alpha > 0$ is a scaling factor, $W = W^+ + W^-$ indicates the set of all proxies, $X = X_w^+ + X_w^-$ indicates the batch of embedding vectors and w^- a negative proxy to w.

In its second variation, our loss function utilizes a small trick in order to exploit more data-to-data relations. In its first variation, our loss is able to capture indirectly data-to-data relations only concerning positives. This happens due to the nature of the Log-Sum-Exp that takes into consideration the relative hardness of positive samples when associating them with their corresponding proxy. However, proxies themselves are treated as negatives and are not associated anyhow with other samples. Our idea is to replace $w^T w^-$ with $\sum_{x \in X}(w^T x)(x^T w^-)$, so that the similarity between proxies is computed by taking into consideration the samples of the batch too. The second variation of our loss function can be formulated as:

$$\mathcal{L}_{\text{OurLoss}_2} = \frac{1}{|W^+|} \sum_{w \in W^+} \log\left(1 + \sum_{x \in X_w^+} e^{-\alpha(w^T x - \lambda)}\right) + \frac{1}{|W|} \sum_{w \in W} \log\left(1 + \sum_{w^- \in W^-} e^{\alpha\left(\sum_{x \in X}(w^T x)(x^T w^-) + \lambda\right)}\right), \tag{6.2}$$

We conduct experiments with the second variation of our loss function using the BNInception with a 512-dimensional embedding on CUB200-2011, CARS196 and SOP datasets. The results are shown in Tables 17, 18, 19 respectively.

Table 19: Comparing the retrieval quality of loss functions on SOP.

	R@1	R@10	R@100	R@1000
Contrastive	77.10	89.01	95.23	97.85
Triplet	73.85	84.93	90.11	92.45
LiftedStructure	77.14	89.62	95.73	98.75
NPair	71.45	82.88	87.99	90.58
ProxyNCA	76.15	88.02	93.14	95.47
Margin	76.54	87.98	92.51	94.89
ArcFace	74.91	85.29	90.81	93.39
MultiSimilarity	77.73	89.88	95.77	98.69
SoftTriple	79.29	**90.70**	95.85	98.53
OurLoss	77.92	90.01	95.89	98.99
ProxyAnchor	**79.42**	90.66	**96.05**	**98.62**

7. CONCLUSIONS AND FUTURE WORK

The remarkable success of Convolutional Neural Networks that brought changes to almost all the domains of Machine Learning and Computer Vision could not but have affected Metric Learning too. Deep Metric Learning, which is introduced by the authors of [1], succeeds Linear and Nonlinear Metric Learning.

While delving into bibliography, we realize that there are a lot of issues related to the Deep Metric Learning setup: unfair comparisons, lack of validation set, etc. This motivates us to conduct extensive experiments using the most common CNN architectures (GoogLeNet, BNInception, ResNet50) on the most common used datasets (CUB200-2011, CARS196, SOP) using 10 different loss functions (Contrastive, Triplet, LiftedStructure, NPair, ProxyNCA, ArcFace, Margin, MultiSimilarity, SoftTriple, ProxyAnchor) and 4 different embedding sizes (64, 128, 512, 1024). This work can be considered as a benchmark for fair comparisons and ablation study.

We present an extensive discussion about our results: Networks, embeddings, datasets, loss functions and finally the setup itself. The observed drawbacks lead us to design and propose:

- A new setup using a fixed validation set that seems to balance perfectly between the computational complexity and retrieval quality trade-off. Moreover, it is indicated for exhaustive hyperparameter search.
- A new loss function that is proxy-based, but is in between the rationale of classification and embedding losses. It treats proxies of different class as negatives that should be pushed away and samples of the same class as positives that should be pulled by the corresponding proxy.

Concerning future work and future directions, we propose:

- The conduction of extensive experiments using our fixed validation setup for all the losses. The optimal hyperparameters found by this setup may significantly improve their retrieval quality.
- Redesigning our loss function to capture more data-to-data relations.
- The utilization of some kind of memory and/or offline mining for our loss function.
- Experimenting with a semi-supervised or even unsupervised setup for Deep Metric Learning. This is something that has not been studied and would be interesting.

Code of this work is available in GitHub repository: `https://github.com/billpsomas/metric-learning`

A. APPENDIX

A.1 Remaining experiments

Let us present the results of experiments that were not included in Chapter 4.

A.1.1 CARS196 ResNet50

Table 20: CARS196 ResNet50 experiments.

(a) embedding size = 64.

	R@1	R@2	R@4	R@8
Contrastive	73.83	82.73	88.83	92.90
Triplet	71.11	80.76	87.57	92.40
LiftedStructure	75.48	83.89	89.68	93.57
NPair	69.10	79.18	86.53	91.80
ProxyNCA	73.20	81.70	88.03	92.33
Margin	75.88	84.44	90.30	94.39
ArcFace	73.55	81.84	87.30	91.91
MultiSimilarity	79.26	87.11	92.08	95.73
SoftTriple	79.81	87.76	92.94	96.05
ProxyAnchor	**81.63**	**88.54**	**93.14**	**95.88**

(b) embedding size = 128.

	R@1	R@2	R@4	R@8
Contrastive	77.11	85.54	90.95	94.51
Triplet	74.84	83.82	89.98	94.05
LiftedStructure	78.15	86.03	90.78	94.45
NPair	72.49	82.23	88.51	92.60
ProxyNCA	78.42	85.59	90.76	94.55
Margin	79.60	86.75	91.49	94.81
ArcFace	77.78	85.14	89.76	93.42
MultiSimilarity	84.09	90.16	94.37	96.99
SoftTriple	83.09	89.64	93.47	**96.46**
ProxyAnchor	**84.77**	**90.53**	**94.01**	96.42

(c) embedding size = 512.

	R@1	R@2	R@4	R@8
Contrastive	82.17	88.63	93.17	95.98
Triplet	77.27	85.38	90.78	94.33
LiftedStructure	81.40	88.28	92.37	95.03
NPair	75.23	83.94	88.57	92.34
ProxyNCA	81.18	87.92	92.58	95.43
Margin	82.09	88.67	92.71	95.68
ArcFace	79.18	86.45	91.12	94.53
MultiSimilarity	**89.21**	**93.56**	**96.40**	**97.87**
SoftTriple	86.50	91.93	95.29	97.29
ProxyAnchor	87.59	92.26	95.53	97.37

(d) embedding size = 1024.

	R@1	R@2	R@4	R@8
Contrastive	83.79	90.00	94.00	96.66
Triplet	77.64	85.62	90.83	94.33
LiftedStructure	82.52	88.75	92.89	95.39
NPair	75.98	83.62	88.37	92.14
ProxyNCA	81.02	87.87	92.59	95.77
Margin	80.63	87.58	92.39	95.30
ArcFace	80.80	87.39	92.11	95.18
MultiSimilarity	**88.80**	**93.32**	**96.11**	**97.82**
SoftTriple	87.28	92.34	95.73	97.74
ProxyAnchor	87.86	92.57	95.47	97.36

We present the CARS196 experiments made using ResNet50. NPair and Triplet have the worst performance. ArcFace, Margin, ProxyNCA, LiftedStructure and Contrastive are in general ranked in the middle. SoftTriple, MultiSimilarity and ProxyAnchor have the best performance. It is worth mentioning that MultiSimilarity exceeds the performance of ProxyAnchor by more than 1% when using an embedding size of 512 and 1024 as can be seen in 20c and 20d.

A.1.2 CARS196 GoogLeNet

We also present the CARS196 experiments made using GoogLeNet. NPair and Triplet perform the worst for one more time. ProxyNCA and Margin are ranked close to them in most of the experiments. ArcFace's ranking seems to vary a lot, as it has the third worst performance when using an embedding size of 64 (Table 21a), but the third best

performance when using an embedding size of 1024 (Table 21d). Contrastive and LiftedStructure are performing way better than expected and what is presented in papers and they even surpass MultiSimilarity in some experiments. Finally, SoftTriple and ProxyAnchor have the best performance among all the loss functions.

Table 21: CARS196 GoogLeNet experiments.

(a) embedding size = 64.

	R@1	R@2	R@4	R@8
Contrastive	71.71	81.14	87.85	92.40
Triplet	67.89	76.87	83.68	88.74
LiftedStructure	72.67	81.33	87.98	92.78
NPair	66.99	76.01	82.89	87.77
ProxyNCA	69.09	77.68	84.02	90.37
Margin	71.82	80.88	87.15	91.87
ArcFace	68.60	77.80	84.93	90.49
MultiSimilarity	72.18	81.74	88.69	93.35
SoftTriple	74.87	83.85	90.23	94.16
ProxyAnchor	**77.54**	**85.68**	**91.23**	**94.75**

(b) embedding size = 128.

	R@1	R@2	R@4	R@8
Contrastive	76.69	85.17	90.46	94.06
Triplet	71.13	80.78	86.04	91.82
LiftedStructure	77.36	84.95	90.67	94.22
NPair	69.26	78.13	84.38	90.03
ProxyNCA	74.12	82.78	88.16	92.87
Margin	74.76	83.10	89.19	93.67
ArcFace	76.03	83.83	89.25	93.29
MultiSimilarity	76.42	84.66	90.37	94.29
SoftTriple	78.16	86.52	92.07	95.29
ProxyAnchor	**81.61**	**88.06**	**92.68**	**95.62**

(c) embedding size = 512.

	R@1	R@2	R@4	R@8
Contrastive	80.61	87.54	92.15	95.22
Triplet	73.46	80.59	84.50	91.63
LiftedStructure	80.21	87.11	91.87	94.85
NPair	71.93	78.11	82.27	89.30
ProxyNCA	76.13	82.31	87.81	92.59
Margin	78.46	85.68	90.90	94.11
ArcFace	81.81	88.14	92.40	95.63
MultiSimilarity	81.69	88.60	93.06	95.92
SoftTriple	82.31	89.23	93.54	96.32
ProxyAnchor	**85.48**	**90.76**	**94.55**	**96.90**

(d) embedding size = 1024.

	R@1	R@2	R@4	R@8
Contrastive	80.96	87.92	92.24	95.15
Triplet	74.21	82.62	86.35	92.25
LiftedStructure	81.05	88.10	92.44	95.49
NPair	72.78	80.81	83.42	88.68
ProxyNCA	76.49	85.47	91.03	94.22
Margin	78.93	85.31	91.25	94.08
ArcFace	80.53	87.40	91.80	95.18
MultiSimilarity	82.20	88.96	93.14	96.08
SoftTriple	83.04	89.69	93.96	96.69
ProxyAnchor	**85.97**	**91.51**	**94.85**	**97.04**

ABBREVIATIONS - ACRONYMS

GPU	Graphical Processing Unit
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
ReLU	Rectified Linear Unit
CNN	Convolutional Neural Networks
SUV	Sport Utility Vehicle
POLA	Pseudo-Metric Online Learning Algorithm
LMNN	Large-Margin Nearest Neighbors
NCA	Neighborhood Component Analysis
MCML	Maximally Collapsing Metric Learning
MLP	Multi-layer Perceptron

REFERENCES

[1] H. Oh Song, Y. Xiang, S. Jegelka, and S. Savarese, “Deep metric learning via lifted structured feature embedding,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[2] B. Kulis *et al.*, “Metric learning: A survey,” *Foundations and trends in machine learning*, vol. 5, no. 4, pp. 287–364, 2012.

[3] Y. Mahdid, “Perceptron algorithm from scratch in python,” 2020. https://yacinemahdid.com/static/7c425dc2a439bb4bcc6e627eb549b010/6c315/thumbnail.jpg.

[4] Y. Avrithis, “Lecture 5: Learning,” 2019. https://sif-dlv.github.io/slides/learn.pdf.

[5] A. Mohanty, “Multi layer perceptron (mlp) models on real world banking data,” 2019. https://miro.medium.com/max/700/1*-IPQlOd46dlsutIbUq1Zcw.png.

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[7] Y. Avrithis, “Lecture 7: Convolution and network architectures,” 2019. https://sif-dlv.github.io/slides/conv.pdf.

[8] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. Berg, and L. Fei-Fei, “Imagenet large scale visual recognition challenge,” *International Journal of Computer Vision*, vol. 115, 09 2014.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[10] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015.

[11] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826, 2016.

[12] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[13] F. Schroff, D. Kalenichenko, and J. Philbin, “Facenet: A unified embedding for face recognition and clustering,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 815–823, 2015.

[14] X. Wang, X. Han, W. Huang, D. Dong, and M. R. Scott, “Multi-similarity loss with general pair weighting for deep metric learning,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[15] Y. Movshovitz-Attias, A. Toshev, T. K. Leung, S. Ioffe, and S. Singh, “No fuss distance metric learning using proxies,” in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 360–368, 2017.

[16] J. Deng, J. Guo, N. Xue, and S. Zafeiriou, “Arcface: Additive angular margin loss for deep face recognition,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4685–4694, 2019.

[17] Q. Qian, L. Shang, B. Sun, J. Hu, T. Tacoma, H. Li, and R. Jin, “Softtriple loss: Deep metric learning without triplet sampling,” in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 6449–6457, 2019.

[18] S. Kim, D. Kim, M. Cho, and S. Kwak, “Proxy anchor loss for deep metric learning,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 3238–3247, 2020.

[19] C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie, “The Caltech-UCSD Birds-200-2011 Dataset,” tech. rep., 2011.

[20] J. Krause, M. Stark, J. Deng, and L. Fei-Fei, "3d object representations for fine-grained categorization," *2013 IEEE International Conference on Computer Vision Workshops*, pp. 554–561, 2013.

[21] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines.," in *ICML* (J. Fürnkranz and T. Joachims, eds.), pp. 807–814, Omnipress, 2010.

[22] S. Chopra, R. Hadsell, and Y. Lecun, "Learning a similarity metric discriminatively, with application to face verification," vol. 1, pp. 539– 546 vol. 1, 07 2005.

[23] R. Hadsell, S. Chopra, and Y. Lecun, "Dimensionality reduction by learning an invariant mapping," pp. 1735 – 1742, 02 2006.

[24] K. Q. Weinberger, J. Blitzer, and L. K. Saul, "Distance metric learning for large margin nearest neighbor classification," in *In NIPS*, MIT Press, 2006.

[25] A. Hermans, L. Beyer, and B. Leibe, "In defense of the triplet loss for person re-identification," *CoRR*, vol. abs/1703.07737, 2017.

[26] C. Wu, R. Manmatha, A. J. Smola, and P. Krähenbühl, "Sampling matters in deep embedding learning," in *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 2859–2867, 2017.

[27] B. Harwood, V. K. B G, G. Carneiro, I. Reid, and T. Drummond, "Smart mining for deep metric learning," pp. 2840–2848, 10 2017.

[28] I. Fehérvári, A. Ravichandran, and S. Appalaraju, "Unbiased evaluation of deep metric learning algorithms," *ArXiv*, vol. abs/1911.12528, 2019.

[29] M. Schultz and T. Joachims, "Learning a distance metric from relative comparisons," in *Advances in Neural Information Processing Systems 16* (S. Thrun, L. K. Saul, and B. Schölkopf, eds.), pp. 41–48, MIT Press, 2004.

[30] J. T. Kwok and I. W. Tsang, "Learning with idealized kernels," in *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, p. 400–407, AAAI Press, 2003.

[31] S. Shalev-Shwartz, Y. Singer, and A. Ng, "Online and batch learning of pseudo-metrics," 01 2004.

[32] K. Q. Weinberger and L. K. Saul, "Distance metric learning for large margin nearest neighbor classification," *J. Mach. Learn. Res.*, vol. 10, p. 207–244, June 2009.

[33] K. Q. Weinberger and L. K. Saul, "Distance metric learning for large margin nearest neighbor classification," *J. Mach. Learn. Res.*, vol. 10, p. 207–244, June 2009.

[34] A. Globerson and S. Roweis, "Metric learning by collapsing classes," in *Proceedings of the 18th International Conference on Neural Information Processing Systems*, NIPS'05, (Cambridge, MA, USA), p. 451–458, MIT Press, 2005.

[35] G. W. Leibniz, "Memoir using the chain rule (cited in TMME 7:2&3 p 321-332, 2010)," 1676.

[36] G. de L'Hospital, *Analyse des infiniment petits pour l'intelligence des lignes courbes*. de L'Imprimerie royale, 1696.

[37] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Cornell Aeronautical Laboratory. Report no. VG-1196-G-8, Spartan Books, 1962.

[38] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Mit Press, 1972.

[39] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, pp. 193–202, 1980.

[40] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034, 2015.

[41] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," vol. 37 of *Proceedings of Machine Learning Research*, (Lille, France), pp. 448–456, PMLR, 07–09 Jul 2015.

[42] S. Bell and K. Bala, "Learning visual similarity for product design with convolutional neural networks," *ACM Trans. Graph.*, vol. 34, July 2015.

[43] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, “Signature verification using a ”siamese” time delay neural network,” in *Proceedings of the 6th International Conference on Neural Information Processing Systems*, NIPS’93, (San Francisco, CA, USA), p. 737–744, Morgan Kaufmann Publishers Inc., 1993.

[44] H. Robbins, “A stochastic approximation method,” *Annals of Mathematical Statistics*, vol. 22, pp. 400–407, 2007.

[45] M. Massie, F. A. Nothaft, C. Hartl, C. Kozanitis, A. Schumacher, A. Joseph, and D. Patterson, “Adam: Genomics formats and processing patterns for cloud scale computing,” 2013.

[46] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, M. Franklin, A. D. Joseph, and D. A. Patterson, “Rethinking data-intensive science using scalable analytics systems,” in *Proceedings of the 2015 International Conference on Management of Data (SIGMOD ’15)*, ACM, 2015.

[47] E. Hoffer and N. Ailon, “Deep metric learning using triplet network,” in *International Workshop on Similarity-Based Pattern Recognition*, pp. 84–92, Springer, 2015.

[48] K. Sohn, “Improved deep metric learning with multi-class n-pair loss objective,” in *Advances in Neural Information Processing Systems 29* (D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, eds.), pp. 1857–1865, Curran Associates, Inc., 2016.

[49] M. D. Zeiler and R. Fergus, “Visualizing and understanding convolutional networks,” in *European conference on computer vision*, pp. 818–833, Springer, 2014.

[50] M. Lin, Q. Chen, and S. Yan, “Network in network,” *arXiv preprint arXiv:1312.4400*, 2013.

[51] H. Jégou, M. Douze, and C. Schmid, “Product quantization for nearest neighbor search,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 117–128, 2011.

[52] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization. arxiv 2017,” *arXiv preprint arXiv:1711.05101*.

[53] F. Cakir, K. He, X. Xia, B. Kulis, and S. Sclaroff, “Deep metric learning to rank,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1861–1870, 2019.

[54] Y. Yuan, K. Yang, and C. Zhang, “Hard-aware deeply cascaded embedding,” in *Proceedings of the IEEE international conference on computer vision*, pp. 814–823, 2017.

[55] B. Yu and D. Tao, “Deep metric learning with tuplet margin loss,” in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 6489–6498, 2019.

[56] X. Wang, Y. Hua, E. Kodirov, G. Hu, R. Garnier, and N. Robertson, “Ranked list loss for deep metric learning. arxiv 2019,” *arXiv preprint arXiv:1903.03238*.

[57] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.