Introduction to Machine Learning

Report on the Retrieval Competition

Paolo Fabbri Davide Sbreglia

University of Trento, Italy

paolo.fabbri@studenti.unitn.it davide.sbreglia@studenti.unitn.it

1 Introduction

1.1 Task Description

In the Image-to–Image Competition of the Introduction to Machine Learning course, we were tasked with recognizing celebrities when their appearance changes substantially: each natural *query* photograph of a celebrity must be paired with ten synthetic *gallery* renderings of the same identity produced in an artistic style.

Jared Leto *(natural query photograph)*

Joker *(sinthetic gallery rendering)*

The competition dataset comprises two splits: training and test. The training split contains roughly five thousand images ($n_{\text{train}} = 4994$) grouped by identity and is used to expose models to both natural and synthetic representations of each celebrity. By contrast, the test split is reserved exclusively for evaluation: about three thousand images ($n_{\text{test}} = 2935$) are organized into a *query* folder with unmodified photographs and a *gallery* folder with stylized counterparts.

Performance is measured with three complementary accuracy metrics whose weighted sum yields the final score (maximum 1000 points). Top–1 counts a correct match at rank one; Top–5 checks whether any of the first five predictions is correct; Top–10 applies the same criterion to the full returned list. Weights of 600, 300, and 100 points for Top–1, Top–5, and Top–10, respectively, place greater emphasis on precision at the highest ranks.

1.2 Overview of Approaches

Before any training or retrieval, we had to fetch and unpack the competition images from Google Drive. The script `data_extraction.py` automates this: it checks whether `dataset.zip` is already present in the working directory; if not, it downloads the archive via `gdown` from the official Drive link, extracts its contents into the project folder, and finally deletes the zip to save space.

Once the files are in place, `data.py` handles preprocessing and dataset objects. It resizes images to 224×224, converts them to tensors, and normalizes colors with ImageNet statistics; during training it adds light randomness (random resized crop, horizontal flip) to reduce overfitting. It provides a supervised `ClassifyDataset` (for head training) and an unlabeled `RetrievalDataset` (for embedding extraction over query/gallery).

We build an ImageNet-pretrained backbone (`efficientnet_b0`, `resnet50`, or `vit_b_16`), freeze its weights, and attach a small classification head. For retrieval, we convert the trained network into an *embedder* that outputs ℓ_2-normalized vectors (unit length), so inner product equals cosine similarity.

Only the small head is optimized (backbone frozen) with cross-entropy and Adam; the best checkpoint by validation loss is saved to `best_ft.pth`. This keeps training fast and stable on our dataset.

We compute embeddings for all *gallery* images and store them in a FAISS `IndexFlatIP` (exact inner-product search). Then, for each *query*, we compute its embedding, retrieve the top-k (which was 10) nearest gallery vectors, and write the ranked filenames to `submission.json`.

We organized the whole pipeline through a single runner, `main.py`, which exposes four essential stages: (1) Split: create identity-stratified train/val lists (JSON maps from filename to label); (2) Train: fine-tune a linear classification head on top of an ImageNet-pretrained, frozen backbone (this is handled by `train_ft.py`), while `data.py` provides preprocessing (resize/normalize) and loaders, and `model.py` builds the chosen backbone with the new head; (3) Index: discard the head and turn the model into an embedding extractor (`model.extract_embedding_model`), compute ℓ_2-normalized 512-D embeddings for all *gallery* images, and build a FAISS `IndexFlatIP` (cosine similarity), implemented in `retrieve.py`; (4) Retrieve: embed *query* images, search the FAISS index, and write a `submission.json` with the ranked top-k gallery matches.

We compared ResNet-50, EfficientNet-B0, and ViT-B/16 with a quick retrieval check: a 20/80 query–gallery split inside validation, embedding extraction, cosine similarity, and Top-k accuracy plus average inference time. `EfficientNet-B0` consistently offered the best accuracy–per–time trade–off and stable optimization, so we fixed this backbone for the final pipeline.The internal benchmark showed that EfficientNet-B0 had the best accuracy–time trade-off, so it was used as the backbone for training, embedding extraction, FAISS indexing, and final retrieval.

1.3 Summary of Results

Overall, our system reached a final leaderboard score of 37/1000 (score = 600·Recall@1 +300·Recall@5 +100·Recall@10). The full source code is available at: github.com/camillabonomo02/MLOps_project.

2 Models Considered

2.1 Convolutional Neural Networks (CNNs)

A convolutional neural network [7, 1] is a kind of Artificial Neural Network (ANN) designed for pattern recognition within images. Like other ANNs it has layers of neurons that take inputs, apply weights, a non-linear function and pass results forward; training adjusts weights to reduce a loss. The main problem with ANNs is that the images are big. A modest 64x64 color image (3 channels) already gives 12,288 inputs to a single neuron and real models would need many such neurons and layers. This could lead to many parameters, slow training, high computational cost but most important risk of overfitting. To address these issues, CNNs add three key ideas. First, local connectivity: each neuron looks only at a small patch of the image (its receptive field) instead of all pixels. Second, weight sharing: the same small set of weights (a filter) is reused as it slides over the image, producing a feature map that fires whenever a pattern (e.g., an edge or corner) appears anywhere. Third, downsampling (pooling) reduces the spatial size of feature maps, which cuts computation and makes the representation more robust to small shifts and distortions.

The simplest overall architecture is composed by several layers.

(1) *Convolutional Layer*. CNNs use the convolutional operation

$$S(i,j) = (I*K)(i,j) = \sum_m \sum_n I(i-m, j-n)\, K(m,n).$$

within the convolutional layers. A convolutional layer applies several learnable kernels/filters over the input volume (H×W×C) to produce several activation maps. Each activation is the dot-product between the kernel and the local patch, then passed through a nonlinearity. Finally, this type of layers are controlled by a few hyperparameters: the kernel size (receptive field), the stride (step), the padding (zeros at the border) and the number of filters.

(2) *Detector (nonlinearity) Layer*. After the convolution gives a linear response, the detector layer applies a nonlinearity to turn that response into a clear "presence signal" of a pattern. Rectified Linear Unit (ReLU) is the default nonlinearity in modern CNNs.

$$\mathrm{ReLU}(x) = \max(0,x), \qquad \mathrm{ReLU}'(x) = \begin{cases} 1 & x > 0, \\ 0 & x \le 0\,. \end{cases}$$

It keeps positive evidence and sets negative responses to zero, turning a filter's linear output into a clear detection signal of a pattern. ReLU helps deep models train faster because it does not saturate on the positive side (gradients stay strong), and it naturally creates sparse activations (many exact zeros), which often improves generalization and efficiency.

(3) *Pooling Layer*. After convolution and the detector (nonlinearity), a CNN often applies a pooling function to further process the feature maps. Pooling replaces the output at a location with a summary of nearby outputs. The ones that are widely use are max pooling and average pooling among others.

$$y(i,j) = \max_{(u,v)\in W(i,j)} x(u,v).$$

$$y(i,j) = \frac{1}{|W(i,j)|} \sum_{(u,v)\in W(i,j)} x(u,v).$$

The main effect is robustness to small translations of the input. If the pattern moves a few pixels, the pooled value barely changes. This matches tasks where we care more about whether a feature is present than about its exact pixel location. Pooling also improves efficiency. If we pool over non-overlapping regions spaced k pixels apart, the next layer receives roughly k times fewer inputs, reducing computation and memory. In practice, pooling is useful to standardize the size of the representation by outputting a small, fixed set of summary statistics per region.

(4) *Fully-Connected (FC) head*. [8] After several blocks of convolution + nonlinearity + pooling, the network has a set of high-level feature maps of size H×W×C. An FC layer flattens them into a vector and applies an affine transform. Because every neuron connects to all previous activations, the FC head can mix information across channels and spatial locations and then produce class scores via softmax.

$$p_k = \frac{e^{y_k}}{\sum_{j=1}^{K} e^{y_j}}, \qquad k = 1, \ldots, K. \tag{1}$$

For retrieval variants, the head outputs a fixed-dimensional embedding that we L2-normalize and compare with cosine similarity when doing classification we train with cross-entropy

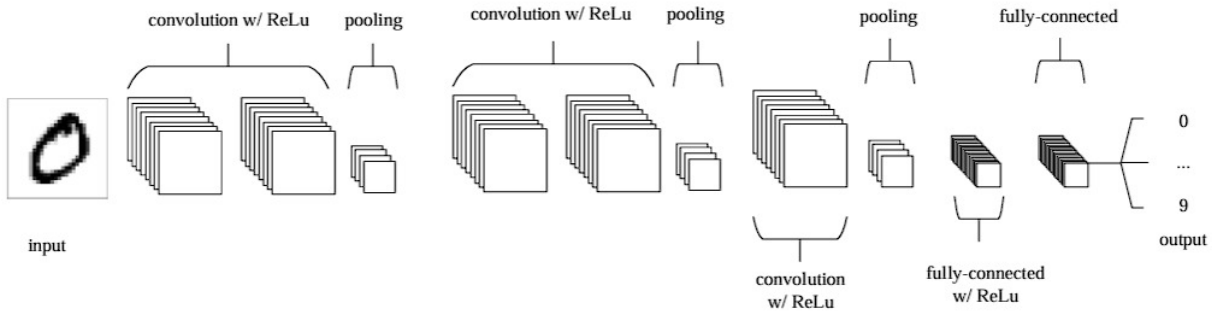
$$\mathcal{L}(\Theta) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{i,k}\, \log\left(\hat{y}_{i,k}\right)$$

where N denotes the number of training examples and K the number of classes. The term $y_{i,k}$ represents the true label of example i for class k, usually encoded as a one-hot vector, while $\hat{y}_{i,k}$ corresponds to the predicted probability that example i belongs to class k, typically obtained through a softmax function. Finally, Θ indicates the set of model parameters.

In practice, a CNN is built by repeating Convolution → Nonlinearity (e.g., ReLU) → Pooling, and ending with one or more Fully-Connected (FC) layers that map the final features to class scores for classification and regression purposes. In figure an overall general CNN architecture.

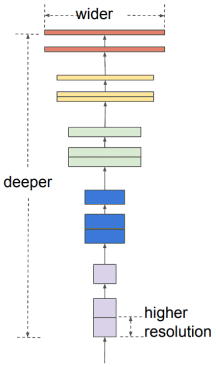
2.1.1 EfficientNet-B0

EfficientNet [2] was designed to deliver higher accuracy with less computation by starting from a compact baseline network, called B0, and then scaling that baseline in a principled way. The key observation is that convolutional neural networks get better when you increase three things: how many layers they have (depth), how many channels each layer carries (width), and how large the input image



Overall general CNN's architecture (adapted from O'Shea and Nash, 2015).

is (resolution), but doing so along only one axis tends to hit diminishing returns. EfficientNet proposes to balance all three together using a single scale parameter, so the network grows "evenly" in depth, width, and resolution as your compute budget increases. This idea is referred to as "compound scaling".



Compound scaling in EfficientNet (adapted from Tan & Le, 2019)

The B0 architecture itself follows a straightforward backbone layout: an initial stem, a sequence of stages, and a final head similar to a classical CNN. Inside the stages, EfficientNet-B0 uses MBConv blocks, short for "mobile inverted bottleneck". These blocks briefly expand the channel dimension, apply a depthwise convolution (which is an economical variant that processes each channel separately) and then project back down to a smaller channel count. That expand-process-project pattern preserves expressive power while keeping computation low. On top of MBConv, the model includes a Squeeze-and-Excitation (SE) step that learns a per-channel reweighting: the network measures how informative each channel is (via a global average) and then scales channels up or down accordingly, so compute is focused on the most useful signals. The activation function used is a smooth nonlinearity (often called Swish or SiLU), which helps optimization without changing the overall design.

2.1.2 ResNet-50

Deep plain networks (many stacked layers without shortcuts) can suffer from a "degradation" phenomenon: training error worsens as depth increases. ResNet changes what each block learns. Instead of fitting a full mapping $H(x)$ from an input x, a residual block is asked to learn only the residual

$$F(x;\Theta) \;=\; H(x) - x, \tag{2}$$

and outputs

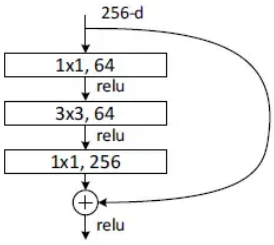
$$y \;=\; F(x;\Theta) + x, \tag{3}$$

where the addition is a shortcut (skip) that carries x forward unchanged. If the identity is close to optimal, the network can simply drive $F(x;\Theta) \to 0$.

A residual block [4, 3] is thus a small stack of convolutions whose output $F(x)$ is element-wise added to the input x. When input and output shapes match, the shortcut is a pure identity; when shapes differ (e.g., downsampling or channel change), a 1×1 *projection* W_s aligns dimensions:

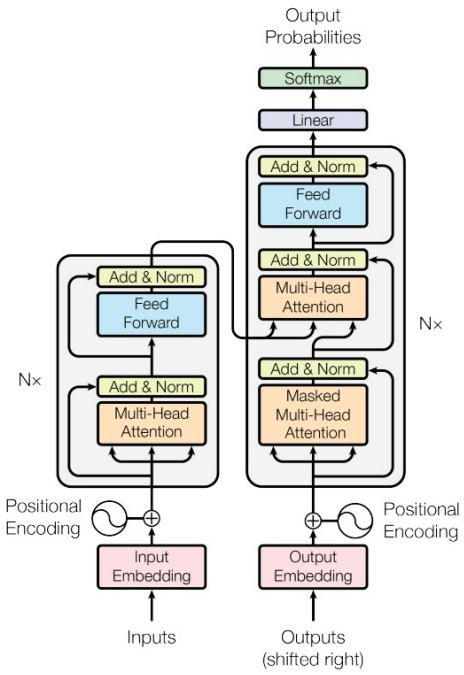
$$y \;=\; F(x;\Theta) + W_s x.$$

For deeper, computationally practical models, the authors introduce a bottleneck block with three convolutions arranged 1×1 → 3×3 → 1×1. The first 1×1 layer reduces channels (compress), the 3×3 does most of the spatial processing, and the final 1×1 restores channels. This expand–process–restore pattern keeps compute modest while enabling very deep nets. ResNet-50 [4] is obtained by replacing each 2-layer residual block in a 34-layer design with this 3-layer bottleneck; stage counts then yield 50 layers in total.



ResNet-50 bottleneck residual block (adapted from He et al., 2016)

As in classic CNNs, ResNet processes an input image through a stem (a 7×7 convolution and pooling that produce initial feature maps), followed by four stages (often denoted conv2-x to conv5-x). Each stage stacks many residual/bottleneck blocks; when moving to the next stage, spatial resolution halves and channel count increases, so later



Overall Transformer Architecture (adapted from Vaswani et al., 2017)

stages represent larger, more abstract patterns. A global average pooling summarizes each channel to a single number, and a final fully-connected layer produces class probabilities. The shortcut sits alongside every block, either as an identity or a 1×1 projection when shapes change.

2.2 Transformers

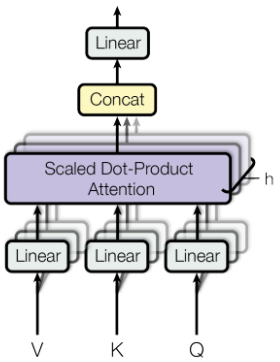
A Transformer [6] is a neural network architecture for sequences that drops recurrence and convolutions entirely and builds everything on top of attention which is a mechanism that lets the model look at all parts of the input at once and decide what matters for the current computation. This shift was motivated by two practical pains with recurrent models like RNNs and LSTMs: they process tokens step-by-step (hard to parallelize) and they struggle to keep very long-range information alive. Transformers process all positions in parallel, yet still capture long-distance dependencies via attention weights computed across the whole sequence

The original Transformer follows an encoder–decoder layout. The encoder reads the whole input and produces a sequence of hidden vectors; the decoder generates the output one token at a time, but at each step it can "consult" the encoder's representations. Each encoder layer has two sub-parts: (1) multi-head self-attention, and (2) a position-wise feed-forward network. Each sub-part is wrapped with a residual connection and layer normalization. Decoder layers mirror this design, with an additional encoder–decoder attention block; the decoder's self-attention is also masked to prevent access to future tokens during generation.

To compute attention, each position is mapped to three vectors: a *Query* (Q), a *Key* (K), and a *Value* (V), obtained by learned linear projections of the input. Attention scores come from the similarity between queries and keys; after a softmax, these scores weight the values and produce a new, context-aware representation for each position. In the Transformer, this is the scaled dot-product attention:

$$\text{Attention}(Q, K, V) \;=\; \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V, \qquad (4)$$

where d_k is the dimensionality of the keys. The scaling factor $1/\sqrt{d_k}$ stabilizes gradients when d_k is large.

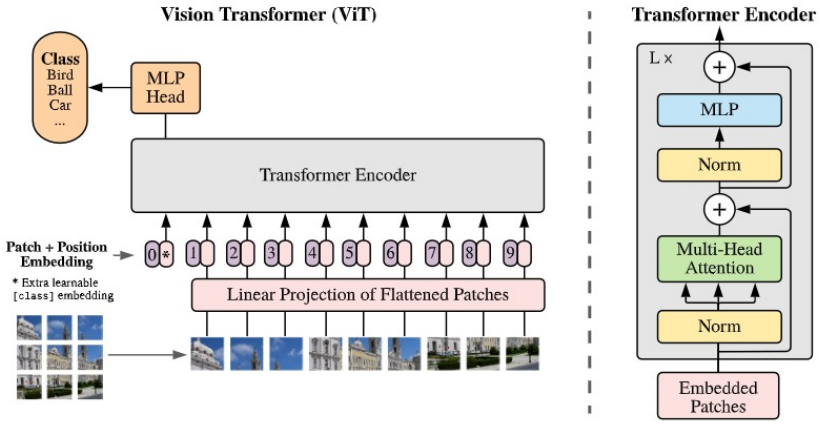


Scaled dot-product attention (adapted from Vaswani et al., 2017)

Instead of a single attention operation, the Transformer runs several in parallel (multiple heads) each with its own learned projections. Different heads can specialize in different relations (e.g., local syntax vs. long-range dependencies). Their outputs are concatenated and projected back to the model dimension, which improves expressivity without increase the computational cost.

Since self-attention has no inherent notion of order, Transformers add a positional encoding to each input embedding so the model knows which token is first, second, and so on.

After self-attention, each position passes through the same small feed-forward network (two linear layers with a nonlinearity, applied independently to each position). At-



ViT Architecture (adapted from Dosovitskiy et al., 2021)

tention mixes information across positions, while the feed-forward block transforms each mixed representation locally. Residual connections and normalization wrap both sub-layers to help gradients flow and stabilize training.

Although originally designed for language, nothing in the self-attention mechanism requires words. If an image is divided into small patches and each patch is treated as a "token," the same encoder stack can process images. This idea is at the heart of Vision Transformers (ViT), where 16×16 pixel patches become the tokens fed into a Transformer encoder.

2.2.1 ViT-B/16

A Vision Transformer [5] applies the Transformer encoder directly to images. The core idea is simple: instead of scanning an image with convolutional filters, we first turn the image into a short sequence of visual tokens and then let self–attention decide which tokens should talk to which. This gives the model a global view from the very first layer and keeps the computation highly parallel.

Given an input image of size $H \times W \times C$, we split it into non-overlapping patches of size $P \times P$. Each patch is flattened and passed through a learned linear projection to obtain a D-dimensional *patch embedding*. We then attach a learnable `[CLS]` token whose final representation will summarize the whole image for classification, and we add a positional embedding to every token (patches + `[CLS]`) so that the model is aware of where each patch comes from.

The sequence of tokens is processed by L identical encoder layers. Each layer contains a Multi-Head Self-Attention block that mixes information across all tokens, followed by a small position-wise MLP that refines each token locally. Both sublayers are wrapped with residual connections and normalization (Add & Norm) to stabilize training. After the last layer, the `[CLS]` vector is fed to a lightweight head (often a single linear layer or a tiny MLP) to produce class scores. When the goal is retrieval rather than classification, the `[CLS]` vector (or a pooled combination of patch tokens) serves as the image embedding (typically ℓ_2-normalized and compared with cosine similarity).

ViT-B/16 uses patches of 16×16 pixels. With the standard 224×224 input, this yields $14 \times 14 = 196$ patch tokens. Adding the `[CLS]` token gives a sequence length of 197 tokens per image. The "B" (Base) configuration has $L = 12$ layers, hidden size $D = 768$, MLP hidden size 3072, and 12 attention heads, for roughly 86M parameters.

Because ViT does not hard-code locality and translation equivariance the way CNNs do, it relies more on data and regularization to learn robust features. Pretraining on larger image collections (or careful augmentation/weight decay) often helps when the target dataset is small or exhibits strong domain shifts.

3 Evaluation

3.1 Quantitative Evaluation

We evaluate our retrieval models on the Labeled Faces in the Wild (LFW) dataset [10] post-competition. LFW is a long-standing benchmark for unconstrained face recognition and is commonly treated as a de facto standard for face verification and recognition in the wild. In our setting, we used a curated LFW subset (3,023 images across 62 celebrity identities); we adopt a 50/20/30 split for train/validation/test and compute the same competition-style score used during the challenge (a weighted sum of Recall@1, Recall@5, and Recall@10). In this metric, a query is considered correct if at least one relevant item appears among the first k retrieved results. This definition aligns with Recall@k in information retrieval literature. Formally [9]:

$$\text{Recall@}k = \frac{1}{Q}\sum_{q=1}^{Q} \mathbf{1}\Big\{\exists\, i \le k : y_q(i) = 1\Big\},$$

where $y_q(i) = 1$ if the i-th retrieved item for query q is relevant, and $\mathbf{1}\{\cdot\}$ is the indicator function.

As a complementary measure we report the mean Average Precision (mAP) [9]. For a set of Q queries,

$$\text{mAP} = \frac{1}{Q}\sum_{q=1}^{Q} \text{AP}(q),$$

where the Average Precision for a single query q is

$$\mathrm{AP}(q) = \frac{1}{N_q} \sum_{k=1}^{N} P_q(k)\, \mathbf{1}\{y_q(k) = 1\}.$$

Here N_q is the number of relevant items for query q, N is the length of the ranked list, and $P_q(k)$ denotes the precision computed up to rank k. The mAP summarizes the ranking quality across all queries by averaging precision at the positions where relevant items occur.

All models load ImageNet-pretrained weights with frozen backbones; only the classification head is trained using Adam and cross-entropy. Inputs are resized to 224×224, normalized with ImageNet statistics, and augmented with light randomness during training; embeddings are ℓ_2-normalized for cosine retrieval. This mirrors the pipeline employed throughout the project. Experiments were conducted on a single NVIDIA GPU with PyTorch 2.0. We extract features from the penultimate layer and apply L2 normalization for retrieval. EfficientNet-B0 serves as our baseline model throughout the evaluation as it "wins" our internal benchmark in the actual competition.

Table 1: Architecture comparison under controlled conditions.

Architecture	Top-1	Top-5	Top-10	mAP	Score†	Time (s)
EfficientNet-B0*	0.320	0.586	**0.707**	0.370	438.7	**118**
ViT-B/16	**0.331**	0.580	0.702	**0.372**	**443.1**	738
ResNet50	0.171	0.431	0.608	0.253	292.8	293
Relative Performance						
EfficientNet-B0	–	–	–	–	98.9%	6.3× faster
ViT-B/16	–	–	–	–	100%	1.0×
ResNet50	–	–	–	–	66.0%	2.5× faster

†Competition Score = 600 × Top-1 + 300 × Top-5 + 100 × Top-10

*Baseline model used throughout experiments

Training conditions: LR=0.001, 3 epochs, BS=32, with augmentation, ImageNet normalization

Under matched hyperparameters (LR $= 10^{-3}$, 3 epochs, BS $= 32$), ViT-B/16 attains the highest competition score, while EfficientNet-B0 reaches essentially the same accuracy band at a fraction of the training time; ResNet-50 lags behind by a sizeable margin. In practice, EfficientNet achieves about 99% of ViT-B/16's score while being an order of magnitude faster to train (see Table 1).

3.1.1 Training dynamics

To optimize the classification head we employed the cross-entropy loss in combination with the Adam optimizer. Cross-entropy is the standard objective for multi-class classification, as it penalizes the model whenever the predicted probability distribution diverges from the true one-hot label. Adam complements this loss by adapting the learning rate of each parameter through estimates of first and second moments of the gradients, which makes training more stable.

To visualize learning behaviour we report the cross-entropy trend for EfficientNet-B0 (see Figure 2). Training loss decreases steadily across epochs, while validation loss follows with a milder slope, suggesting rapid early adaptation followed by a slower regime where overfitting can emerge. This pattern substantiates the early-stopping observation made during development, namely that performance peaked very early and then gradually degraded with additional epochs (see Figure 3).

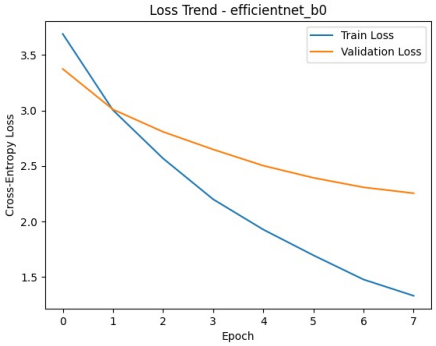


Figure 2: Cross-entropy loss on LFW for EfficientNet-B0 (train vs. validation).

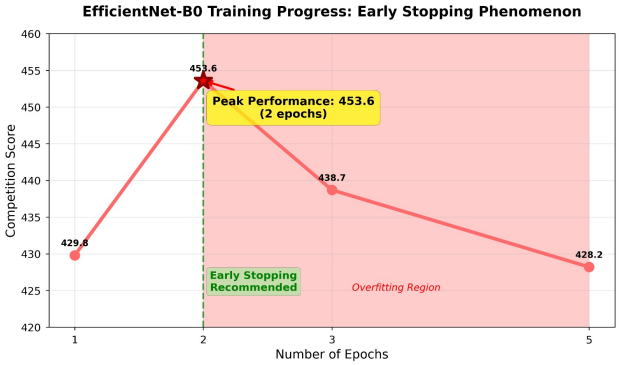


Figure 3: Early stopping behaviour for EfficientNet-B0 on LFW: validation performance peaks within the first few epochs and declines afterwards

3.1.2 Ablation study

To better understand the sensitivity of our models to design choices, we performed a series of ablation experiments. The first dimension we explored was the learning rate. As shown in Figure 4, a value of 10^{-3} consistently yielded the most stable and competitive performance across backbones. EfficientNet-B0 and ViT-B/16 benefited from this setting, reaching competition scores above 430, while higher learning rates (10^{-2} or 10^{-1}) degraded the results substantially, especially for ResNet-50, whose score dropped below 300. This pattern confirms the strong dependence of convergence on careful learning rate selection.

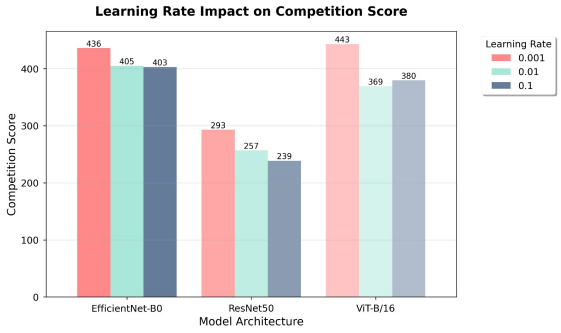


Figure 4: Impact of learning rate on competition score across backbones

We then analyzed the impact of data preprocessing and training setup. Figure 5 illustrates how removing ImageNet normalization led to the largest decrease in score, with a drop of more than 30 points compared to the baseline. Similarly, reducing the batch size to 8 or enlarging it to 64 harmed performance, indicating that a moderate value of 32 provided the best balance between stability and generalization. The absence of data augmentation

also produced a measurable decline, although less dramatic than the lack of normalization. Taken together, these experiments underline that both preprocessing and mini-batch design influence retrieval accuracy.

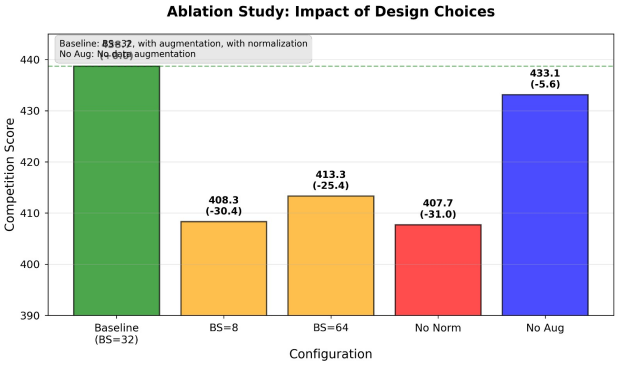


Figure 5: Ablation results showing how batch size, normalization, and data augmentation affect competition score

Finally, we summarize the top-performing configurations in Table 2. ViT-B/16 with a learning rate of 10^{-3} and five training epochs achieved the highest overall score of 450.8, outperforming all other settings. However, EfficientNet-B0 proved remarkably competitive, achieving 438.7 with the same learning rate in only three epochs, while requiring just two minutes of training time compared to nearly twenty for ViT-B/16.

Table 2: Top-10 model configurations on the LFW dataset. The best values for each metric are highlighted in bold.

Model	LR	Epochs	Score	Top-1	Top-5	Top-10	mAP	Time (min)
ViT-B/16	0.001	5	**450.8**	0.320	**0.630**	0.696	**0.399**	19.7
ViT-B/16	0.001	3	443.1	**0.331**	0.580	**0.702**	0.372	12.3
EfficientNet-B0	0.001	3	438.7	0.320	0.586	0.707	0.370	**2.0**
EfficientNet-B0	0.100	5	437.6	0.304	0.619	0.696	0.374	3.1
ViT-B/16	0.010	5	432.0	0.320	0.575	0.674	0.370	19.4
EfficientNet-B0	0.001	5	428.2	0.309	0.580	0.685	0.381	3.1
EfficientNet-B0	0.010	3	405.0	0.293	0.547	0.652	0.353	2.0
EfficientNet-B0	0.100	3	402.8	0.276	0.569	0.663	0.342	1.9
ViT-B/16	0.100	5	396.1	0.276	0.547	0.663	0.344	19.6
EfficientNet-B0	0.010	5	395.0	0.260	0.575	0.669	0.326	3.1

3.2 Qualitative Evaluation

We only had access to the `submission.json` (lists of Top-K results per query) and did not retain model checkpoints or image embeddings. This prevents any feature–space analysis (e.g., t-SNE/UMAP) and any recomputation of similarities; therefore we relied on a visual inspection of query panels.

Concretely, we sampled representative queries and displayed the Top-10 retrieved images side by side. Without ground truth labels we cannot declare hits/misses (see Figure 6).

Figure 6: Qualitative example from the competition. Query (left) and top-10 retrieved gallery images (right) based on cosine similarity

In this case, by visual inspection, none appears to match the query identity. This aligns with our low competition score and reflects model limitations under challenging face recognition conditions. Although some returned images resemble the query in pose, background, or illumination, they lack true identity consistency.

4 Discussion & Limitations

During development we used a fixed train/validation split built from the same set of identities. Even if the images were different, many belonged to the same people across train and validation. This made validation accuracy look good, but it did not match the competition, where the test set contained new identities and a mix of real and synthetic faces (a strong domain shift).

We also froze the ImageNet backbone and trained only a small classification head with cross-entropy. This setup teaches the model to choose among the known classes, not to build an embedding space where "same person = close, different person = far". At inference we then removed the classifier and used features from the penultimate layer (L2-normalized, cosine similarity with FAISS) for retrieval. This creates a mismatch: we trained as a closed-set classifier, but we evaluated as an open-set retrieval system.

Data was limited (few images per identity) and our augmentations were light (mainly resize and horizontal flip). With the backbone frozen, the network could not adapt to faces with varied pose, lighting, makeup, or synthetic artifacts. As a result, the model often relied on basic cues (backgrounds, style, color) rather than identity-specific details.

Our internal benchmark also did not reflect the real task. We selected models using accuracy on the validation split of seen identities (sometimes even querying and searching within the same pool), which mostly measures classification on familiar classes, not retrieval on unseen people.

Finally, we used a learning rate that was too high for Adam with only a few epochs (e.g., 0.1 for 3 epochs), leading to unstable updates and limited learning in this low-data setting.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. Available at: http://www.deeplearningbook.org.

[2] M. Tan and Q. V. Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *International Conference on Machine Learning (ICML)*, 2019. https://arxiv.org/abs/1905.11946.

[3] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. ResNet Model. In: *Dive into Deep Learning (D2L)*, 2020. https://d2l.ai/chapter_convolutional-modern/resnet.html.

[4] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778. https://arxiv.org/abs/1512.03385.

[5] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations (ICLR)*, 2021. https://arxiv.org/abs/2010.11929.

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I. Polosukhin. Attention is All You Need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. https://arxiv.org/abs/1706.03762.

[7] K. O'Shea and R. Nash. An Introduction to Convolutional Neural Networks. *arXiv preprint*, 2015. https://arxiv.org/abs/1511.08458.

[8] A. Babenko, A. Vedaldi, A. Zisserman, and V. Lempitsky. Neural Codes for Image Retrieval. In *European Conference on Computer Vision (ECCV)*, 2014, pp. 584–599. Springer. https://arxiv.org/abs/1404.1777.

[9] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008. (Reference for Recall@k and mAP formulas.)

[10] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller. Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments. Technical Report 07-49, University of Massachusetts, Amherst, 2007. http://vis-www.cs.umass.edu/lfw/.

Workload distribution across tasks and team members

Task	**Camilla**	**Paolo**	**Davide**
Data Preprocessing		✓	
Model Design & Implementation	✓	✓	✓
Training and Fine-Tuning	✓	✓	✓
Model Benchmarking			✓
Report Writing			
Introduction			✓
Models Section		✓	✓
Quantitative Eval.			✓
Qualitative Eval.		✓	
Final Discussion		✓	