# 1 Introduction

The members of this group are: Alberto Tamajo(**at2n19**), Alessandro Nerla(**an1g19**), Giovanni Arcudi(**ga1g19**) and Jury D'Alessio(**jd3n18**). Every member of the group has contributed equally.

In what follows, a description of the implementation of the classifiers for the three runs, including information on how they were trained and tuned, and the specific parameters used for configuring the feature extractors and classifiers, will be provided.

# 2 Run 1

## 2.1 Description

The goal of Run 1 is to develop a k-nearest-neighbour classifier that uses the "tiny image" feature extractor. The "tiny image" feature extractor represents an image through a vector. This vector is the result of cropping an image to a square about the centre, resizing it to a small, fixed resolution and then concatenating each row into a one-dimensional feature. Mean centring and normalising the "tiny image" vector leads to a slightly better representation.

## 2.2 Implementation

The implementation of Run 1 is straightforward if the functionalities provided by the OpenImaj library are well understood and used. Indeed, OpenImaj provides a class, "KNNAnnotator", that allows training a k-nearest-neighbour classifier out of the box. On the other hand, OpenImaj does not implement the "tiny image" extractor. Consequently, a class named "TinyImageExtractor" has been created to implement the feature extractor in question. This class implements the functional interface "FeatureExtractor" and overrides the "extractFeature" method appropriately. The "tiny image" vector returned by the "extractFeature" method is mean-centred and normalised. To ease the training process, a class named "KNearestNeighborTinyImages" has been created as well. It extends the "KNNAnnotator" and uses the "TinyImageExtractor" by default. Given the nature of this task, the Euclidean distance comparison measure is used for the k-nearest-neighbour classifier.

## 2.3 Training and parameter tuning

Two parameters need to be tuned for this run: the number of neighbours $k$ and the crop width $w$. The crop width is the width of the square image obtained when cropping the original image about the centre. From the description of this run, it can be assumed that an arbitrary crop width can be used as long as the cropped image is then resized to the recommended fixed resolution (16x16). To find the optimal values for $k$ and $w$, 5-fold cross-validation has been run for all values of $w$ ranging from 16 to 200 and all values of $k$ ranging from 1 to 101. The highest value of $w$ is 200 because the smallest width or height in the training and test datasets is of the same magnitude. Thus, it would not be possible to crop the images in the dataset with a higher value of $w$. The utilities used for the training and evaluation of the different models are contained in the "TrainKNearestNeighborTinyImages" class.

## 2.4 Optimal parameters, accuracy and expectations

The pair of parameters that have achieved the highest average accuracy (22.4%) on the 5-fold cross-validation iterations are $k = 5$ and $w = 196$. While it was impossible to predict the optimal value for $k$ in advance, the optimal value of $w$ was expected, given that larger values of $w$ preserve more information than smaller ones. The "tiny image" feature extractor is not an effective image representation because it discards all high-frequency image content and is not translation, rotation, scale, and illumination invariant. Thus, the accuracy achieved in this run is in line with the effective extraction capabilities of the "tiny image" technique.

## 2.5 Test dataset predictions

Given that $k = 5$ and $w = 196$ are the optimal pair of parameters according to the 5-fold cross-validation, it seems reasonable to use them for training the "KNearestNeighborTinyImages" on the whole training dataset, and then annotate the images on the test dataset. This is exactly what has been done to produce the predictions contained in the run1.txt file.

# 3 Run 2

## 3.1 Description

This run involves the implementation of a set of linear classifiers that use a bag-of-visual-words feature based on fixed-size densely-sampled pixel patches. In other words, for each image, a feature vector is extracted by taking the pixels from the image patches, flattening them into a vector and then using vector quantisation to map each patch to a visual word. If the pixels from the image patches are flattened and then mean-centred and normalised, a better representation is obtained. The vector quantisation process can be carried out only after having learnt a codebook of visual words. A codebook of $K$ visual words is learnt by taking a set of samples from the training dataset and then performing K-means on all the features extracted from the dataset. The $K$ centroids learnt by the K-means algorithm represent the $K$ visual words of the codebook.

## 3.2 Implementation

The implementation of this run is a little bit more complicated than the previous one. While OpenImaj provides a class, "LiblinearAnnotator", that allows creating 15 one-vs-all linear classifiers automatically, it does not implement the feature extractor required for this run directly. Three classes have been created to implement the feature extractor: SlidingWindowExtractor, SlidingWindowLocalFeature, DenselySampledPatchesBvW

### 3.2.1 SlidingWindowExtractor

To the best of this group's knowledge, OpenImaj does not offer the users a class that slides a window across an image and extracts features from those regions according to a given feature extractor. This is the reason why the SlidingWindowExtractor class has been created. The SlidingWindowExtractor slides a window of a given width and height across an image according to a given step along the x and y directions. As the window slides, a feature is extracted from the region inside the window with the given feature extractor. This class implements the functional interface "FeatureExtractor". The "extractFeature" method returns a "LocalFeatureList" containing as many "SlidingWindowLocalFeature" as the number of features extracted during the sliding window process.

### 3.2.2 SlidingWindowLocalFeature

The "SlidingWindowLocalFeature" implements the "LocalFeature" interface. A "SlidingWindowLocalFeature" instance is produced to hold the feature extracted from a given region of the image as the sliding window moves. In addition to the feature extracted, it also stores the "SpatialLocation" associated with the feature. The x and y coordinates of the "SpatialLocation" are the x and y coordinates of the top-left left pixel of the region in question.

### 3.2.3 DenselySampledPatchesBvW

The "DenselySampledPatchesBvW" class extracts a bag-of-visual-words feature from an image based on a codebook learnt with a given training dataset. The codebook is learnt by clustering the features extracted from a set of training dataset samples. The features are extracted from the training dataset samples using a "SlidingWindowExtractor", which means-centres, normalises and then flattens the regions encountered. This class implements the "FeatureExtractor" interface so that it can be passed as argument to the "LiblinearAnnotator" class. The main constructor of this class requires the following arguments: the width and height of the sliding window, the steps along the x and y directions to be taken by the window, the number of visual words to be learnt, the number of samples to learn the visual words and in the end the training dataset. The samples drawn from the training dataset to learn the visual words contain an equal number of instances from each class. The quantiser of the class is learnt at construction time so that the returned instance is ready for extracting bag-of-visual-words features from an image. Given that training the quantiser requires a significant amount of time, two additional constructors are provided that allow the user to save the quantiser in a file and then successively load it.

## 3.3 Training and parameter tuning

The parameters that need to be tuned for this run are: the patch size, the direction steps, the number of visual words, the number of samples used to learn the codebook. To find the optimal values for these parameters, 5-fold cross-validation has been run for all possible combinations of the following values:

- patch size = [3,4,5,6,7,8,9,10]
- number of visual words = [500,600,700]
- number of samples used to learn the codebook = [750]
- direction steps = [4]

As it is possible to notice, the number of samples and direction steps have been kept fixed. This has been done because, unlike the previous run, performing 5-fold cross-validation for this run is a time-consuming task. Besides, decreasing the number of direction steps to less than four has thrown an "OutOfMemoryError". In any case, setting the number of samples used to 750 (half of the training dataset) and the direction steps along the x and y axis to 4 is believed to be a reasonable and adequate choice given the good performance that has been obtained. The linear classifier that is trained during each cross-validation iteration is a Support Vector Machine. The training parameters for the Support Vector Machine are C = 1 and eps = 0.00001. The utilities used for the training and evaluation of the different models are contained in the "TrainDenselySampledPatchesBvW" class. Given that learning the codebook is quite time-consuming, the assigner learnt by the "DenselySampledPatchesBvW" class is saved into a file for every possible combination of the parameters.

### 3.4 Optimal parameters, accuracy and expectations

The parameters that have achieved the highest average accuracy (66.3%) on the 5-fold cross-validation iterations are patch size = 3 and number of visual words = 700. These parameter values were expected. Given that the feature extractor used to encode each patch is not translation, rotation, scale, and illumination invariant, a large patch size would discard much more information than a small one. This is why the 3x3 patches perform better than all the other larger patch sizes. Besides, smaller patches extract a larger number of features from an image. Regarding the number of visual words, the larger the vocabulary of the codebook, the more distinctive the visual words become. However, if the vocabulary size becomes too large, the visual words become too distinctive, and the performance diminishes. 700 was expected to be the optimal vocabulary size for this run because it is larger than all the other vocabulary sizes but still not too large. The accuracy achieved in this run is in line with our expectations. Indeed, the method implemented for this run is much more advanced than the previous run's approach. Furthermore, it is believed that the accuracy achieved is almost the maximum reachable performance for this method, given that in the successive run, PHOW, a much more advanced approach, achieves 82.3% accuracy.

### 3.5 Test dataset predictions

Given that patch size = 3 and number of visual words = 700 are the optimal parameters according to the 5-fold cross-validation, it seems reasonable to train a Support Vector Machine on the whole training dataset, using as extractor the "DenselySampledPatchesBvW" class instantiated with such parameters, and then annotate the images on the test dataset. Given that the codebook for patch size = 3 and number of visual words = 700 was saved during the 5-fold cross-validation iterations, it only needs to be loaded by the instance of the "DenselySampledPatchesBvW" class. This is exactly what has been done to produce the predictions contained in the run2.txt file.

## 4 Run 3

### 4.1 Description

This run gives freedom of choice for the feature, encoding and classifier to be used. The goal is to develop the classifier that achieves the best performance on the test dataset.

### 4.2 Implementations

Two different approaches have been taken to develop the best possible classifier: traditional computer vision and deep learning.

#### 4.2.1 Traditional computer vision

When building a computer vision classification system, the choice of the feature extractor is a fundamental step in the traditional computer vision approach. Once a feature extractor is chosen, several machine learning algorithms can be used for the purpose of classification. Given that this task's goal is to develop the best possible classifier for the provided dataset, it is necessary to opt for a feature extractor that gives the best possible representation of the images. However, it is unknown

in advance which feature extractor achieves the best performance on a given dataset; only an empirical approach can show which feature extractor is optimal. However, this group has opted to try only one feature extractor, SIFT, rather than multiple ones due to lack of time. SIFT has been chosen because, generally, it is deemed to be one of the best feature extractors out there. More specifically, the architecture described in exercise 3, chapter 12 of the OpenImaj tutorial has been implemented. Three classifiers have been trained using the abovementioned feature extractor: Logistic regression, Linear Support Vector Machines , Non-linear Support Vector Machines.

Many different parameters have been tuned as well to try to squeeze the performance of each classifier as much as possible:

- The number of Dense SIFT steps
- The number of visual words
- The epsilon value (for SVMs)
- The training sample size used for the creation of the codebook
- The C value (for SVMs)
- The homogeneous kernel type (for non-linear SVMs)

In the end, the best performance (82.3% accuracy on the validation dataset) has been achieved by the non-linear SVM. The parameters used for achieving such performance are:

- Number of Dense SIFT steps: **3**
- Training sample size used for the creation of the codebook: **500**
- Epsilon value = **0.00001**
- Number of visual words: **484**
- C value: **50**
- Homogeneous kernel type: **CHI2**

The utilities used for the training and evaluation of the classifiers are contained in the "TrainPHOWExtractor" class.

#### 4.2.2 Deep Learning

According to this group, the best accuracy achieved through the Computer vision traditional approach is not satisfactory enough. Given that the Deep learning approach currently achieves state-of-the-art performance on several computer vision classification tasks, it is reasonable to implement a deep neural network architecture to improve the accuracy. Over the past few years, three deep learning frameworks (Tensorflow, Keras, and PyTorch) have gained momentum because of their ease of use, extensive usage in academic research, and commercial code and extensibility. This group has opted to use Keras because it is designed to provide fast experimentation with deep neural networks; thus, it allows us to experiment with different neural network architectures and select the model that achieves the best performance on the provided dataset. Google Colab Pro has been used to quickly train the deep neural network architectures, thanks to the GPU accelerator runtime environment. Training the neural network architectures has not been an easy task at first. Indeed, several models from the "keras.applications" package have been trained on the dataset (without using the pre-trained weights), but all of them have heavily overfitted the training data, achieving at most a 75% accuracy on the validation dataset, because the dataset is small. Data augmentation is a technique used to artificially expand the size of a training dataset to improve the generalisation performance of deep learning models. One possible way to expand a dataset is to apply random horizontal and vertical flips and rotations to the images. However, this solution has not solved the overfitting problem. The solution that has solved the overfitting problem is the simultaneous use of data augmentation, dropout, transfer learning and fine-tuning techniques. Dropout is a regularisation technique where randomly selected neurons are ignored during training so that to emulate the parallel training of several networks. Transfer learning is a technique that consists of taking features learned on one problem and leveraging them on a similar problem. This technique is commonly used when the training dataset is not sufficiently large to train a full-scale model from scratch. For this run, the features learned on the ImageNet dataset have been leveraged on the provided dataset. In what follows, a detailed description of the architecture and training process of the deep learning model that has achieved the best accuracy on the validation dataset is provided.

##### 4.2.2.1 Best deep learning model

The deep learning model that has achieved the best accuracy on the validation dataset is the XCeption neural network (pre-trained on the ImageNet dataset) appended with three neural network layers: a global average pooling layer, a dropout layer, a densely connected neural network layer.

The model has been trained twice. During the first training, only the newly added densely connected neural network layer's weights were updated. The weights of the XCeption model were frozen so as not to destroy any of the information they contained. The model reached convergence at the 50th epoch with a 90% accuracy on the validation dataset. Adam was used as optimiser with a learning rate of 0.01, and the weights were updated following the gradient of the Sparse Categorical Cross-Entropy loss function. To squeeze the performance of the model a little more, fine-tuning has been carried out as well. The entire model was unfrozen and re-trained with a very low learning rate (0.00001) to avoid overfitting. The performance of the model has reached 94% accuracy on the validation dataset. Then, this model has been used to annotate the images on the test dataset.