# Assignment 2 Part 2 - Face Verification

Introduction to Deep Learning
Emeritus & Carnegie Mellon University

## 1 Overview

Have you ever wondered how facial recognition works? In this assignment, we'll be tackling another real-world application of DL: face verification.

**Objective**: Given two pictures of human faces, determine if the faces belong to the same person.

**As with the last assignment, make sure you start this assignment early.** This pipeline will be a little complex, in order to give you exposure to what real DL work is like, and introduce you to some key concepts like transfer learning and embeddings.

### 1.1 Pipeline Overview

We're going to take an indirect approach to this problem, using transfer-learning. You could take a more direct approach, but pipelines for that tend to be much more complicated to both implement AND train. Even though this approach is indirect, it's still valid, and indirect approaches with transfer learning are very common in DL.

1. Classification
    - Train a computer vision (CV) model to classify which person an image of a face belongs to
    - The model will be a variation of **ResNet**, a landmark CV model that's still used today for benchmarking and DL theory.
        - It's important you're familiar with this model, as it introduces important tricks that enable training of very deep models, and because it appears a lot in research.
2. Verification
    - We can use the model we trained for classification to do verification as well.
    - Feed two pictures of faces into the same CV model we trained above, but instead of taking the last layer's output (which we normally give to `CrossEntropyLoss`), take the output of **second-to-last layer** for both inputs.
        - This penultimate output is called an "embedding", it contains an abstract representation of the input.
    - We then compare each embedding using some distance measure, and if the distance is sufficiently small, say that the images are of the same person.

There's a lot of intermediate details in this pipeline that we'll have to cover. Buckle in!

### 1.2 Important: Grading

**You will only be graded on how you do on the verification test dataset** and NOT on classification. But you still will need to do classification in order to complete the assignment.

## 2 Classification

Let's cover classification in more depth.

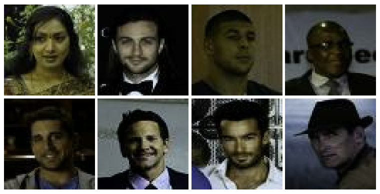


Figure 1: Examples of faces in our dataset.

Say we have a 380,639 RGB 64x64 images of the faces of 4,000 different people. We want to identify which person is in each photo. How do we do this?

One way to handle this is via a simple classification model.

Here's the pipeline:

1. We group several images into a batch shaped `(batch_size, in_channels, height, width)`, and pass it into a multi-layer 2d CNN.
2. The output of this multi-layer 2d CNN is a batch of **embeddings**, shaped `(batch_size, num_channels, height, width)`.
    - As we've discussed earlier, embeddings are generally abstract representations of the original input images.
    - But in some cases, embeddings or parts of embeddings can end up representing easily interpretable information, like explicit representations of things like the distance between ears, the relative positioning and size of the eyes, the shape of the head, etc.
3. We average these embeddings across the `height` and `width` dimensions

Note that interpreting a model as being an extractor + classifier works for many different types of models in multiple problem spaces. In fact, the phoneme classifier from the first homework can even be interpreted in this way. It's also very common to take the embedding from the second-to-last layer of a trained model and use it for things.

## 3 Verification

In verification, the goal is to determine whether a given pair of faces belong to the same person.

However, **the people in the verification dataset were NOT in the original classification dataset**. In other words, we want a general model that can verify that two photos are of the same person without assigning specific IDs for that person.

Why? Because there are over 7 billion people on earth; classifying across all 7 billion of those people in order to compare them is nearly impossible and unnecessary.

This motivates our indirect approach: what if we compared their facial features?

Remember that the embeddings from our pretrained classification model encode information about the face in the photo. So if the embeddings are similar enough (i.e. the facial features are similar enough), the faces likely belong to the same person.

### 3.1 Measuring Similarity

But how do you precisely measure how 'similar' two faces are?

Remember that we're working with vector embeddings, which we can interpret as just 'points' in some high dimensional space.

**So we could just measure the distance between these two vectors, and apply some kind of threshold to determine if they're similar or not similar.**

### 3.2 Euclidean Distance

**Note: Don't use this, just explaining.**

One way to measure the distance between two points in space is by using **euclidean distance**.

$$\text{Euclidean}(p, q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

Where $p$ and $q$ are $n$-dimensional vectors.

**However**, euclidean distance ranges from $[0, \infty]$, so we can't really neatly place a threshold.

#### 3.2.1 Cosine Similarity

Instead, we recommend you use **cosine similarity**, which is neatly bounded between $[0, 1]$.

$$\text{CosineSim}(p, q) = \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} p_i^2} \sqrt{\sum_{i=1}^{n} q_i^2}}$$

(Use `nn.CosineSimilarity` for this.)

After getting your score, you can then apply a threshold of $\geq 0.5$ to determine sufficient similarity.

Bam, we have an approach for our problem! Now to figure out what model to use.

## 4 Model: ResNet

The ResNet model was created in 2015 by He, Zhang, Ren, and Sun ↗.

It was based on the VGG model ("Visual Geometry Group" at Oxford), developed by Simonyan and Zisserman ↗ for the ILSVRC 2014 challenge.

Its most notable improvement over VGG was the introduction of **"Residual Connections"** (aka **"Skip Connections"**) to help facilitate the training of very deep NNs.

It was remarkably successful, and has since come to replace VGG as the standard CV model. It's now considered the field's baseline for benchmarking and quick usage. Even in mid-2021, a ResNet variation called MobileNetV3 is allegedly planned to be used for the NeuralHash algorithm in every Apple iPhone (source ↗).

### 4.1 What are Residual/Skip Connections?

The key to ResNet's success are its **skip/residual connections**[1]. They're so useful, that even in 2021, skip connections now appear in almost every mainstream DL architecture (transformers, YOLOv3, MLP mixer, etc) across many problem spaces.

So what are they?

Residual/skip connections are simply when you skip a tensor past one or more layers and combine it with the output of some later layer.

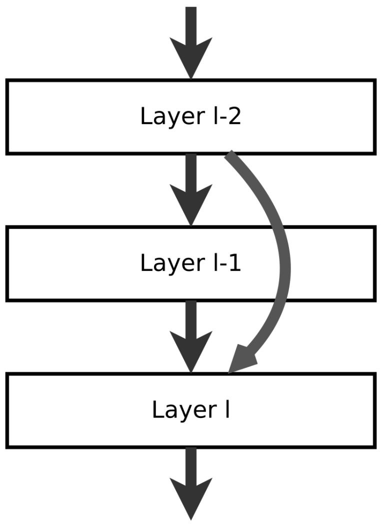


Figure 2: The output of layer $l-2$ is fed into both $l-1$ AND added to $l-1$'s output. Assume the output of layer $l-2$ and $l-1$ are the same shape for this example.

Usually, we add the skipped tensor to a downstream output, but there are variations involving averaging, pooling, etc.

[1]Terms are interchangeable, I prefer 'skip' as it's more intuitive

## 4.2 Why do Skip Connections help?

While there's no universal consensus on why these work, we do have some generally-accepted hypotheses.

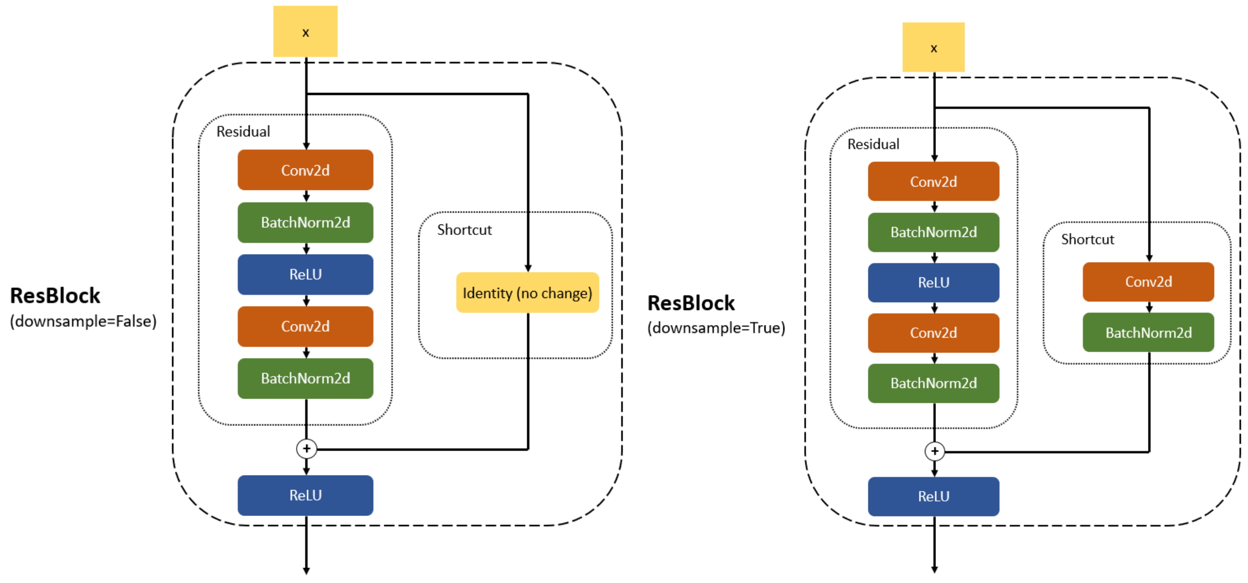
One significant hypothesis is that important information in the original input can be lost or become muddled in later parts of the model. This especially applies for extremely deep models (dozens to hundreds of layers), and also depends on your choice of activation functions.

So adding back earlier signals to downstream layers may help preserve the original signals' strength, while still allowing the intermediate layers to extract information.

We won't get into too much detail, but the main takeaways are that skip connections may help preserve signals from earlier in the model, and that they're widely accepted as being helpful in very deep models.

## 4.3 ResNet Architecture

The basic unit of ResNet is the "residual block". These blocks are usually structured identically throughout the model; we just tweak their param sizes and the overall number of blocks there are.



(a) ResBlock with no downsampling (b) ResBlock with downsampling

Figure 3: Typical ResBlock, diagram emphasizing how shortcut strategy changes depending on `downsample` parameter

Notice the skip connection from the input of the block to just before the last `ReLU`.

Also note that the architecture changes based on the `downsample` parameter.

- If `downsample=False`, the first `Conv2d` in Residual has `stride=1`, and the shortcut becomes an identity transformation (no change).
  - The output will have the same width and height as the input, although the number of channels may have changed.
- If `downsample=True`, the first `Conv2d` in Residual has `stride=2`, and the shortcut becomes an additional `Conv2d`.

Don't worry if this seems confusing; we'll walk you through implementing this in the notebook.

## 4.4 Recommended Architecture

The below table is in order from top to bottom.

| |
|---|
| `Conv2d(in_channels=3, out_channels=64, kernel_size=3, stride=1, padding=1)` |
| `ResBlock(in_channels=64, out_channels=64, kernel_size=3, downsample=False)` |
| `ResBlock(in_channels=64, out_channels=64, kernel_size=3, downsample=False)` |
| `ResBlock(in_channels=64, out_channels=128, kernel_size=3, downsample=True)` |
| `ResBlock(in_channels=128, out_channels=128, kernel_size=3, downsample=False)` |
| `ResBlock(in_channels=128, out_channels=256, kernel_size=3, downsample=True)` |
| `ResBlock(in_channels=256, out_channels=256, kernel_size=3, downsample=False)` |
| `ResBlock(in_channels=256, out_channels=512, kernel_size=3, downsample=True)` |
| `ResBlock(in_channels=512, out_channels=512, kernel_size=3, downsample=False)` |
| Average along last two axes (see section 4.4.1 below) |
| `Linear(512, 4000)` |

Notice that `downsample=True` every time we change the channels. We'll walk you through how to implement downsampling in the notebook.

### 4.4.1 Averaging along last axes

The averaging step is needed to convert our batch of 3D image embeddings into a batch of 1D embeddings.

We need to do this in order to calculate cosine similarity in verification, which is only typically used between 1D vectors. However, we still do this for classification as well, because we try to get the network to make its 1D embeddings as rich as possible.

```
Before averaging:
(batch_size, num_channels, height, width)

After averaging:
(batch_size, num_channels)
```

To do this, you can simply perform `.mean(dim=2)` twice.

## 5 Dataset

### 5.1 Classification Files

You'll receive two folders for classification:

- `classification_train`
  - Contains 4,000 subfolders (1 for each person), which contain images of that person's face.
  - Total: 380,639 RGB images of faces, each sized 64x64
- `classification_val`
  - Same 4,000 people, but only 2 photos per person (8,000 total).

These can be very easily loaded in with `torchvision.datasets.ImageFolder`. Read the documentation for it online to understand how to call and use it.

Remember, there is no test data for classification, as you'll only be graded for verification.

### 5.2 Verification Files

You'll receive one folder and two text files for this task.

- `verification_test`
  - No subfolders; contains around 69,000 image files of faces (also RGB and 64x64), each with an integer ID in its filename.
- `verification_pairs_test.txt`
  - Text file, where each line is space separated (no headers). For example:

    ```
    verification_data/00041961.jpg verification_data/00044353.jpg 0
    ```

  - The first two strings are the two images to be compared, and the last integer is a boolean (1 if they're the same person, else 0).
- `verification_pairs_val.txt`
  - Same as above. Just in case you want to validate your performance on verification.

You can easily load these in with a custom `Dataset` class we're giving you called `VerifyDataset`, located in `utils.py`; make sure you understand how it works.

## 6 Training Tips

Last section! We'll cover some optimizations for **speed** of training and inference that we recommend you use to cut down training time.

### Introduction: Computer Vision is Expensive!

A batch of 64 images shaped 64x64 large is represented by $64 * 64 * 64 * 3 = 786,432$ 32-bit floating point numbers. Each layer performs multiple operations on potentially all of these numbers.

And we intentionally gave you smaller images to be kind to your compute needs. A practical CNN may need to process a 1024x512 image. A batch of just 16 of these is $1024 \times 512 \times 3 \times 16 = 25,165,824$ numbers!

In addition, the models themselves are often massive. Their size is well deserved, given that they need to adequately represent the dynamics of objects, shapes, patterns, and their interactions with light in a 3D world. CV models often take gigabytes of memory. And because gradients need to be stored as well, your memory requirement doubles.

You get the point; CV is expensive!

Below, we'll recommend two techniques you can apply that will reduce the time and memory footprint you'll need to train your model dramatically.

## 6.1 Technique 1: Automatic Mixed Precision (AMP)

As stated above, in PyTorch we mostly work with 32-bit `FloatTensor`s. But for some parts of your pipeline, this level of precision is often unnecessary, and just takes up extra memory and computation.

Here's where **Automatic Mixed Precision (AMP)** comes in.

### 6.1.1 What is AMP?

AMP intelligently detects which parts of your pipeline could use 16-bit precision without hurting performance. It can even find opportunities to re-scale numbers if it'll make scaling to 16-bit precision possible.

The benefit of this is so significant that people frequently report between a 30% to a 300% speedup to training and inference. When we added it, training time per epoch went from 23 to 13 minutes!

For a deeper introduction to AMP, we recommend you read this article by Nvidia.

### 6.1.2 `torch.cuda.amp`

Thankfully, `torch` has access to an implementation of this that's reasonably simple to use.

We recommend you read this tutorial. It demonstrates how to use AMP very clearly!

The main takeaway is that you'll need to modify your `train_epoch()` method in order to get this working. We'll remind you to check the above tutorial at the appropriate part of the notebook.

## 6.2 Technique 2: Center Loss

**Center loss** is another loss function you can use in addition to `CrossEntropyLoss`. And we mean "in addition" literally, because you add the loss value generated by `CrossEntropyLoss` to the loss value outputted by `CenterLoss`.

It should help you converge faster, not because it makes the computations faster like AMP did, but because center loss will help your model converge to a better minimum in fewer iterations.

### 6.2.1 Motivation: Separating Clusters in Feature Space

Center loss pushes embeddings of different classes further away from each other.

Why is this important? Because our method of measuring similarity between faces relies on a distance metric. So in embedding space, we want faces that are similar to be close to each other, and faces that are different to be farther away.

Here's a (very simplified) visualization of center loss's impact on training.

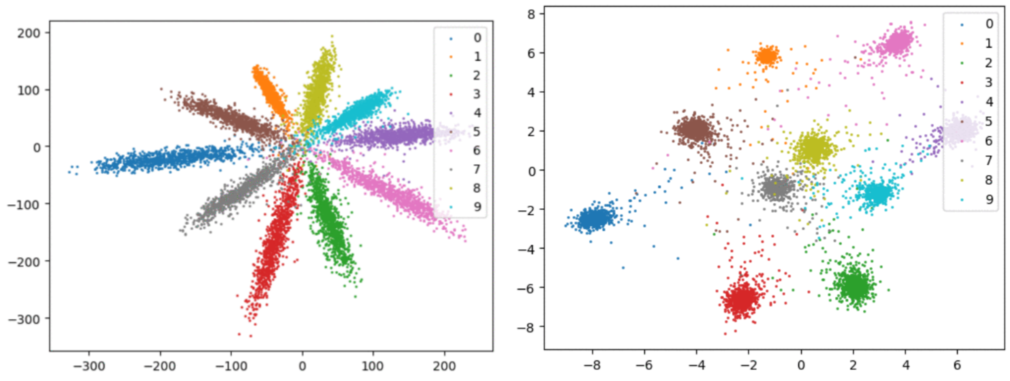


Figure 4: Before and after applying center loss

Each point in the plot above represents a single embedding's location in embedding space. Each point is color-coded by class.

There are only 10 possible classes in this picture (for us, this would be 4,000 different people). Also notice that the embeddings are just 2-dimensional vectors, instead of the 512-dimensional arrays from our recommended architecture.

The main takeaway of the above diagram is that the points close to the middle of the plot are very close together, so it will be hard to differentiate between them using cosine similarity.

### 6.2.2 How Center Loss Works

To address this, center loss coaxes clusters into tighter groups.

To do this, it chooses a random center point for each class. It then penalizes embeddings that are far away from those centers. The further away an embedding is from the center point, the more it'll be penalized.

### 6.2.3 How to use it

We've provided you a full implementation of it that's modified to work with mixed precision. For the code and a detailed readme on how to use it, see here .