Visual Analytics Portfolio

Jacob Lillelund

20 May 2025

Contents

1 Visual Analytics Exam Project

This repository contains the exam project for the Visual Analytics course, implementing various computer vision and image analysis techniques.

1.1 Project Overview

The project is organized into multiple assignments, each focusing on different aspects of visual analytics:

- **Assignment 1**: Image search algorithm using color histograms and embedding search
- **Assignment 2**: Image classification with Logistic Regression and Neural Network
- **Assignment 3**: Transfer learning with CNNs for Lego brick classification
- **Assignment 4**: Face detection with MTCNN

1.2 Data Preparation

Assignments includes specific guides on how to obtain the necessary datasets where relevant. All data should be placed in the top-level `/data` directory (outside the `/src` directory) and have this overall structure:

```
data/
├── 17flowers/    # Flower dataset used in Assignment 1
├── lego/         # Lego brick dataset used in Assignment 3
└── newspapers/   # Newspaper dataset used in Assignment 4
```

1.3 Quickstart

Use the interactive assignment runner:

```
# Make the script executable (if needed)
chmod +x run.sh

# Run the interactive assignment runner
./run.sh
```

This script will: 1. Check if the environment is setup, and if not, prompts queries whether the user would like to run `setup.py` 2. Present an interactive menu to select and run any assignment (after environment setup completes) 3. Display and point to results after an assignment is run.

Note: The interactive assignment runner executes all assignments with their default configurations only. For customized runs with different parameters, please refer to the "Manual Setup and Running" section below or the assignment-specific READMEs.

1.3.1 Setup without using run.sh

If you prefer to set up and run assignments manually, follow these steps:

```
# Make the setup script executable (if needed)
chmod +x setup.sh

# Run the setup script
./setup.sh
```

After setup, you can run individual assignments:

```
# Activate the virtual environment (if not already active)
source .venv/bin/activate

# Navigate to src directory
cd src

# Run Assignment 1
uv run python -m assignment_1.main --method both
uv run python -m assignment_1.main --method histogram # Run only
 ↪  histogram-based search
uv run python -m assignment_1.main --method embedding # Run only
 ↪  embedding-based search

# Run Assignment 2
uv run python -m assignment_2.main # Run both models
uv run python -m assignment_2.main --model logistic_regression # Run
 ↪  only logistic regression model
```

```
uv run python -m assignment_2.main --model neural_network # Run only
 ↪  neural network model

# Run Assignment 3
uv run python -m assignment_3.main # Run both CNN and VGG16 models
uv run python -m assignment_3.main --cnn_only # Run only the direct
 ↪  CNN model
uv run python -m assignment_3.main --vgg16_only # Run only the VGG16
 ↪  transfer learning model

# Run Assignment 4
uv run python -m assignment_4.main # Run face detection
```

Note: See assignment-specific README's for more information on running individual assignments and configuration options.

1.4 Assignment Specific Documentation

Each assignment has its own README file with detailed information:

- Assignment 1 README
- Assignment 2 README
- Assignment 3 README
- Assignment 4 README

2 Assignment 1: Image Search with Histograms and Embeddings

2.1 Overview

This project implements two different image search algorithms to find visually similar images:

1. **Histogram-based Search**: Uses color histograms from OpenCV to compare image similarity

2. **Embedding-based Search**: Use embeddings obtained from a pre-trained VGG16 model to compare image similarity using cosine similarity.

2.2 Data

The project uses the 17 Category Flower Dataset from the Visual Geometry Group at the University of Oxford. This dataset contains over 1000 images of flowers spanning 17 different species. The full dataset can be accessed from the official website.

Note: The dataset will be automatically downloaded using the `download_data.py` module when running the assignment through the assignment runner. You don't need to manually download the dataset.

2.3 Quickstart

The simplest way to run the assignment is using the provided run.sh script:

```
./run.sh
```

Then select option 1 from the menu.

Note: This will prompt whether you want to download the data. Click "Y". to continue.

You can also run the code without relying on run.sh, but only if you already have downloaded the dataset and placed it in `data/17flowers`.

```
cd src
uv run -m assignment_1.main
```

The will:

- Check for the flower dataset and **download it automatically** if missing
- Load the flower dataset (default: `data/17flowers`)
- Compare images to the target image (default: `image_0001.jpg`)
- Find similar images (default: 5) based on the chosen comparison method(s)
- Save results to output path

2.4 Configuration

The configuration of the models and assignment is defined in `config.py` using a hierarchy of Pydantic BaseSettings classes.

You can modify the default values directly in the `config.py`.

2.4.1 Command Line Options

Additionally, CLI commands allow you to control which search(es) you execute directly, bypassing the `config.py`.

You can specify which method to use with the `--method` option:

```
# Run only histogram-based search
uv run -m assignment_1.main --method histogram

# Run only embedding-based search
uv run -m assignment_1.main --method embedding

# Run both methods (default)
uv run -m assignment_1.main --method both
```

2.5 Project Structure

```
assignment_1/
├── __init__.py                 # Package initialization
├── config.py                   # Configuration settings
├── main.py                     # Main orchestration script
├── services/                   # Core services
│   ├── __init__.py
│   ├── histogram_search_service.py  # Histogram-based search service
│   └── embedding_search_service.py  # Embedding-based search service
└── scripts/                    # Search scripts
    ├── __init__.py
    ├── histogram_search.py     # Histogram search orchestrator
    └── embedding_search.py     # Embedding search orchestrator
```

2.6 Implementation Details

2.6.1 Histogram-based Search

This approach uses color histograms for image similarity comparison:

1. **Color Histogram Extraction**: BGR color space histograms with 8 bins per channel
2. **Comparison**: Chi-Square distance metric (`cv2.HISTCMP_CHISQR`)
3. **Ranking**: Images ranked by histogram distance (lower = more similar)

2.6.2 Embedding-based Search

This approach uses deep learning embeddings for image similarity comparison:

1. **Feature Extraction**: Pre-trained VGG16 model without top layers to extract image embeddings
2. **Comparison**: Cosine similarity between embeddings
3. **Ranking**: Images ranked by similarity score (higher = more similar)

2.6.3 Key Components

2.6.3.1 `HistogramSearchService` Class This service class encapsulates the core functionality for histogram-based search:

- **Initialization**: Sets up the search parameters including image directory, histogram bins, color space, and comparison method
- **Histogram Extraction**: Processes all images in the dataset and stores their histograms
- **Similar Image Search**: Compares the target image histogram with all others and returns the most similar ones
- **Results Export**: Saves the search results to a CSV file

2.6.3.2 `EmbeddingSearchService` Class This service class encapsulates the core functionality for embedding-based search:

- **Initialization**: Sets up the VGG16 model with the specified parameters
- **Feature Extraction**: Processes images to extract embeddings using the pre-trained model

- **Similar Image Search**: Compares the target image embedding with all others using cosine similarity
- **Results Export**: Saves the search results to a CSV file

2.7 Results

2.7.1 Histogram-based Search

Using the Chi-Square distance metric (lower values indicate more similarity), our histogram-based search found the following most similar images to the target image (image_0001.jpg), listed first:

Rank	Filename	Distance
1	image_0001.jpg	0.0000
2	image_0597.jpg	4.8699
3	image_0594.jpg	5.0323
4	image_0614.jpg	5.5547
5	image_0104.jpg	5.6719
6	image_1126.jpg	5.6807

Manual inspection of these results showed that the histogram approach successfully identified visually similar photos in terms of color composition - most featured yellow flowers in the center with some green in the background, matching the color pattern of the target image.

2.7.2 Embedding-based Search

Using VGG16 embeddings and cosine similarity (higher values indicate more similarity), our embedding-based search found the following most similar images to the target image (image_0001.jpg), listed first:

Rank	Filename	Similarity
1	image_0001.jpg	1.0000
2	image_0037.jpg	0.8675
3	image_0016.jpg	0.8610
4	image_0036.jpg	0.8394
5	image_0017.jpg	0.8376
6	image_0049.jpg	0.8361

What's particularly interesting is that manual inspection of these results revealed all the similar images are yellow daffodils. Unlike the histogram method which simply matched color patterns, the CNN approach recognized the specific flower species in the image. This nicely illustrates how convolutional neural networks can capture a wider range of structural features rather than just low-level visual features.

These results show how different feature extraction methods can yield different notions of similarity. The histogram-based approach focuses more on color distribution, while the embedding-based approach captures higher-level semantic features, resulting in completely different sets of similar images.

3 Assignment 2: CIFAR-10 Image Classification

This assignment implements classification models for the CIFAR-10 dataset using scikit-learn. The implementation includes two classification approaches:

1. Logistic Regression classifier
2. Neural Network classifier (MLPClassifier)

3.1 Quickstart

The simplest way to run the assignment is using the provided run.sh script:

```
./run.sh
```

Then select option 2 from the menu.

Note: Using the run.sh script will execute the assignment with default configurations only. For customized runs with different parameters, see the Configuration section below. To run the assignment without run.sh, navigate to the src directory and execute:

```
cd src
uv run python -m assignment_2.main
```

3.1.0.1 Configuration

There are two ways to customize the configuration parameters:

1. **Edit config.py:** Modify the default configuration values directly in the `config.py` file.
2. **Use command line options:** Override specific settings using command line flags when running the script.

You can customize the run using command line options which will override the corresponding settings in `config.py`. The main.py script accepts the following commands:

```
--model  Which model to run [logistic_regression, neural_network, both]
```

Example usage:

```
cd src
uv run python -m assignment_2.main --model neural_network
```

When the `--model` flag is provided, it overrides the `run_models` setting in the configuration. If not provided, the script uses the value from `config.py`.

3.2 Project Structure

```
src/assignment_2/
├── README.md                          # This documentation file
├── main.py                          # Main script to run classifiers
├── config.py                       # Configuration class for all parameters
├── models/                        # Classification model implementations
│   ├── base_classifier.py          # Abstract base class for classifiers
│    ├── logistic_regression.py
```

```
│   └── neural_network.py
├── utils/                          # Assignment-specific utilities
│   ├── cifar_10.py
│   ├── image_utils.py
│   └── model_evaluation.py
└── output/                     # Primary results and output files directory
        ...
```

Note: This project also has dependencies on the shared_lib module which provides common utilities and services for image processing, file handling, logging, model evaluation, and visualization.

3.3 Architecture Overview

The implementation follows a modular architecture with the following components:

- **Data Processing**: Loading and preprocessing the CIFAR-10 dataset (conversion to grayscale, normalization)
- **Model Training**: Training both Logistic Regression and Neural Network classifiers
- **Evaluation**: Generating classification reports, confusion matrices, and performance metrics
- **Visualization**: Plotting loss curves (for Neural Network)
- **Configuration module**: Uses Pydantic models for flexible configuration

3.4 Data

3.4.1 Data Splitting

It's worth noting how the data splits are not identical when training each model:

1. **Initial Split**: The CIFAR-10 dataset comes pre-divided into 50,000 training images and 10,000 test images.
2. **Logistic Regression Classifier**:
    - Uses a simple two-way split

- Training set: All 50,000 training images
- Test set: 10,000 images (original test set)
- No validation set is needed since there's no early stopping mechanism

3. **Neural Network Classifier** (with early stopping enabled):
    - Creates a three-way split of the data
    - Training set: 45,000 images (90% of original training data)
    - Validation set: 5,000 images (10% of original training data, controlled by validation_fraction parameter)
    - Test set: 10,000 images (original test set)
    - The validation set is created internally by scikit-learn's MLPClassifier and used to monitor performance for early stopping

3.4.2 Data Preprocessing

1. Load the CIFAR-10 dataset using TensorFlow's keras.datasets
2. Convert images to grayscale (optional)
3. Normalize pixel values to [0, 1]
4. Flatten the images to be compatible with scikit-learn classifiers

3.5 Model Overview

3.5.1 Logistic Regression Classifier

The implementation uses scikit-learn's LogisticRegression with the following default parameters: - Solver: saga (efficient for larger datasets) - Maximum iterations: 1000 - Tolerance: 0.001 - Regularization parameter (C): 0.1

Note that in scikit-learn's LogisticRegression: - `max_iter` refers to the maximum number of solver iterations for convergence (not epochs) - The solver will stop earlier if the model converges (when the improvement in loss is less than `tol`) - The actual number of iterations is typically lower than the maximum when convergence is reached (in our case, 121 iterations)

3.5.2 Neural Network Classifier

The implementation uses scikit-learn's MLPClassifier with the following default parameters: - Hidden layer sizes: (200, 100, 50) - Activation: ReLU - Solver: adam - Alpha: 0.0001 - Learning rate: adaptive - Batch size: 200 - Maximum iterations: 200 - Early stopping: True - Validation fraction: 0.1 - n_iter_no_change: 10 - Tolerance: 0.0001

Note that in scikit-learn's MLPClassifier: - `max_iter` directly corresponds to the maximum number of epochs (complete passes through the training data) - With `early_stopping` enabled, training may stop before reaching the maximum number of epochs - Training stops after `n_iter_no_change` consecutive epochs without improvement on the validation set - The actual number of epochs conducted is typically lower than the maximum (as shown in the Results Analysis section, the model stopped after 50 epochs due to early stopping)

3.6 Output

The classifiers generate the following outputs in the specified output directory:

1. **Classification Report**: Precision, recall, and F1-score for each class
2. **Confusion Matrix**: Visual representation of prediction accuracy
3. **Loss Curve**: Training loss over iterations (Neural Network only)
4. **Model Information**: Model parameters and training details

3.7 Notes on Performance

- The Logistic Regression classifier provides a baseline performance but is limited in capturing complex patterns in image data.
- The Neural Network classifier generally achieves higher accuracy but requires more computational resources.
- Converting to grayscale reduces dimensionality (and training time) but impacts accuracy for color-sensitive classes.

3.8 Results Analysis

The experiments on the CIFAR-10 dataset shows significant performance differences between the two classification approaches. Analysis of both models demonstrates clear

strengths and weaknesses when classifying different object categories. Object categories with distinct structural features (vehicles) were consistently better classified across both models, while classes with variable appearances and postures (animals) presented greater challenges.

3.8.1 Logistic Regression Classifier

- Achieved an overall accuracy of 29.59% on the test set, establishing a baseline performance
- Best performance on vehicle classes: trucks (40.28% F1-score) and automobiles (36.84% F1-score)
- Struggled significantly with animal classes, particularly cats (18.64% F1-score) and deer (20.83% F1-score)
- Converged after 121 iterations with the saga solver, well before the maximum 1000 iterations limit
- Training completed efficiently due to the relatively simple model architecture

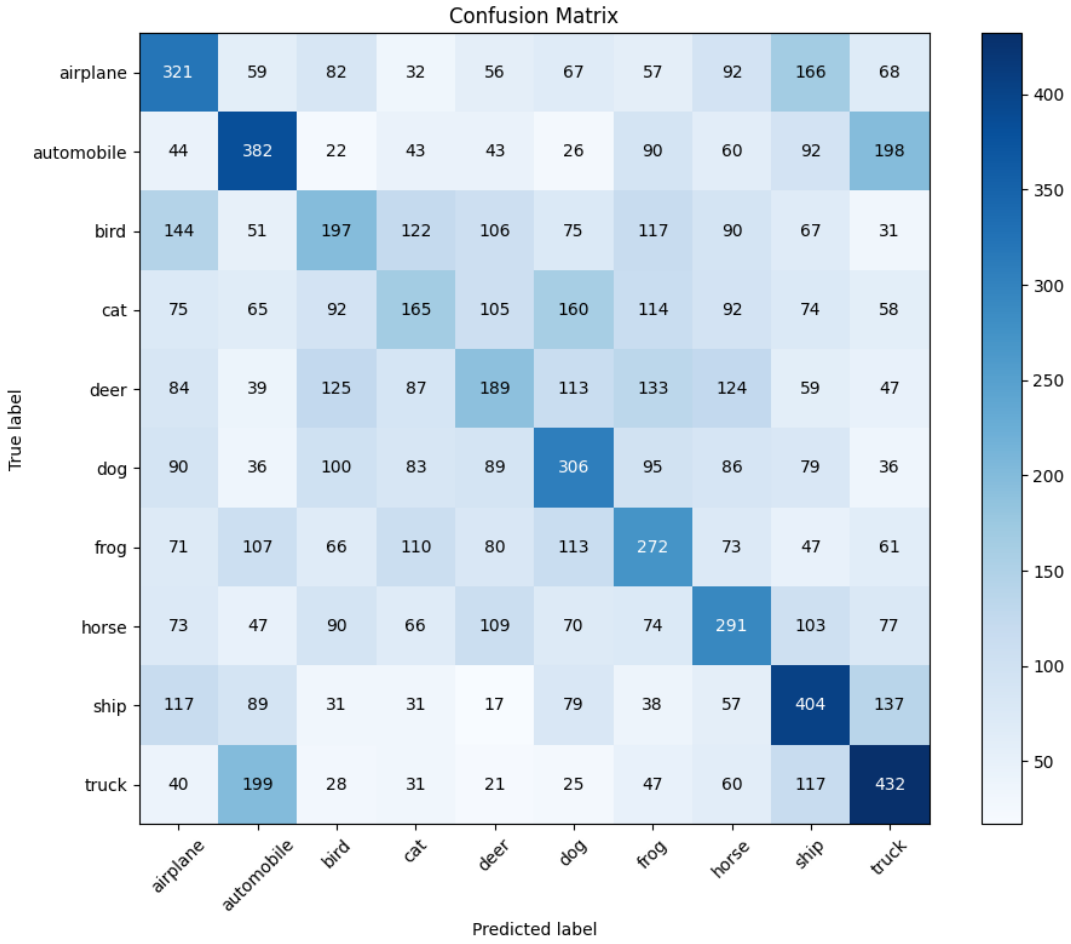

Figure 1: Confusion matrix for the Logistic Regression classifier showing classification performance across the 10 CIFAR-10 classes. Diagonal elements represent correct classifications, while off-diagonal elements show misclassifications.

3.8.2 Neural Network Classifier

- Demonstrated substantially better performance with an overall accuracy of 43.85% (14.26% improvement over logistic regression)
- Strongest classification performance on vehicles: trucks (51.44% F1-score), ships (51.24% F1-score), and automobiles (50.58% F1-score)
- Despite improved performance overall, still struggled with cat classification (24.88% F1-score)
- Early stopping triggered after 50 epochs (of maximum 200), indicating efficient convergence, though the loss curve suggests potential for further improvement

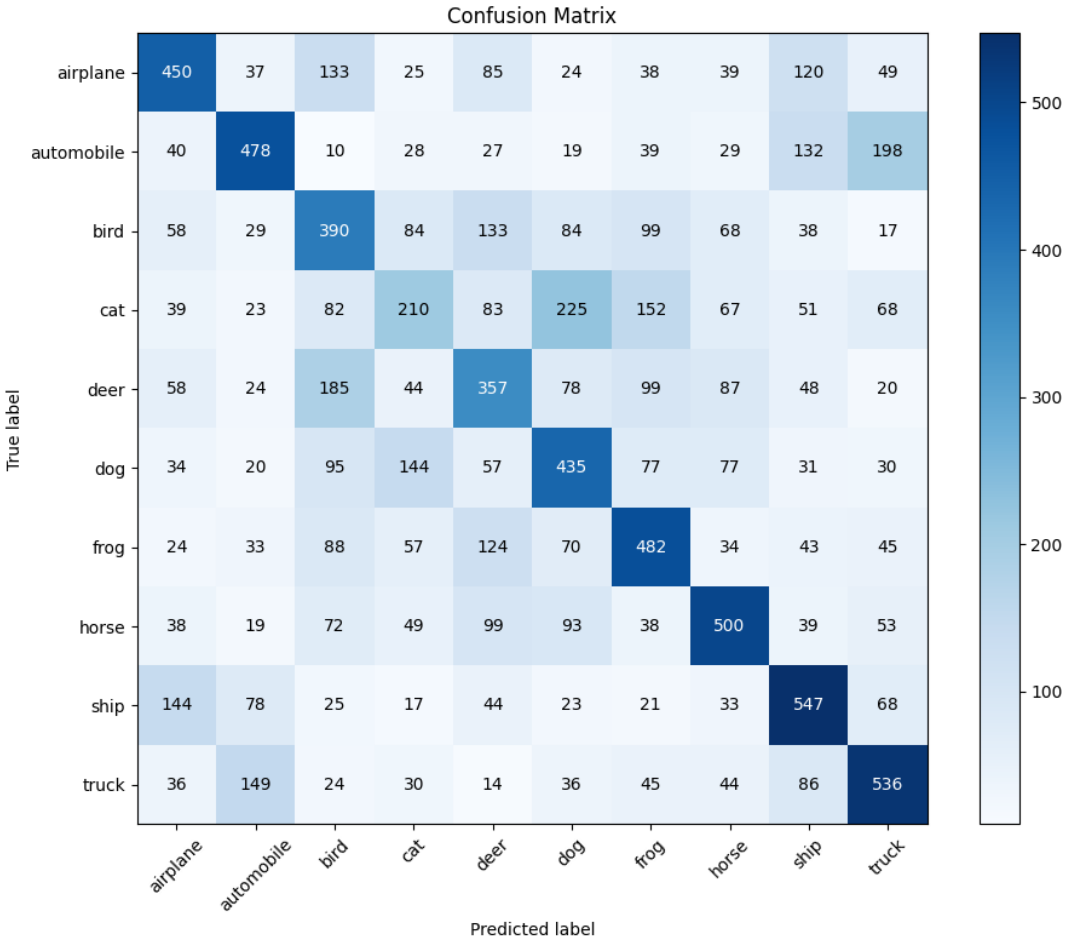

Figure 2: Confusion matrix for the Neural Network classifier. Note the improved diagonal values compared to Logistic Regression, indicating better classification performance across most classes.

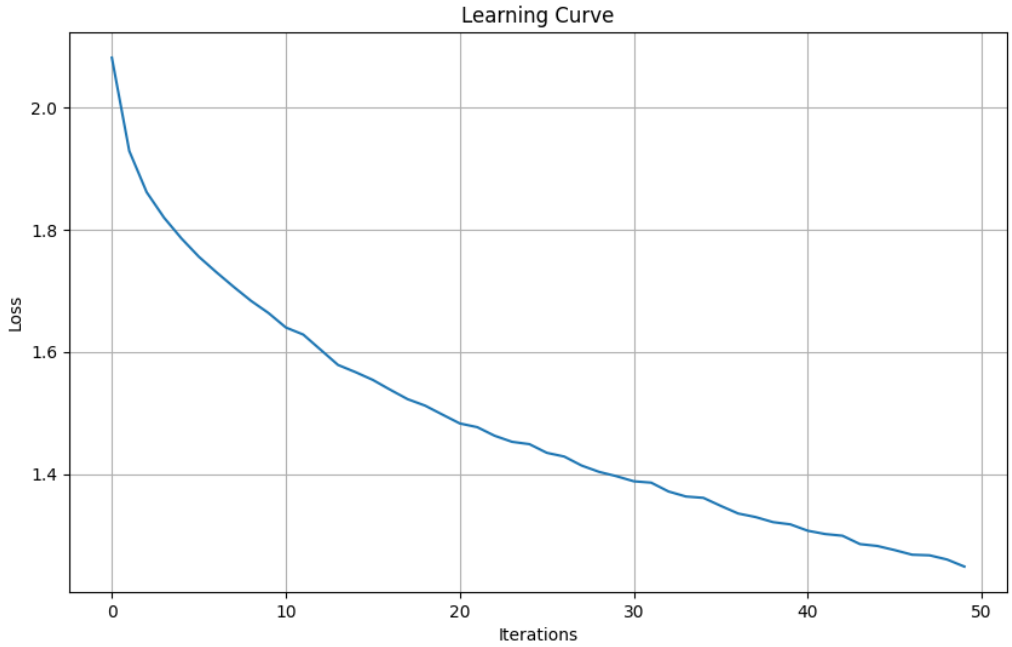

Figure 3: Loss curve showing the neural network's training progression over 50 epochs. The decreasing trend shows continuous improvement without plateauing, suggesting early stopping may have been triggered prematurely despite the model's potential for further optimization. However, the early stopping was triggered due to the accuracy on the validation data not increasing meaningfully over 10 epochs.

3.9 Limitations

Several important limitations affected the performance and scope of this classification work:

3.9.1 Data Preprocessing Limitations

- Converting images to grayscale significantly reduces the information available to the models, particularly affecting classes distinguished by color (e.g., birds)
- No data augmentation was implemented, limiting the models' ability to generalize across different object orientations and scales

3.9.2 Model Architecture Limitations

- The logistic regression model assumes linear separability between classes, which is unrealistic for complex image data
- The MLPClassifier implementation in scikit-learn lacks modern deep learning techniques like convolutions, which are specifically designed for image data
- Both models are very simple compared to state-of-the-art computer vision approaches, explaining the relatively low accuracy.

3.9.3 Computational Constraints

- Using scikit-learn's MLPClassifier without GPU acceleration limits the feasible model size and training time
- The neural network parameters were constrained by computation time considerations rather than optimal performance
- Hyperparameter tuning was limited, with only a single configuration tested for each model type

3.9.4 Evaluation Limitations

- The evaluation metrics focus on classification accuracy without considering confidence scores or error analysis
- No cross-validation was performed
- No analysis of misclassified examples was conducted to identify potential improvements

3.10 References

- CIFAR-10 dataset: https://www.cs.toronto.edu/~kriz/cifar.html
- scikit-learn documentation: https://scikit-learn.org/stable/
- Pydantic documentation: https://docs.pydantic.dev/

4 Assignment 3: Transfer Learning with Pretrained CNNs

4.1 Overview

This assignment implements two approaches for classifying Lego brick images:

1. A custom CNN trained directly on the image data
2. A transfer learning approach using a pre-trained VGG16

The goal is to compare these two approaches and determine whether transfer learning improves performance for this specific image classification task.

4.2 Dataset

- The dataset consists of images of various Lego bricks arranged in folders named after the brick type.
- We use only the data from the `data/lego/cropped` folder, which contains images with backgrounds removed
- The data should be placed in the `data/lego/cropped/*` directory.

4.3 Quickstart

The simplest way to run the assignment is using the provided run.sh script:

```
./run.sh
```

Then select option 3 from the menu.

You can also run the code without run.sh:

```
# Navigate to src from the project root
cd src

# Run as module
uv run python -m assignment_3.main
```

4.4 Configuration

The project uses a central configuration system in `config.py` that can be modified to customize various aspects:

- Data Configuration: Control image dimensions and data directory paths settings
- CNN Model Configuration: Adjust architecture parameters, learning rates, and training settings
- VGG16 Model Configuration: Configure transfer learning parameters, fine-tuning depth, and training settings
- Output Configuration: Set output directory paths for saving models, reports, and visualizations

All settings are managed through Pydantic classes, making it easy to modify behavior without changing code.

4.4.0.1 Command Line Arguments

- `--data-dir`: Path to the Lego data directory (must exist and be a directory)
- `--output-dir`: Path to save the output (default: ./src/assignment_3/output)
- `--cnn-only`: Train only the CNN model
- `--vgg16-only`: Train only the VGG16 model

4.5 Project Structure

```
src/assignment_3/
├── config.py                    # Configuration settings
├── main.py                      # Main entry point
├── assignment_description.md    # Assignment description
├── README.md                    # Project documentation
├── session9_inclass_rdkm.ipynb  # In-class notebook reference
├── data/                        # Data modules
│   └── data_loader.py           # Data loader
├── models/                      # Service modules
│   ├── base_classifier_model.py # Base model class
│   ├── cnn_model.py             # Direct CNN classifier
```

```
│   └── vgg16_transfer_learning_model.py  # VGG16-based classifier
├── utils/                          # Utility functions
│   └── model_comparison.py         # Model comparison utilities
└── output/                         # Output directory
    ├── cnn/                        # Output for CNN model
    │   ├── classification_report.txt
    │   ├── learning_curves.png
    │   └── training_history.json
    ├── vgg16/                      # Output for VGG16 model
        ├── classification_report.txt
        ├── learning_curves.png
        └── training_history.json
```

4.6 Implementation Details

4.6.1 Data Processing

- Images are resized to 224×224 pixels with 3 color channels
- Batches of 32 images are processed at a time
- Data is split with 80/10/10 % train, test and validation.
- We experimented with various data augmentation techniques (rotation, zoom, flip, shift) but found that models achieved better validation results without augmentation. The final implementation therefore uses the original images without augmentation.

4.6.2 CNN Model Architecture

The custom CNN is configured with the following architecture: - Input shape: (224, 224, 3) - Convolutional layers with increasing filter complexity: [32, 64, 128] - 3×3 kernel sizes with 2×2 max pooling - Two dense layers [512, 256] with ReLU activation - Dropout (0.5) for regularization - Softmax output layer for classification - Adam optimizer with learning rate 0.001 - Early stopping with patience=8 to prevent overfitting - Trained for up to 30 epochs (may stop earlier with early stopping)

4.6.3 VGG16 Transfer Learning Model

- We're using VGG16 (pretrained on ImageNet) as our foundation
- We've removed the top classification layers (include_top=False)
- We're fine-tuning the last 4 convolutional layers
- We're applying global average pooling to simplify feature maps
- The classifier contains:
  - Two dense layers (256→128 units)
  - BatchNormalization to stabilize training
  - Dropout (0.5) to prevent overfitting
  - Standard softmax output for classification
- We use SGD with momentum (0.9) and a lower learning rate (0.0005)
- Early stopping with patience=8 to prevent overfitting
- We are training for up to 30 epochs (may stop earlier with early stopping)

4.6.3.1 Transfer Learning Strategy Our implementation freezes earlier VGG16 layers while making only the final 4 convolutional layers trainable. This approach makes use of the hierarchical nature of convolutional networks, where initial layers capture universal visual primitives (edges, textures, basic shapes) that generalize well across domains, while deeper layers represent increasingly task-specific features.

First, it addresses the potential for overfitting given our relatively constrained dataset size. By limiting the number of trainable parameters, we create a more favorable ratio between trainable weights and available training examples.

Second, this approach significantly reduces computational requirements compared to full fine-tuning.

Third, the method creates an balance between knowledge transfer and domain adaptation. By preserving the robust feature extractors from VGG16's early layers (trained on millions of ImageNet images) while allowing later layers to adapt to Lego-specific characteristics,

Finally, selective freezing provides gradient stability benefits during training. By reducing the network's effective depth from the perspective of backpropagation, we mitigate vanishing gradient issues and prevent catastrophic forgetting of useful pre-trained features.

The transfer learning configuration is parameterized through the `trainable_layers` setting, allowing experimentation with freezing strategies based on dataset characteristics and

target domain similarity to the ImageNet source distribution.

4.6.4 Evaluation Methodology

Our evaluation framework implements a three-way data partitioning, allocating 80% of available data for model training, 10% for validation-based early stopping decision, and 10% for unbiased final performance assessment. This approach prevents potential data leakage between model selection and evaluation phases, giving a more realistic estimate of model performance.

Performance quantification incorporates both primary metrics (accuracy) and secondary distributional metrics (precision, recall, F1-score) to provide insight into classification behavior across all class categories. Temporal learning dynamics are visualized through epoch-wise training and validation metrics, enabling identification of potential optimization issues including underfitting, overfitting, and convergence patterns.

4.7 Results

Our experiments comparing custom CNN and transfer learning approaches for Lego brick classification showed benefits of pre-trained models.

4.7.1 Performance Comparison

The custom CNN achieved 84.40% test accuracy after 28 epochs, with training and validation accuracies of 95.58% and 86.16% respectively. Despite early stopping, the growing gap between training and validation metrics indicated the model was beginning to overfit.

The VGG16 transfer learning approach substantially outperformed the custom CNN with 96.58% test accuracy - a 12 percentage point improvement. The VGG16 model demonstrated rapid learning (reaching >93% validation accuracy by epoch 8) and exceptional generalization, with validation (97.10%) and test (96.58%) accuracies actually exceeding training accuracy (93.40%).

This performance difference shows transfer learning's effectiveness in classification tasks. The model efficiently adapted general visual pattern recognition to Lego-specific characteristics.

4.7.2 Learning Comparison

The custom CNN showed a somewhat uneven learning curve with fluctuations in validation metrics, though with an overall positive trajectory. The training accuracy steadily increased to 95.58% by the final epoch, while validation accuracy peaked at 86.16% in epoch 25 before declining slightly, triggering early stopping. This growing gap between training and validation metrics indicates the model was beginning to overfit to the training data.

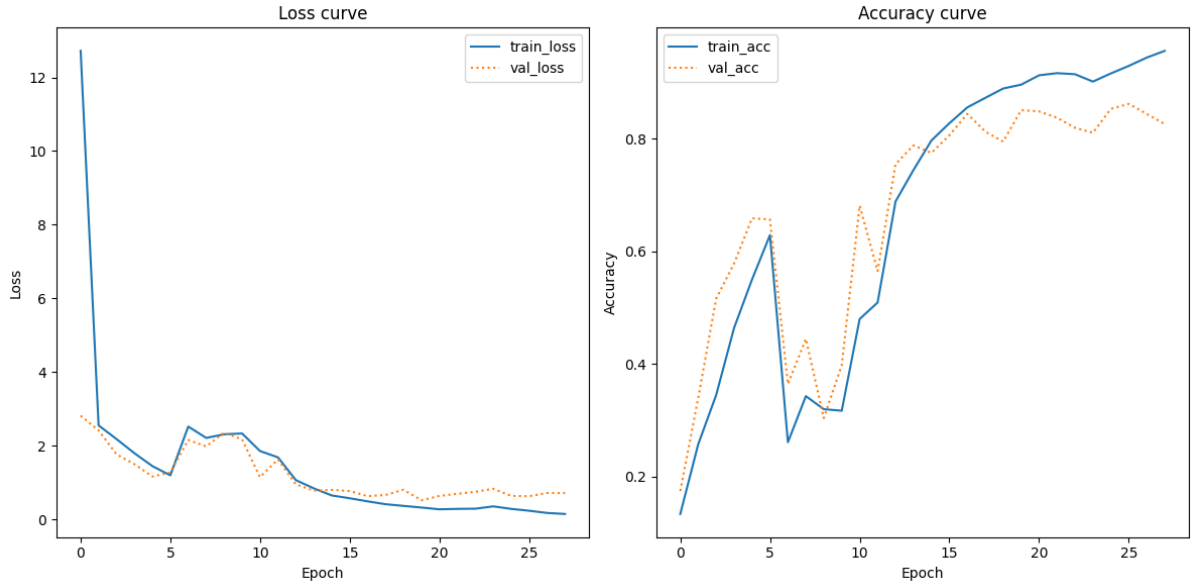

Figure 1: Training and validation loss/accuracy curves for the custom CNN model

The VGG16 model exhibited better learning with some fluctuations in validation accuracy but an overall stronger trajectory. By epoch 8, it had already achieved validation accuracy above 93%. The final validation accuracy of 97.10% actually exceeds the training accuracy of 93.40%, and this pattern continues with the test accuracy of 96.58% - a sign of excellent generalization across all three data splits.

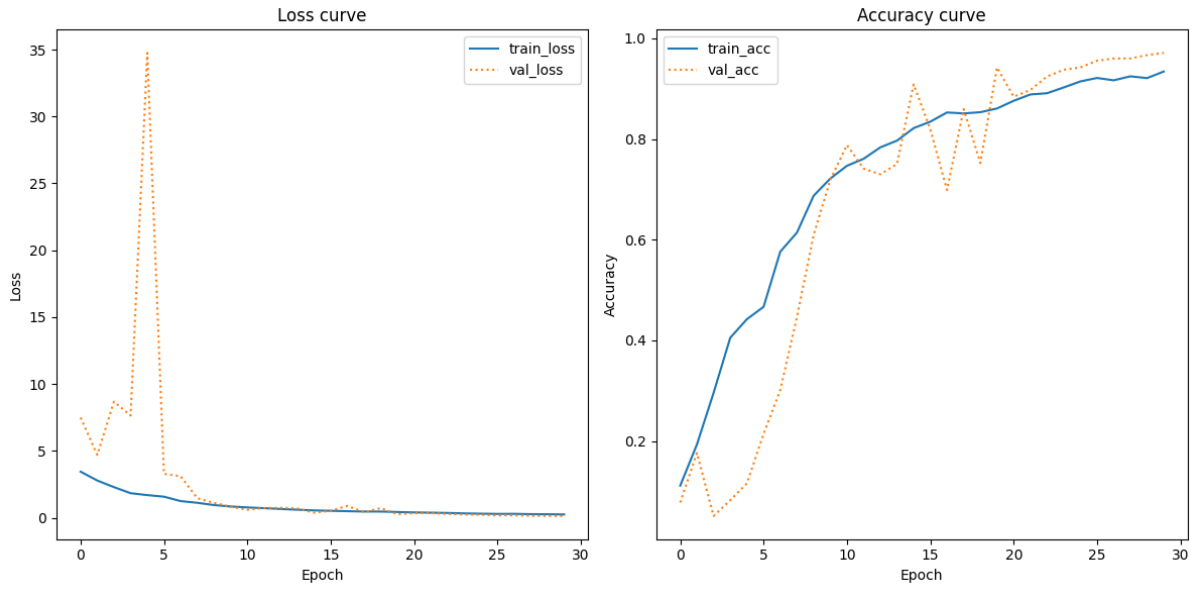

Figure 2: Training and validation loss/accuracy curves for the VGG16 transfer learning model

The big difference in performance between the two models shows why transfer learning works so well for specific image tasks like this one. By using VGG16's pre-trained weights from ImageNet, our model started with ready-made feature detectors that could seemingly already spot patterns in Lego bricks - things like edges, colors, textures, and shapes. By fine-tuning just the last 4 convolutional layers, we let the model adapt these general features to the specific characteristics of Lego bricks.

The jaggedness visible in both models' validation curves (particularly pronounced in the CNN model) is likely due to the relatively small validation set. These fluctuations are a common challenge in ML practice and directly influenced the training setup - specifically, the early stopping patience was increased from 5 to 8 epochs to prevent prematurely halting training during these unstable periods. This adjustment allowed both models to reach more stable performance in later epochs, providing more reliable results for our comparison.

4.7.3 Limitations

We did not perform hyperparameter optimization for either model, instead using fixed configurations. We implemented early stopping with a patience of 8 epochs to prevent overfitting. Additionally, we limited our exploration to two specific architectures.

Although our initial experiments with data augmentation showed better validation results without augmentation, further experimentation with different augmentation strategies

(such as more targeted transformations specific to the Lego domain) might further improve results. Similarly, more extensive fine-tuning of the VGG16 layers might yield even better performance. These limitations present opportunities for future work to further improve performance on the Lego classification task.

5 Assignment 4: Detecting Faces in Historical Newspapers

5.1 Overview

This project analyzes the presence of human faces in historical Swiss newspapers over time. Specifically, we examine three historic Swiss newspapers:

- The Journal de Genève (JDG)
- The Gazette de Lausanne (GDL)
- The Impartial (IMP)

The analysis uses a pre-trained CNN model (MTCNN) to detect faces in newspaper page images, groups the results by decade, and visualizes trends in the prevalence of human faces in print media over approximately 200 years.

5.2 Data

This project uses the Swiss newspapers corpus from this Zenodo dataset. The dataset contains:

- 1008 newspaper pages from the Gazette de Lausanne (1790s-1990s)
- 1982 newspaper pages from the Journal de Genève (1820s-1990s)
- 1634 newspaper pages from the Impartial (1880s-2010s)

The images should placed in `data/newspapers/images/` and organized into three subfolders (GDL, JDG, IMP) corresponding to each newspaper.

5.3 Quickstart

The simplest way to run the assignment is using the provided run.sh script:

`./run.sh`

Then select option 4 from the menu.

Alternatively, you can run assignment 4 without run.sh as a module:

```
# Ensure you are in the right directory (src)
cd src

# Process all newspapers
uv run -m assignment_4.main

# Process only a specific newspaper
uv run -m assignment_4.main --newspaper GDL
```

5.4 Configuration

The project uses a central configuration system in `config.py` that can be modified to customize various aspects:

- Face Detection Configuration: Control MTCNN model parameters including detection thresholds, minimum face size, and scale factor
- Data Configuration: Set data directory paths and specify which newspapers to analyze
- Output Configuration: Define output directory paths for saving results and visualizations

5.4.1 Command Line Options

Furthermore, a few options are also available via the CLI:

- `--data-dir`: Path to the newspaper images directory (must exist and be a directory)
- `--output-dir`: Path to save the output
- `--newspaper`: Process only a specific newspaper (choices: GDL, JDG, IMP)

5.5 Project Structure

`assignment_4/`

```
├── config.py                    # Configuration settings
├── main.py                     # Main script to run the analysis
├── services/                    # Service modules
│   ├── data_service.py          # Data loading and processing
│   └── face_detection_service.py  # Face detection with MTCNN
├── utils/                       # Utility modules
│   └── visualization.py         # Visualization utilities
└── output/                      # Output directory
    ├── results/                 # CSV results
    └── plots/                   # Visualization plots
```

5.6 Methodology

The analysis follows these steps:

1. Data Loading: Load newspaper images from the dataset.
2. Year Extraction: Extract the year of publication from each filename.
3. Face Detection: Use MTCNN to detect faces in each newspaper page.
4. Decade Aggregation: Group results by decade to analyze temporal trends.
5. Visualization: Create plots showing the percentage of pages with faces over time.

5.7 Results

The analysis produces the following outputs:

1. CSV Files:
    - Individual CSV files for each newspaper showing face count and percentage by decade
    - A combined CSV with results from all newspapers
    - Summary statistics
2. Visualizations:
    - Individual plots for each newspaper showing percentage of pages with faces by decade
    - A comparison plot showing trends across all newspapers

The outputs are saved to the `output/` directory.

Initial detection using default MTCNN thresholds ([0.6, 0.7, 0.7]) produced significant false positives, especially in early decades (1790s-1840s) when photography wasn't yet used in newspapers. This observation led to adjusting the thresholds to [0.7, 0.9, 0.9], substantially increasing the model's strictness. The refined results better align with the historical development of photography and its adoption in print media.

5.8 Analysis

After running our face detection analysis with adjusted thresholds on the three Swiss newspapers, we see more historically accurate trends in how human faces have been represented in print media over the past two centuries.

5.8.1 Overall Trends

The late 19th century saw newspapers starting the adoption of photography (with faces) in print media.

Looking at the data chronologically: - Pre-1850s: Minimal to no faces detected, consistent with the technological limitations of the era - 1850s-1900s: Gradual introduction of faces, corresponding to early photography adoption in newspapers - 1900s-1950s: Steady increase as photo reproduction techniques improved - 1950s-2010s: Dramatic rise, hinting towards a modern visual journalism

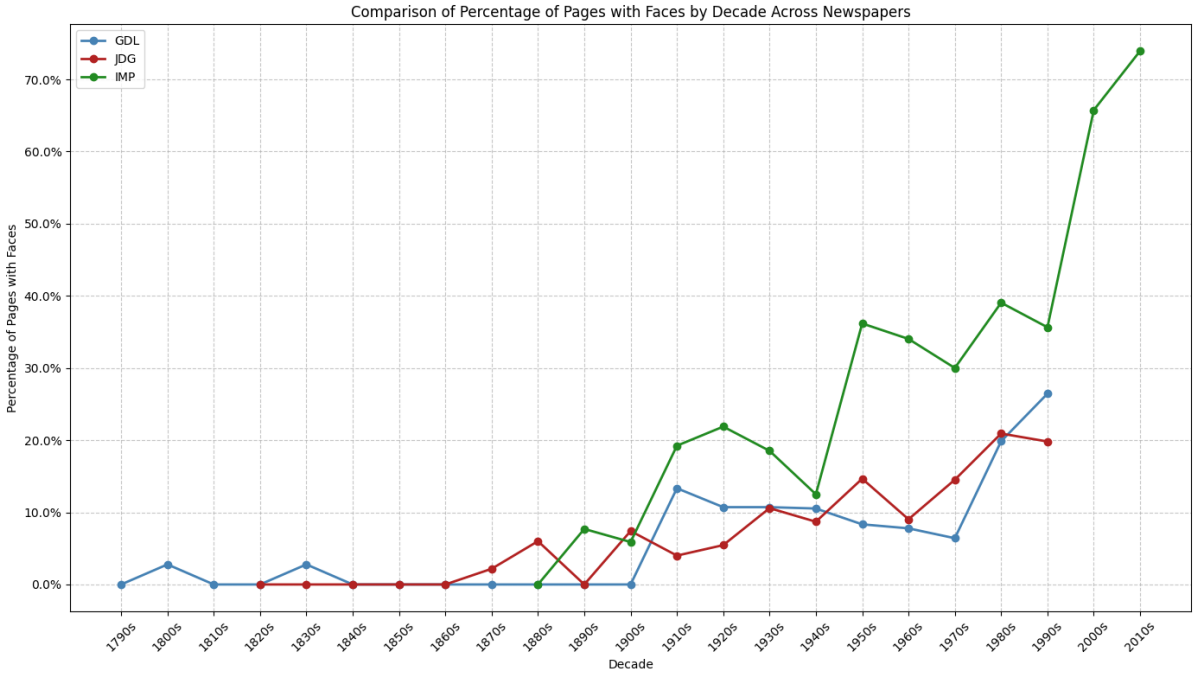

Figure 1: Newspaper Comparison

As shown in the comparison visualization above, GDL and JDG followed similar trajectories in adoptation throughout the decades, where as IMP rapidly increased the use of faces from the 1950's and up until the most recent decades included faces in their newspapers significantly more than GDL and JDG. It's also worth noting that we only have data for GDL and JDF up to the 1990's. A search on Google showed that these two newspapers in fact merged in 1991 and continued under a new name, which is a good explanation as to why we don't have any data after that.

5.8.2 Individual Newspaper Patterns

5.8.2.1 The Impartial (IMP, 1881-2017) IMP shows a distinctive pattern: - Starting with 0% in the 1880s, faces appear in around 8% of pages in the 1890s - Early 1900s shows relatively low percentages (5-6%) - A significant jump occurs in the 1910s-1920s, reaching around 20% - The percentage stabilizes around 18-22% through the 1930s - A noticeable jump to 36% in the 1950s - After some fluctuation in mid-century, there's a dramatic increase in the late 20th/early 21st century - The most recent periods show high prevalence - reaching over 65% in the 2000s and exceeding 70% in the 2010s

5.8.2.2 The Gazette de Lausanne (GDL, 1804-1991) For GDL, the results indicate: - The earliest periods (1790s-1850s) show minimal face detection (0-3%) which we can interpret as being false positives, as the first photography was used in print media in 1848 ("Photojournalism", Wikipedia 2025). - - No faces detected from 1840s through 1890s - A clear emergence starting in the 1900s (13%) - Consistent presence in the 1910s-1930s (10-11%) - Slight decline in mid-century (8-6%) - The percentages increase significantly in the 1980s-1990s (20-26%)

5.8.2.3 The Journal de Genève (JDG, 1826-1994) JDG shows a gradual adoption of facial imagery: - No faces detected prior to the 1870s - Initial appearance in the 1870s at very low levels (2%) - Steady increase through the late 19th century (6% by 1890s) - Relatively stable presence in early 20th century (4-6%) - Gradual increase from the 1920s through 1940s (5-11%) - More significant presence by mid-century (14-15% in 1950s-1960s) - Reaching peak levels in the 1970s-1980s (around 21%) - Slight decrease in the 1990s (19.8%)

JDG's pattern shows a more consistent, gradual increase compared to the other newspapers, with fewer dramatic shifts between decades.

5.8.3 Historical Context and Significance

These patterns reflect several important developments:

1. Technological Evolution: The introduction of photography made it increasingly feasible and affordable to include images in newspapers, clearly visible in the minimal detection before 1870s and gradual increase thereafter ("Photojournalism", Wikipedia 2025).
2. Modern Visual Culture: The dramatic increase in the most recent decades (particularly for IMP exceeding 70% by the 2010s) mirrors broader cultural shifts toward visual media
3. Regional Differences: The significant difference between IMP and the other two newspapers might reflect different editorial philosophies or regional preferences. IMP shows a notable early adoption in the 1910s-1920s (20%) and dramatically accelerated use by the 1950s (36%), while GDL and JDG followed more conservative trajectories.

It's worth noting that our dataset shows an uneven distribution of pages across decades, with more pages available from recent decades. This reflects the changing volume of newspaper

production over time but may also influence the reliability of our decade-to-decade comparisons.

5.9 Limitations

This analysis has several limitations to consider:

1. Model Training Data: MTCNN was only trained on modern photographs, making it prone to misdetecting other objects as faces in historical materials. Other face detection algorithms might perform differently on historical materials.
2. Image Quality: Historical newspaper scans vary in quality, affecting face detection accuracy.
3. Temporal Distribution: Uneven distribution of pages across time periods may affect decade-to-decade comparisons.
4. Detection Thresholds: Analysis required adjusted confidence thresholds ([0.7, 0.9, 0.9] vs default [0.6, 0.7, 0.7]) to reduce false positives. This adjustment was based on visual inspection rather than systematic optimization.
5. Qualitative Analysis: Our analysis counts faces but doesn't distinguish between photographs, illustrations, or engravings, nor considers face size, placement, or prominence.
6. Lack of Ground Truth: Without manually annotated historical newspaper data, we couldn't quantitatively evaluate detection performance across different eras.

5.10 Credits

This project was created as part of the Cultural Data Science - Visual Analytics course at Aarhus University.

- Dataset: Swiss newspapers corpus from Zenodo
- Face detection: FaceNet-PyTorch implementation of MTCNN
- Note in MTCNN training on modern photographs: FaceNet: A unified embedding for face recognition and clustering
- First photography in newspaper: Wikipedia