Bibliography

Why deep learning?

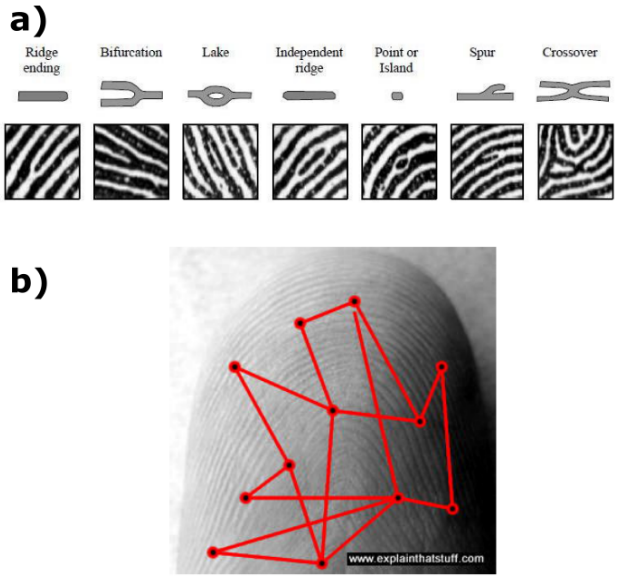

Figure 2: ***a)*** *Example of key features in fingerprints ("minutiae").*
b) *Once features are detected, their relative locations are stored to be compared with other fingerprints.*

Fingerprint recognition existed long before deep learning using classic computer vision techniques. How does this work? Why not use the same methods for violin recognition, and what are the advantages of deep-learning for this specific task?

Fingerprints are identified using feature extraction and matching [1]. Some specific features (see figure 1a) are detected using classical computer vision techniques, such as filters, thresholding, skeletonization, and pixel pattern detection.

Once these features are detected, their relative positions are stored as numeric codes (see figure 1b). If a few of these features and locations fit another fingerprint, then it's a match. This method is robust enough to identify an individual using a partial fingerprint, like those found on a crime scene. In another field, the famous song-identifying application *Shazam* uses the same method to some extent [2], with sound frequencies instead of visual features.

Figure 3: Illustration of the variety of wood aspects in violins.

If this method is so effective for fingerprints, why not use it for violins? Theoretically, it could be possible. But the aspect of a fingerprint is always roughly the same, while violin wood appearance varies widely. Figure 3 shows a few examples of different violin backs: the classic wood flames, plain wood with almost no visible flames, the complex "bird's eye" patterns, etc. Writing a single algorithm that can extract features and uniquely identify these very different violins might be very complex and time consuming.

Luckily, this task is perfect for deep learning, specifically **convolutional neural network** (CNN) architectures. It is not the purpose of this paper to explain in depth how CNN operates. But in a nutshell, a CNN is composed of several **convolutional layers** (CL), each of which extracts features from the preceding CL. Therefore, the extracted features can become more and more complex as the network gets deeper (see figure 4). A CNN can focus on simple small-scale features as well as larger complex ones, and combine said features to achieve the desired result. The scale, complexity, and combination of extracted features will depend entirely on the training process.

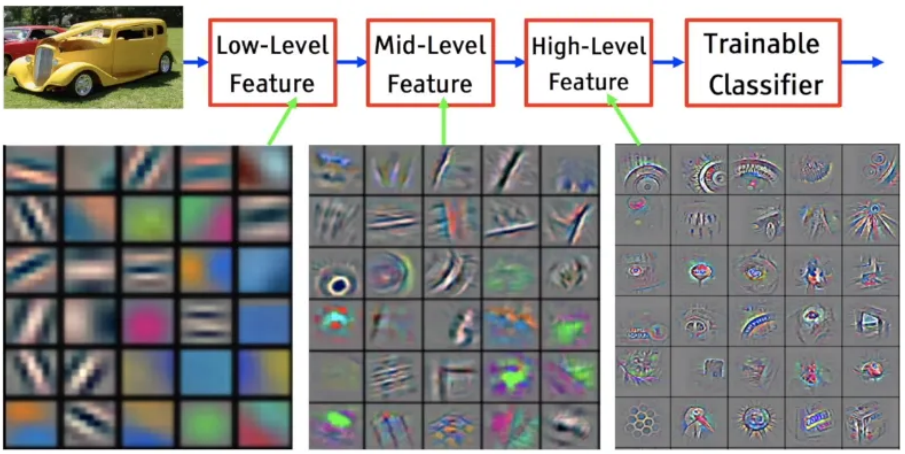

Figure 4: Different levels of feature complexity on different stages of a CNN (image source).

In the last decades, face recognition has been a subject of intense research for biometrics, surveillance, and other more or less desirable applications. Techniques used in face recognition are interesting regarding violin recognition, as they are based on CNN training. Moreover, the challenges encountered by face recognition applications are very similar to our own. Indeed, face recognition applications are open-set classifications.

Typical deep learning classifiers are **closed-set classifiers**. Such a classifier can only recognise classes found in the training dataset. For example, adding new instruments to a violin recognition model implies retraining the model with these new violins in the dataset. In our case Cozio, the violin database, is regularly updated, with batches of new instruments added several times yearly. Therefore, a closed-set classifier is really not a practical option.

An **open-set classifier** can recognise classes that were not in the training dataset. For violin recognition, it means that as soon as a new violin is "shown" to the model, from that point on, the model can recognise it without any retraining. Therefore, updating the database does not require model retraining.

While classical classifiers are relatively simple to build and train, open-set classifiers are much more complex and rely on **deep metric learning** (DML). In the next chapter, we will explain what DML is. We will also show and compare two training methods tested for this project: **triplet loss** and **arcface**.

Deep Metric Learning.

A classical closed-set classifier is straightforward: the model takes an image as an input and outputs a class (i.e. an instrument's ID). An open-set classifier must be able to classify an indefinite number of classes. The model's output cannot, therefore, be a class, as we cannot simply add new classes in the last layer of the model without retraining. Instead, open-set classifiers rely on deep metric learning (DML).

In computer vision, the idea behind DML is to use a CNN to transform an image into an **embedding** and not into a class. Embeddings can be thought of as the coordinates of a point, like GPS coordinates. However, GPS coordinates are 2-dimensional (latitude and longitude), while embeddings are N-dimensional vectors. An embedding represents the coordinates of a point in an N-dimensional space called the **embedding space**. The location of embeddings in this space depends on the aspect of input images, according to the features extracted by the CNN.

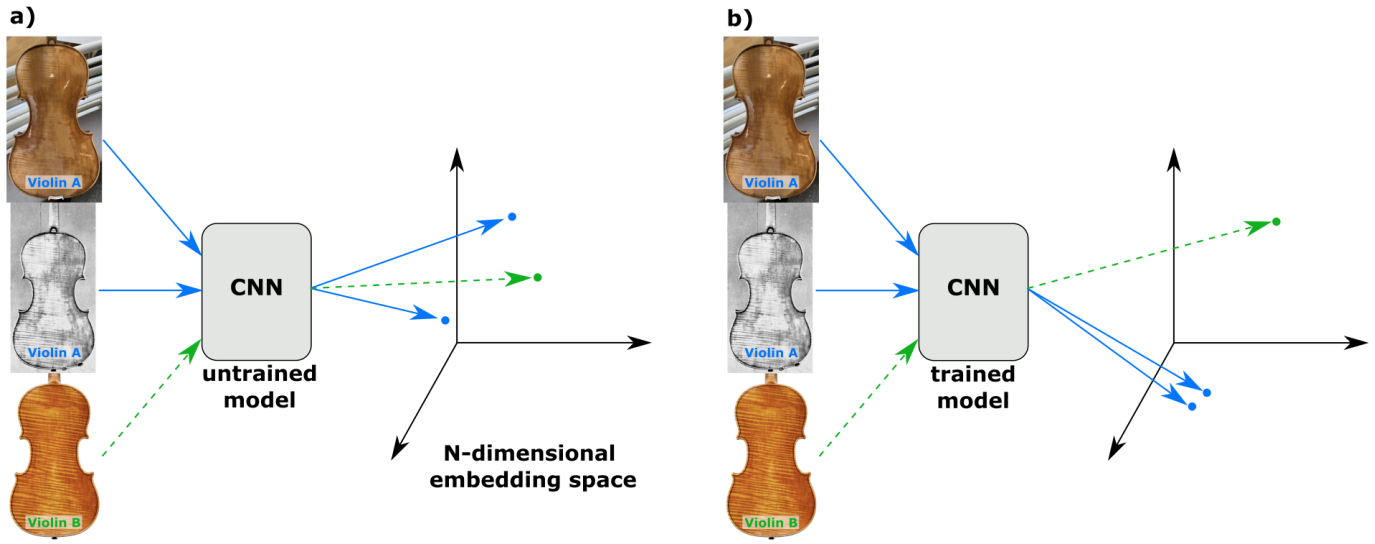

Figure 5: Deep metric learning transformation of violin images into embeddings, with an untrained (a) and a trained (b) model.

As shown in figure 5a, an untrained model will place images randomly in the embedding space. The objective of DML is to train the model so that similar images end up close to one another while different images end up far apart. But it cannot just be any visual similarity, as illustrated in Figure 5b. Here, the two pictures of violin A are very different in appearance, one being black and white and grainy, the other in color with light reflections and a background. Violin B picture is studio-quality with perfect lighting. We want the model to focus on meaningful features, like wood flame patterns, and ignore details like background, light reflections, or picture graininess to determine the location of the embeddings.

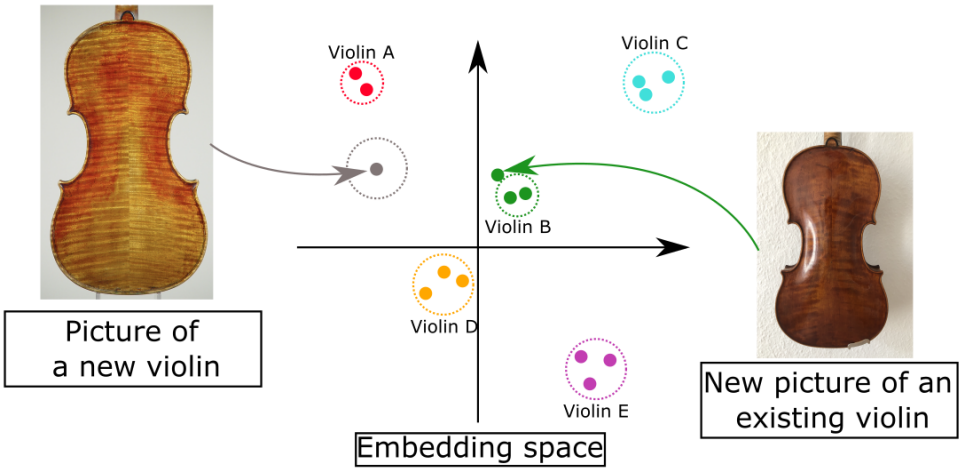

Figure 6: Illustration of how a trained model locates embeddings of a new violin, and of an existing violin, in a 2D embedding space containing embeddings from the database.

Then, how can DML models using embeddings be used as classifiers? Let's imagine an ideally trained model for violins. Such a model would assign a small part of the embedding space to one and only one violin, regardless of the quality or variations of the pictures. This location will depend on meaningful visual features the model learned to see during training. If new violins are added to the database, the model will assign them to new unoccupied locations in the embedding space, away from existing violins. Finally, when end users take a picture of a violin to check what this violin is, all they have to do is get this picture through the model and transform it into an embedding. If the violin exists in the database, this embedding will be situated right next to embeddings of an existing violin (see violin B in figure 6). If not, the model will put this picture's embedding far away from all other embeddings.

With an ideal model, a simple computer program that checks the distance between embeddings is enough to determine: 1. if a violin exists in the database and 2. what the violin ID is. Unfortunately, in the real world, even the best model isn't guaranteed to be ideal. Therefore, it is preferable to introduce some manual confirmation. For example, a computer program can fetch the five embeddings closest to the user's picture embedding. The user can then compare pictures to see if one corresponds to his violin.

Training methods: triplet loss and arcface.

For over a decade, research in DML has produced multiple training methods to achieve the results described in the previous chapter. Two of these methods were tested and used for our violin recognition project. These methods are **triplet loss** and **arcface**.

We will describe these techniques as clearly and concisely as possible. However, these descriptions might get technical, especially for arcface, which relies on softmax loss. Therefore, basic knowledge of loss functions and neural network training is preferable.

Triplet loss:

Triplet loss is perhaps the most well-known training method in DML. It was also considered the most reliable method until the emergence of angular-softmax based losses like SphereFace, CosFace, and ArcFace. Even if the concept has existed since the early 2000s, it has only been introduced in the field of face recognition in the 2015 paper "*FaceNet: A Unified Embedding for Face Recognition and Clustering*" by Schroff & al [3]. The principle of triplet loss is straightforward, as depicted in figure 7.

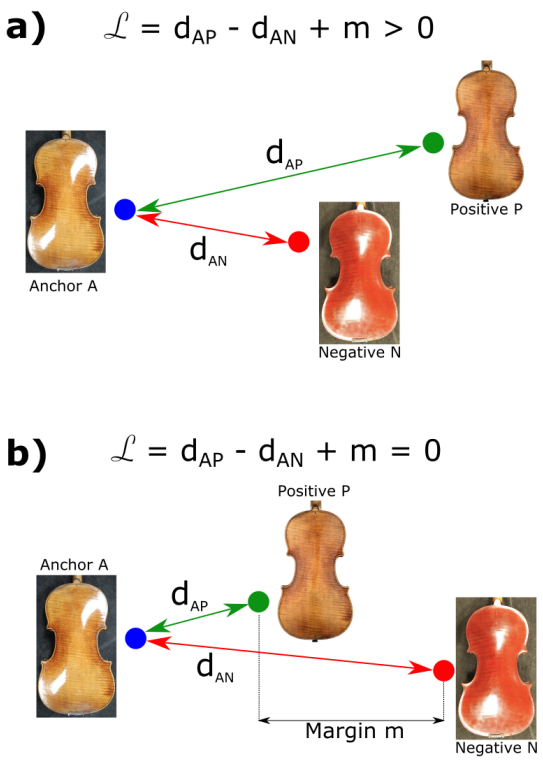

Figure 7: triplet loss principle. a) before training, the negative image is closer to the anchor than the positive. b) after training, the positive has been brought closer and the negative farther from the anchor, leaving a distance m between the positive and the negative.

To calculate the loss function, the model transforms **three** images into embeddings. The first image is named the **anchor**, the point of reference. The second image is the **positive** image, which belongs to the same class as the anchor (i.e. same violin). The third image, the **negative** image, is from a different class. The training's goal is to minimize the distance between the anchor and positive embeddings and to maximize the distance between the anchor and negative embeddings.
The loss formula is:

$$\mathcal{L} = \max(\ 0\ ,\ d_{AP} - d_{AN} + m) \qquad (0)$$

Where d_{AP} is the distance between the anchor and the positive embeddings, d_{AN} the distance between the anchor and the negative embedding, and m is the margin. Using this loss function, an ideally trained model will organize embeddings into same-labeled clusters like in figure 6, with every cluster separated by at least the margin m.

While the triplet loss theory is relatively simple, its implementation presents many difficulties. One difficulty is the selection of relevant triplets, called **triplet mining**. Triplet mining refers to the process of selecting triplets fed to the model. Indeed, the number of triplet

combinations increases exponentially with the number of samples in the dataset, and most of these triplets are irrelevant. For a triplet to be relevant, it has to satisfy at least the following requirements:

1. The positive sample is the same class as the anchor.
2. The negative is from a different class than the anchor.
3. The loss has to be greater than 0.

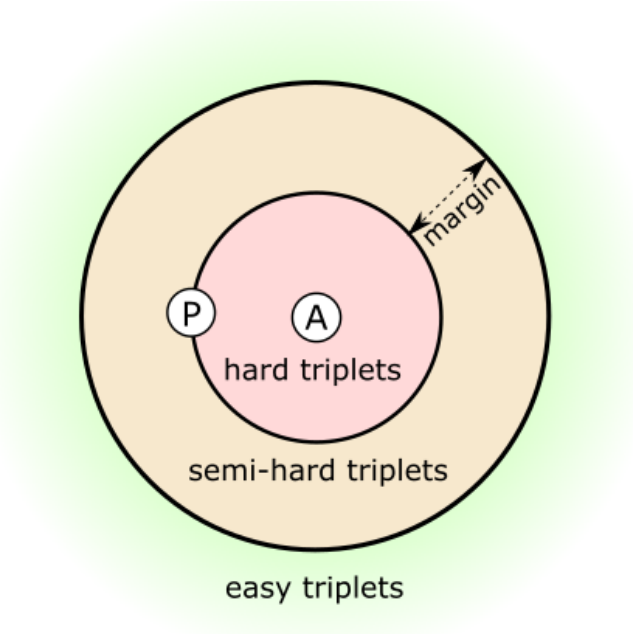

Figure 8: The three kinds of triplets according to the position of the negative embedding, given an anchor (A) and a positive (P).

Triplets with a loss lower than 0 are named "**easy triplets**" (see figure 8). These triplets' negative embedding distance from the anchor is greater than the positive distance plus the margin. Among the valid triplets, we can distinguish between hard and semi-hard triplets. **Hard triplets** have the negative closer to the anchor than the positive, while **semi-hard triplets** have the negative within the margin. Choosing which triplets appear during training has a significant impact on the results. One can feed the model only hard triplets, hard and semi-hard triplets, use only the hardest triplets in the whole dataset, etc. Some papers even argued that choosing assortments of easy positive/hard negative and hard positive/easy negative can result in a more robust model [4,5]. The possibilities are numerous, but it requires control over the batches' content. Moreover, converting images into embeddings is necessary to know the nature of a triplet (easy, hard, or semi-hard). The process of triplet mining makes triplet loss harder to implement than classical classifiers.

Several mining strategies exist. **Offline mining** requires computing all embeddings of the training set at the beginning of each epoch. Using these embeddings it is possible to create a list of triplets of interest (e.g. the hardest triplets or any desired combination) and to form training batches according to this list. Thus, offline mining offers complete control over the batches' content. However, this process presents two main drawbacks: 1. computing all images embeddings from the dataset is excessively time-consuming, and 2. the model weights change after each batch, therefore the nature of triplets can also change during an epoch. There is no guarantee that a hard triplet at the beginning of an epoch doesn't become easy by the end of that epoch.

Online mining is another method that doesn't require computing every training embeddings at the beginning of each epoch. Instead, batches are generated randomly, like in regular batch gradient descent. The relevant triplet selection then occurs within each batch. This process is simpler and faster. However, the possible triplets combinations are limited to the batches' content and depend entirely on a random process.

Apart from triplet mining, another downside of using triplet loss is that the model is prone to embedding collapse, a phenomenon comparable to overfitting. Models trained using triplet loss tend to give all embedding the exact same value. To avoid this, we can use the same techniques usually used to avoid overfitting: regularization, dropout, control over the learning rate, etc. Another practice that is very effective regarding embedding collapse is to project all embeddings onto a hypersphere using L2 normalization on the embedding layer.

ArcFace:

Because of triplet mining and embedding collapse, training a model using triplet loss is time-consuming, hard to implement and unstable. To overcome these problems, face recognition researchers found new training methods. The most notable new techniques are derived from **softmax loss** and are often grouped under the designation "**angular softmax loss**". This term generally refers to SphereFace, CosFace and ArcFace [6, 7, 8], three approaches based on the same concept.

Before we describe angular softmax losses, we must explain what softmax loss is. Softmax loss is a well-known loss function typically used for multi-class deep learning classifiers. It is the combination of **softmax activation function** and **cross-entropy loss**.

A *softmax function* transforms the raw output of a neural network into a vector of **probabilities**. This transformation ensures that all of the vector's components are **positive** and their **sum** adds up to **1**. This function is:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}} \qquad (1a)$$

Where σ is the softmax function, N is the number of classes/number of neurons in the input vector, and z_i is the i^{th} component of this input vector.

Cross entropy measures the distance between the predicted probability distribution of a classification model and the real distribution. This function outputs a number between 0 and 1, 0 being a perfect prediction. The goal is to get the model's output as close to 0 as possible. It is formulated as:

$$\mathcal{L} = -\sum_{i=1}^{N} y_i \log(\hat{y}_i) = -\sum_{i=1}^{N} y_i \log(\sigma(z_i)) \qquad (2a)$$

With $\hat{y}_i$ the predicted probability and y_i the true probability value for the i^{th} class C_i.

Softmax loss is ideal for multi-class closed-set classifiers, but cannot perform open-set classification. However, by performing relatively small changes to the classic softmax loss, **angular softmax losses** (ASL) have proven very effective at this task. Figure 9 (source: [3]) illustrates the concept of ASL. In this figure, each point represents a two-dimensional embedding from a face recognition toy dataset. This dataset has 8 different classes/identities, each represented by a color. The first and second rows present the embeddings distribution in Euclidean and angular space, respectively.

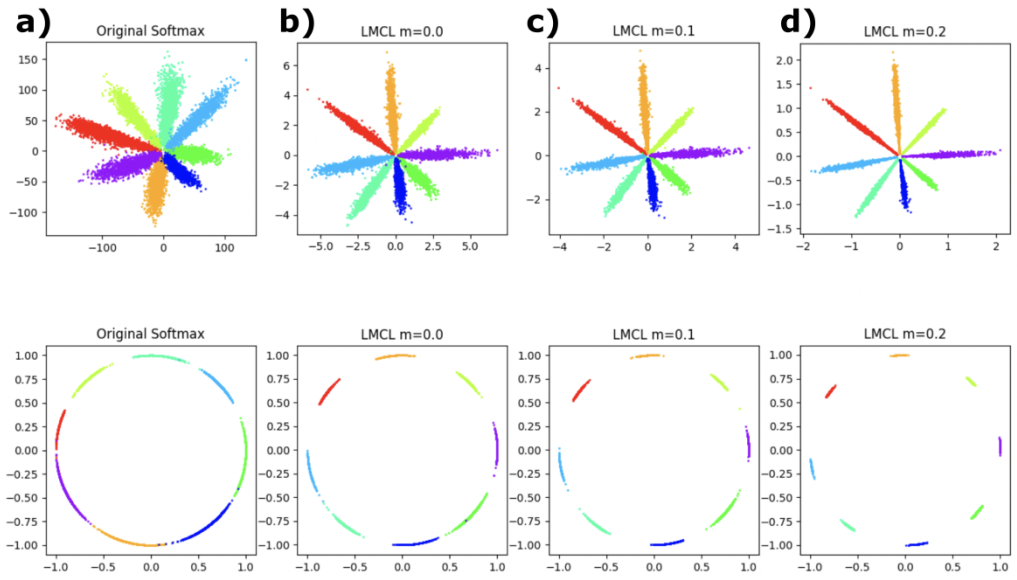

Figure 9: Illustration from CosFace paper [7]. *"A toy experiment of different loss functions on 8 identities with 2D features. The first row maps the 2D features onto the Euclidean space, while the second row projects the 2D features onto the angular space." a) softmax loss, b) c) and d) CosFace with increasing margin value.*

Figure 9a shows embeddings for a model trained with a classical softmax loss. In this model, the embedding layer precedes the softmax layer. We can observe that features learned by softmax-loss have intrinsic angular distribution. The boundaries between classes, also known as decision boundaries, are intrinsically angular. That is why the embeddings are shown in the angular space in the lower part of Fig. 9. We can observe that embeddings tend to spread across the angular space. That's because there is no incentive in softmax-loss for intra-class compactness or inter-class discrepancy. This is why softmax-loss cannot perform open-set classification: any embedding from a new class will get projected in a region of the embedding space already occupied by another class. The idea of ASL is to introduce an **angular margin** in the softmax loss formula. To explain this, we will have to delve further into mathematical explanations… and it will involve vector calculation.

Let's go back to equation (1a). z_i is the i^{th} component of a fully connected layer, which has the embedding layer as an input. By definition, z_i can be expressed as :

$$z_i = \mathbf{W}_i^T \mathbf{x} + b_i \qquad (3)$$

With $\mathbf{W}_i$ the weight vector associated with the i^{th} class/neuron, b_i the bias, and $\mathbf{x}$ the embedding vector. Using (3), (1a) becomes:

$$\sigma(z_i) = \frac{e^{\mathbf{W}_i^T \mathbf{x} + b_i}}{\sum_{j=1}^{N} e^{\mathbf{W}_j^T \mathbf{x} + b_j}} \qquad (1b)$$

If the model is given an image from the i^{th} class, the real label vector $\mathbf{y}$ in equation (2a) becomes 0 except for the i^{th} class where $y_i = 1$. Therefore, (2a) becomes:

$$\mathcal{L} = -\log(\sigma(z_i)) = -\log \frac{e^{\mathbf{W}_i^T \mathbf{x} + b_i}}{\sum_{j=1}^{N} e^{\mathbf{W}_j^T \mathbf{x} + b_j}} \qquad (2b)$$

We introduce the angle θ_j between the embedding vector $\mathbf{x}$ and the weight vector for the j^{th} class $\mathbf{W}_j$ using $\mathbf{W}_j^T \mathbf{x}_i = \|\mathrm{W}_j\| \|\mathrm{x}_i\| \cos(\theta_j)$. To make the model only depend on θ_j, we can set individual weight norms $\|W_j\| = 1 \quad (\forall j)$ using L2 normalization and fix the bias $b_j = 0$. We also project all embeddings onto a hypersphere of radius s using L2 normalization on $\mathbf{x}$. The formula (2b) then becomes:

$$\mathcal{L} = -\log \frac{e^{s\cos(\theta_i)}}{e^{s\cos(\theta_i)} + \sum_{j=1, j\neq i}^{N} e^{s\cos(\theta_j)}} \tag{2c}$$

As a reminder, the purpose of any loss function is to decrease to 0. In (2c), it means the loss function goal is:

$$e^{s\cos(\theta_i)} > e^{s\cos(\theta_j)} \;\Leftrightarrow\; \cos(\theta_i) > \cos(\theta_j) \quad \forall j \neq i \tag{4}$$

with θ_i the angle between the embedding vector and the correct class, and θ_j the angle for any other class. The idea is to introduce a constant m which will add a constraint on θ_i. For example, CosFace loss is defined as:

$$\mathcal{L} = -\log \frac{e^{s(\cos(\theta_i) - m)}}{e^{s(\cos(\theta_i) - m)} + \sum_{j=1, j\neq i}^{N} e^{s\cos(\theta_j)}} \tag{5}$$

As $\cos(\theta_i) - m < \cos(\theta_i)$, the model will have to constrain θ_i even more to achieve (4). Therefore the intra-class discrepancy is reduced. Figure 9b), c) and d) show the result of a model trained with CosFace loss. It can be observed that embeddings occupy a smaller region of the angular space for larger values of m.

ArcFace uses the same concept, but with the margin applied directly to the angle θ_i:

$$\mathcal{L} = -\log \frac{e^{s(\cos(\theta_i + m))}}{e^{s(\cos(\theta_i + m))} + \sum_{j=1, j\neq i}^{N} e^{s\cos(\theta_j)}} \tag{6}$$

Compared to triplet-network, building and training these losses is easier. For this, it is necessary to

1. L2-normalize the embedding layer.
2. build a custom softmax layer with the bias set to 0 and each weight vector L2-normalized.
3. build a custom loss function that computes the loss with the scale factor s and the margin m .

The model can then be trained like any classic softmax classifier.

In this project, ArcFace was preferred to other losses like SphereFace or CosFace because ArcFace was successfully used in Kaggle competitions (humpback whale identification) and achieved 2cd and 3rd place.

References:

[1] Bhattacharya, S., & Kalyani, M. (2011). Fingerprint Recognition Using Minutiae Extraction Method. *Proc. of International Conference on Emerging Technologies (ICET-2011): International Journal of Electrical Engineering and Embedded Systems*.

[2] Jovanovic, J. (n.d.). Music Recognition Algorithms: How Does Shazam Work? Toptal. Retrieved April 28, 2023, from https://www.toptal.com/algorithms/shazam-it-music-processing-fingerprinting-and-recognition

[3] Schroff, F., Kalenichenko, D., & Philbin, J. (2015). *Facenet: A unified embedding for face recognition and clustering.* In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 815-823).

[4] Poorheravi, P. A., Ghojogh, B., Gaudet, V., Karray, F., & Crowley, M. (2020). Acceleration of large margin metric learning for nearest neighbor classification using triplet mining and stratified sampling. *arXiv preprint arXiv:2009.14244*.

[5] Xuan, H., Stylianou, A., & Pless, R. (2020). Improved embeddings with easy positive triplet mining. *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 2474-2482.

[6] Liu, W., Wen, Y., Yu, Z., Li, M., Raj, B., & Song, L. (2017). Sphereface: Deep hypersphere embedding for face recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 212-220).

[7] Wang, H., Wang, Y., Zhou, Z., Ji, X., Gong, D., Zhou, J., ... & Liu, W. (2018). Cosface: Large margin cosine loss for deep face recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 5265-5274).

[8] Deng, J., Guo, J., Xue, N., & Zafeiriou, S. (2019). Arcface: Additive angular margin loss for deep face recognition. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (pp. 4690-4699).