Deep Learning Coursework 1: Question 4

This report summarises the experiments we have run in question 2, in order to select and train our final model. It is complementary to the accompanying notebook. All material, including this report, are temporarily available on the author's git repository. This report follows closely the structure of the notebook.

Model Architectures

We explored four model architectures throughout this project. Two were original designs, one was an adaptation of the well-known `EfficientNet`, and the final one employed transfer learning using a pre-trained instance of `EfficientNet`. All models projected inputs into a 128-dimensional embedding space, consistent with the choice made by Hermans et al. [1]. Although this dimensionality is on the lower end—Sun et al. [4], for instance, used a 512-dimensional embedding—we chose it for its lower training cost and faster convergence. Given more time and computational resources, we would have liked to investigate higher embedding dimensions, potentially combined with L_1 and L_2 regularisation in the final dense layer.

Across all architectures, we used a consistent design for the network head. Specifically, the feature map was first flattened—either via a global average pooling layer or a standard flatten operation—then passed through a fully connected layer without activation, followed by an L_2 normalisation layer to ensure the output embedding had unit norm. The dense layer incorporated L_2 regularisation. In our experiments, the flattening method had a significant impact on performance. Architectures using global average pooling performed poorly and were subsequently discarded early in the development process.

All models used the Swish activation function, which introduced a smooth non-linearity that we found preferable to ReLU, despite the additional computational overhead.

`EfficientNet` Our first approach adapted the `EfficientNetB0` architecture [5]. We selected this model for its compact size (fewer than 5 million parameters) and strong baseline performance in classification tasks. We replaced the original classification head with the architecture described above. However, this model yielded poor results. We hypothesised that this was due to the original design of `EfficientNet`, which performs multiple rounds of expansion and compression to extract fine-grained features suitable for classification. Since our task involved learning embeddings that cluster similar instances together, this expansion-contraction pattern may have been suboptimal. Another possibility was simply insufficient training time. Further details on this architecture are provided in Appendix A.

`SimpleEmbeddingNet` and `SimpleEmbeddingNetV2` The second and third architectures were purpose-built for the task. These networks were composed of stacked convolutional blocks designed to progressively reduce spatial resolution while increasing the number of feature channels. In `SimpleEmbeddingNet`, each block included a 3x3 convolution (stride 1, same padding, no bias), batch normalisation, Swish activation, and a 2x2 pooling layer. We experimented with both max pooling and average pooling, finding that max pooling consistently outperformed its counterpart.

In contrast, `SimpleEmbeddingNetV2` omitted the pooling layer and instead used stride-2 convolutions to downsample the spatial dimensions. This modification achieved the same effect but in a more integrated manner. Our motivation for these designs was rooted in the objective of dimensionality reduction: we sought models that naturally decreased spatial dimensions while increasing channel capacity, thereby allowing the network to learn diverse feature maps. As a small variation, the first two blocks in `SimpleEmbeddingNet` used 5x5 kernels—an idea inspired by `EfficientNet`—to perform more aggressive initial downsampling.

Both models were tested with five and six stacked blocks, followed by the consistent head design. Among all models evaluated, the `SimpleEmbeddingNet` with 5 max pooling blocks and a flatten top performed best. This model had 6,455,840 trainable parameters and was trained for approximately two hours before early stopping was triggered. We evaluated its performance primarily using the third question in our evaluation setup, while monitoring training loss and recall@1.

`EfficientNetPretrained` Our final model used the pre-trained `EfficientNetB0` from Keras with ImageNet weights. We retained the original body of the network and replaced the top as described previously. The pre-trained weights were frozen during training. This model, like the non-pretrained `EfficientNet`, did not perform well, but provided valuable experience in adapting and integrating transfer learning models into our training pipeline.

Data Preprocessing, Batching, and Triplet Mining

Prior to triplet mining, we designed our data loader to ensure that each training batch consisted of exactly P unique identities, each with N associated images. This structure guaranteed a consistent and predictable number of valid triplets per batch. We experimented with values of P between 32 and 50 and N between 4 and 10. Empirically, smaller batch sizes resulted in faster convergence. Consequently, we settled on $P = 32$ and $N = 8$, yielding an effective batch size of 256 images.

Ideally, we would have increased the batch size gradually as training progressed, thereby introducing harder and more diverse triplets in later stages. Unfortunately, this dynamic batching strategy was incompatible with TensorFlow's `@tf.function` decorator, which constructs static computation graphs based on the input signature observed during the first call. Changing the batch shape triggered graph retracing, leading to errors or performance degradation. While TensorFlow does offer some flexibility via dynamic input signatures, this did not extend to functions with variable return types, such as those used in triplet mining.

We implemented four triplet mining strategies: `batch_all`, which returns all valid triplets in the batch [1]; `batch_hard`, which iterates through the batch to find the hardest positive and hardest negative for each anchor [1]; `hard_negative`, which loops over all anchor-positive pairs and selects the hardest negative [3]; and `semi_hard`, which returns only those triplets satisfying $s_p < s_n < s_p + c$, where $s_p = s(\mathbf{e}_a, \mathbf{e}_p)$ and $s_n = s(\mathbf{e}_a, \mathbf{e}_n)$ are the similarities between anchor-positive and anchor-negative embeddings, respectively, and c is a tunable margin [3]. Detailed definitions and discussion of these mining strategies are provided in Appendix B.

Although Circle Loss was originally designed to avoid the need for explicit mining [4], due to its weighting of sample contributions by difficulty, we observed that convergence with `batch_all` alone was extremely slow. Consequently, we trained most of our models using `batch_hard` and `hard_negative` strategies. The `hard_negative` method led to faster convergence in early epochs but tended to plateau, whereas `batch_hard` continued to improve gradually and produced better final embeddings. In hindsight, a promising strategy would have been to adopt `semi_hard` mining while gradually increasing the margin c throughout training to raise the difficulty of selected triplets over time.

Finally, in line with common practice, we applied random data augmentations during training. These transformations were re-sampled each epoch to enrich the training distribution and mitigate overfitting.

Circle Loss and Training Dynamics

In our initial experiments with Circle Loss [4], we adopted fixed hyperparameters γ and m as suggested in the original paper, specifically $\gamma = 256$ and $m = 0.25$. However, we observed that a dynamic schedule for these parameters led to improved performance. We began training with $\gamma = 164$ and $m = 0.22$, and gradually increased them over the course of training. This schedule was designed to reach the values used in the original paper within the first 20 epochs, with our training typically spanning up to 40 epochs.

The rationale for this scheduling was twofold. The parameter γ controlled the weighting of each sample based on its difficulty: higher values of γ placed greater emphasis on hard positives (low similarity) and hard negatives (high similarity), effectively steepening the gradients and refining the optimization process. Gradually increasing γ allowed the model to focus more on difficult examples as training progressed. Similarly, m defined the margin between positive and negative pairs. Starting with a small margin enabled the model to learn a coarse separation between embeddings, while increasing m over time encouraged the formation of tighter, more distinct clusters. This progressive adjustment contributed to a more stable and interpretable training trajectory.

Learning Rate Scheduling

We employed an exponentially decaying learning rate schedule to reflect the diminishing need for large parameter updates as training progressed. This approach was inspired by the schedule used in [1], though we significantly modified the decay onset, rate, and final value based on empirical analysis. We examined loss curves from early experiments that did not use any learning rate scheduling and identified stages where the loss either plateaued or became noisy. These observations guided our decisions regarding when to begin decaying the learning rate and how aggressively to reduce it.

Monitoring Metrics

During training, we monitored both the average loss per batch and the recall@1 metric. Due to the dynamic nature of γ and m, the loss function was not a consistent indicator of model performance over time. Similarly, recall@1 on the training set was subject to variability due to changes in the sampled identities and images each epoch. To address this, we used a fixed validation set consisting of the final 32 identities in the dataset and the first four images per identity. At the end of each epoch, we computed the validation loss and recall@1 using this held-out set.

Early stopping was implemented based on the recall@1 score on the validation set, along with a maximum training time constraint. Further details on monitoring strategies and model checkpointing procedures are provided in Appendix C.

Implementation Challenges, Lessons Learned, and Future Directions

One of the main challenges we encountered was the limited flexibility of `TensorFlow` for highly customised workflows. In particular, the `@tf.function` decorator made it difficult to implement dynamic training behaviours such as varying batch sizes or switching between different triplet mining strategies during training. These limitations stemmed from TensorFlow's static graph tracing, which retraced the graph whenever input shapes or internal logic changed, often resulting in inefficiencies or execution failures.

We also recognised the limitations of using notebooks for experimentation and reproducibility. To address this, we restructured the entire project into Python scripts, which are publicly available in the author's git repository. This allowed for better modularity, clearer version control, and easier integration with automated training workflows.

Another practical challenge was managing idle time during model training. We initially used Google Colab but quickly transitioned away from it due to its dependence on notebook interfaces and the instability of storing intermediate outputs. Uploading datasets repeatedly and the risk of losing code or model checkpoints due to interrupted sessions proved too great for our workflow.

Appendix A: `EfficientNet`

`EfficientNet` consists of a sequence of inverted residual bottleneck blocks, each followed by a squeeze-and-excitation (SE) operation. The architecture was introduced in the paper "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks" [5], and builds on the mobile inverted bottleneck design from `MobileNetV2` [2]. These components collectively form the `MBConv` blocks that underpin `EfficientNet`.

We chose `EfficientNetB0` for its strong performance and compact size (approximately 4.5 million trainable parameters). The original model was designed for image classification and incorporates compound scaling—a principled method for scaling depth (d), width (w), and resolution (r) to maximise performance under a fixed parameter budget. Although our input images were lower resolution than those used in the original model, we retained the default depth and width configuration. Ideally, we would have adjusted these parameters to match our input resolution more closely.

Our main architectural modification was to replace the classification head with a new top that produces a d-dimensional, L_2-normalised embedding vector. We experimented with two top configurations—one using global average pooling and the other using a flatten layer. The flatten layer consistently yielded better performance in our setup.

Table 1: Our `EfficientNet` Architecture

Stage	Operator	Kernel	Stride	In Ch.	Out Ch.	In Res.	Out Res.	Layers
1	Conv2d	3x3	2	3	32	112×112	56×56	1
2	MBConv	3x3	1	32	16	56×56	56×56	1
3	MBConv	3x3	2	16	24	56×56	28×28	2
4	MBConv	5x5	2	24	40	28×28	14×14	2
5	MBConv	3x3	2	40	80	14×14	7×7	3
6	MBConv	5x5	1	80	112	7×7	3×3	3
7	MBConv	5x5	2	112	192	3×3	1×1	4
8	MBConv	5x5	1	192	320	1×1	1×1	1
9	Conv2d	1x1	1	320	1280	1×1	1×1	1
10	Dropout	-	-	1280	1280	1×1	1×1	1
11	Flatten	-	-	1280	1280	1×1	Vector(1280)	1
12	FC Layer	-	-	1280	128	Vector(1280)	Vector(128)	1
13	L2 Norm	-	-	128	128	Vector(128)	Vector(128)	1

Each convolutional layer in the architecture is followed by batch normalisation and **swish** activation, consistent with the original `EfficientNet`. Although the architecture also supports ReLU6, we found swish to perform better and observed instability when using ReLU6 early in our experiments.

As described, the `MBConv` block is a mobile inverted bottleneck block enhanced with SE optimisation. Each block consists of:

1. A 1x1 expansion convolution
2. A depthwise separable convolution
3. A squeeze-and-excite operation:
    - Global average pooling
    - 1x1 convolution to reduce channel dimension
    - 1x1 convolution to restore channel dimension

- Element-wise scaling of the feature map

4. A 1x1 projection convolution
5. A skip connection where applicable

We refer to the composition of these elements as `InvResNet`, which forms the fundamental building block of our network. A detailed version of the architecture and additional implementation notes can be found in the accompanying notebook.

Appendix B: Triplet Selection

In this appendix, we provide details on the triplet mining strategies implemented to construct anchor–positive–negative triplets used for training. As highlighted in [3, 1], the selection of triplets is critical to the convergence and effectiveness of models trained with triplet loss. Conversely, the authors of Circle Loss [4] argue that triplet selection becomes less crucial when using their loss function, since it internally assigns higher weights to harder examples.

We implemented four mining strategies. Each strategy accepts a batch of embedded images and returns a set of index triplets (anchor, positive, negative).

Batch All

This strategy enumerates all valid triplets within a batch. For a batch with P identities and N images per identity, it produces $(P \times N)(N-1)(P \times N - N)$ possible triplets. However, many of these are trivial and contribute little to the loss, resulting in slow convergence. Although conceptually simple, this method was not effective in practice.

Batch Hard

`Batch Hard` selects the most difficult triplet for each anchor: the positive with the *lowest* similarity and the negative with the *highest* similarity. Formally:

$$\mathbf{e}_{\text{pos}}^{\text{hard}} = \arg\min_{\mathbf{e}_p} s(\mathbf{e}_a, \mathbf{e}_p), \quad \mathbf{e}_{\text{neg}}^{\text{hard}} = \arg\max_{\mathbf{e}_n} s(\mathbf{e}_a, \mathbf{e}_n),$$

where $s(\cdot, \cdot)$ is the similarity function (in our case, scaled cosine similarity). While this method is effective in identifying informative triplets, it can be noisy in early training stages.

Semi-Hard

Proposed in [3], this strategy selects negatives that are harder than the positive but not too hard. A triplet (a, p, n) is selected if:

$$s(\mathbf{e}_a, \mathbf{e}_p) < s(\mathbf{e}_a, \mathbf{e}_n) < s(\mathbf{e}_a, \mathbf{e}_p) + c,$$

where c is a margin hyperparameter. This approach avoids both trivial and overly challenging triplets, balancing stability and learning signal. Although not used in our final experiments, we believe this strategy holds promise if paired with a dynamic margin schedule.

Hard Negative

This method identifies the hardest negative for each anchor–positive pair, selecting the negative with the highest similarity to the anchor. It offers a middle ground between exhaustive enumeration and extreme hardness. We found this strategy yielded more stable training than `Batch Hard`, though at the cost of a performance plateau.

In practice, we relied primarily on `Batch Hard` and `Hard Negative`. The former led to better final performance but slower convergence and noisier metrics, while the latter trained faster but often plateaued.

Appendix C: Model Monitoring During Training

To track training progress, we monitored the training loss and recall@1, as well as validation loss and recall@1 at the end of each epoch.

Definition: Recall@k For an anchor embedding $\mathbf{x}_a$, recall@k measures how often a positive match appears in the top-k most similar embeddings:

$$\text{Recall@}k(\mathbf{x}_a) = \frac{\text{Number of correct identities in top } k}{\text{Total number of correct identities}}$$

Batch-level recall@k is computed by averaging this value over all anchors.

Because γ and m in the Circle Loss vary throughout training, training loss alone was an inconsistent proxy for performance. Likewise, recall@1 on training data was sensitive to sampling variations. For consistency, we used a fixed validation set of 32 identities, each with 4 images. While the identities were constant, we unintentionally reshuffled images each epoch until the final run.

We implemented logging for both batch-wise and epoch-wise metrics and stored them as CSV files. Early stopping was triggered if validation recall@1 failed to improve for 5–15 epochs, depending on the experiment.

Model weights were saved after each epoch, along with a separate checkpoint of the best model. All logs and checkpoints are available in the project's GitHub repository. Model loading utilities can be found in `trained_model_getters.py`.

References

[1] Alexander Hermans, Lucas Beyer, and B. Leibe. In defense of the triplet loss for person re-identification. *ArXiv*, abs/1703.07737, 2017.

[2] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.

[3] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, page 815–823. IEEE, June 2015.

[4] Yifan Sun, Changmao Cheng, Yuhan Zhang, Chi Zhang, Liang Zheng, Zhongdao Wang, and Yichen Wei. Circle loss: A unified perspective of pair similarity optimization. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6397–6406, 2020.

[5] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *ArXiv*, abs/1905.11946, 2019.