Supervised Learning Exam Report

Filippo Monaco
Università degli Studi di Milano-Bicocca
Id: 840089

Marco Picione
Università degli Studi di Milano-Bicocca
Id: 827116

***Abstract*—The purpose of this project report is to give an overview of two different models for a classification task: a traditional classifier, and a convolutional neural network. We will showcase how we handled the input data for the two different models, and the different parameters we tuned to push each model to its limit. Finally, we evaluate both models on a test set to rate their performance in terms of accuracy.**

Link to the Colab: https://colab.research.google.com/drive/1hYdeI9Vw5QSoDyLbam4vETHc3qZxYddj?usp=sharing

1. Introduction

Image classification is a computer vision task that aims to assign a single label to an image: the problem may involve two classes (binary classification) or more (multi-class classification), as in the case here reported.
This project aims to compare two different supervised learning algorithms: Visual Bag of Words, which tries to mimic the natural language process, and a convolutional neural network, which takes inspiration from natural vision processing. The two algorithms were implemented and evaluated on a common dataset, using metrics such as accuracy and mean average accuracy.
The difference between the two algorithms is that the Visual Bag of Words one only provides a representation of the images and thus must work in pair with a traditional classifier while the convolutional neural network is able to extract general features in the first convolutional layers, procedure that can be optimized. This dissimilarity is the reason of the high accuracy discrepancy of the two methods.
This report is structured as follows: in section 2 the dataset and the transformations applied to it are discussed; in section 3 the Visual Bag of Words and its implementation are described; in section 4 the convolutional neural network is explored in details.

2. DataPreprocessing

Before classifying the images with the cited algorithms, some preprocessing is mandatory in order to properly set the dimension of the images and apply normalization. This section is structured as follows: in the subsection 2.1 the provided data are properly described while in the sections 2.2 and 2.3 the preprocessings applied for each algorithm is treated.

2.1. Dataset

The provided dataset is a variant of the "TinyImageNet" which consists of one hundred classes each composed of exactly one thousand 64×64 images properly labelled. Even if the provided data was already divided in the three subsets (training, validation, and testing), the test set was ignored and was properly extracted from the validation one. This was purposefully done so that we had some ground-truth labels for the test set as well.

In particular, randomly sampling was performed to pick two exclusive subsets: the test one containing the 80% of the original images for training, and the remaining 20% used for validation. This way, we ended up with a training test containing 80000 images, 20000 for validation, and 10000 for testing. Figure 1 shows some examples of the provided images with the corresponding label.

2.2. Data preprocessing for Bag of Words

Depending on the algorithm, data was preprocessed in a different way. In the case of Bag of Words, the original image dimensions of 64×64 are kept, and the images are only transformed in gray scale, since keypoint extraction algorithms are basically color-independent (so gray-scale means faster computation, with little to no loss in performance).

2.3. Data preprocessing for Neural Network

As will be properly described in section 4, the implemented neural network exploits transfer learning. Since we started from the pre-trained model $efficientnet\_b0$, we had to apply some transformations on our dataset to be able to properly input them into the network. These will be explained later in section 4.1.

In addition, as discussed in the section 4.3 some classes presented a relatively low accuracy when classified with the convolutional network. To tackle the problem some augmentation of those classes was performed. In particular, we augmented the images associated to class with label 56

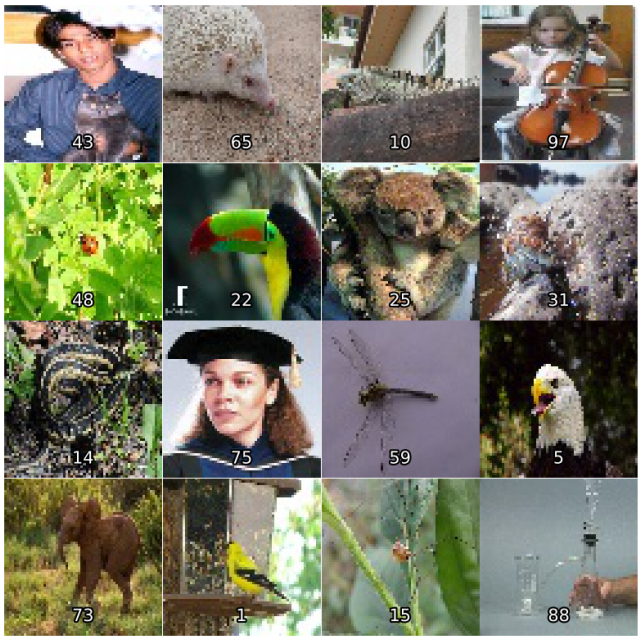


Figure 1: Examples of dataset images and corresponding labels.

(which were images of Cockroaches), creating new images from the starting ones by applying the following transformations:

- Random horizontal or vertical flip (or both)
- Random brightness variation
- Random rotation of $\pm 45°$

Even if the augmentation was performed, the new images were used only at the end in order to see if it was possible to increase the accuracy of that class. Thus the following results were obtained only considering the default images unless specified.

3. BOW

Visual Bag of Words, together with a traditional classifier, is one of the two classification approaches proposed in this work.
This section is structured as follows: in the subsection 3.1 an introduction to Visual Bag of Words is provided, in the section 3.2 the feature extraction is described, follows 3.3 in which the construction of the visual vocabulary is delineated, in the section 3.4 the quantization of local features and the representation of images is outlined and finally, in the section 3.5, the obtained result for this first algorithm are reported.

3.1. Bag of Words walkthrough

Visual Bag of Words algorithm tries to mimic the natural language process dividing each image in smaller ones containing visual features: looking at their frequencies we are able to understand which is the content of our image. A standard classifier was also used in order to classify the images according to the BOW representation. The Visual Bag of Words algorithm consists in several steps, which are properly described in their relative section:

1) Extracting local features in order to find points of interest inside the images
2) Keypoints clustering. The aim of this step is to contruct a visual vocabulary
3) Quantization of local features using the constructed vocabulary
4) Representation of images by frequencies of visual words

3.2. Extracting local features

The first step of the Bag of Words is the extraction of local features. In the work here reported was used, along the SIFT methods, the keypoint extraction which consists in spotting inside the images some points of interest that should remain unaltered under affine transformations, illumination changes and variations in attributes relative to image's quality.

In order to extract them, the choice falled on the SIFT algorithm due to it's computation efficiency (can be used in real time applications) and robustness to changes of scale, rotation, illumination and viewpoint. Follows a brief explanation of the SIFT algorithm. In order to achieve scale inavariance, keypoints are spotted looking for variations at different scales in the so called "Difference of gaussian" that are images obtained subtracting two by two images inside the same blur octave. Involving a threshold and computing the gradient of the found difference it's possible to distinguish between edges and corners: the first carry less information and thus are discarded. The rotation invariance is obtained by assigning to each keypoint one or more orientations in order to allign them: looking at the gradient of the Difference of gaussian is possible to properly construct an histogram of orientation and thus choose the ones with the higher magnitudes. A relevant passage is the robustness to illumination because is achieved by dividing the image in a 4×4 grid each described by an histogram of oriented illumination gradients: each keypoint is thus described by 16×8 (128) features.

The extraction of local features was thus performed applying the SIFT algorithm from the open source library "OpenCV" on all the images in the dataset: their keypoints and descriptors (encoded as one dimensional arrays of 128 elements) were obtained. Among all training images 2951216 keypoints where extracted (36.89 on average). Figure 2 shows some example of spotted keypoints

3.3. Keypoints clustering

The construction of the visual vocabulary is necessary to understand which words describe the training images. This

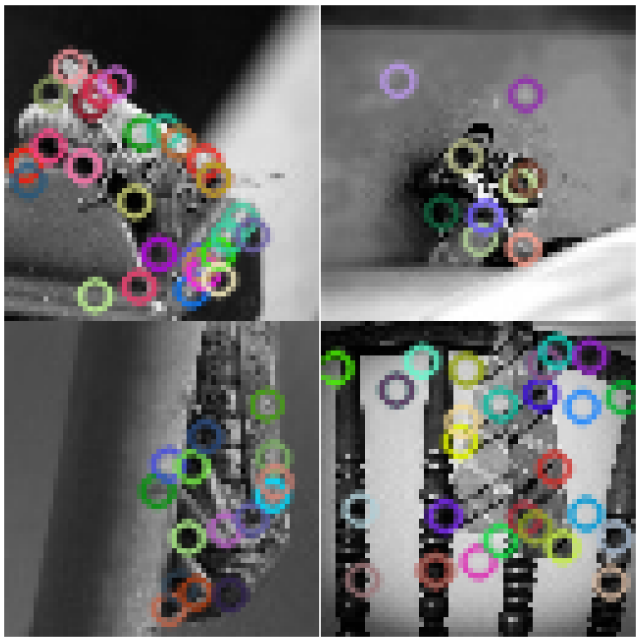


Figure 2: Examples of kepoints inside different images

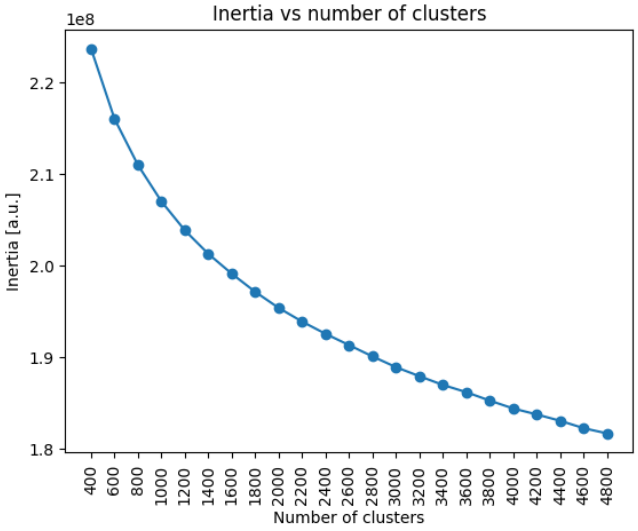


Figure 3: Inertia vs number of clusters

step sees the usage of an unsupervised algorithm in order to cluster all the keypoints inside the 128 dimensional space in a fixed number of words under the assumption that similar elements are different representation of similar words. The set of cluster's centroids constitutes the visual vocabulary.

Due to the high number of training images the choice falled on Mini-Batch K-Means which is a faster (but less accurate) version of the K-Means++ that works on batches instead the whole dataset at the same time. Since we are dealing with a completely unsupervised task, before clustering some preprocess on descriptors was applied in order to (hopefully) increase the clustering accuracy: a standard scaler was applied to each of the 128 features of each keypoint. Both the standard scaler applied and the Mini-Batch K-Means used were taken from the open source library "scikit-learn".

The last step before applying the clustering is to tune the number of cluster which is an hyper parameter defining the number of words that constitutes the visual vocabulary. To do so, the Mini-Batch K-Means was initialized several times with different number of clusters to look for. For each run the inertia of the clustering (sum of distances between each data and it's centroid) was retrieved and was plotted against the set number of clusters. As it's possible to see from the figure 3 no knee was found and thus no conclusion was made.

One possible explanation of this is that there is no such clustering underlying the found keypoints and the found words won't constitute a good visual dictionary. As described in section 3.5 this is probably one of the reasons of such low performance achieved by this first algorithm.

Again, due the the high number of training data and time constraints (clustering time) no other clustering algorithm was available. The number of words was arbitrarily set to 3000 and the visual dictionary was constructed.

3.4. Quantization of local features and representation of images by frequencies of visual words

After constructing the visual vocabulary, the following steps are the quantization of local features and representation of images by frequencies of visual words i.e. understand which words compose every image.
For each element of the training set a histogram with number of bins corresponding to the number of classes was constructed. In particular, extracting the keypoints, it's possible to associate each one of them to a specific word according to the visual vocabulary that makes use of their descriptors. This was simply performed associating each keypoint to it's closest word (cluster centre) according the the Mini-Batch K-Means model previously fitted. Since the height of each bin strongly depends on the number of keypoints found, all the obtained histograms were normalized dividing the content of each bin for the number of words found inside the image. This choice is motivated by the fact that is better to describe each image through the relative words' frequencies instead of directly using their bins' heights. The figure 4 shows an example of two images together with their extracted keypoints and their corresponding histograms.

3.5. Results

Visual Bag of Words algorithm provides a description of the images which was exploited in order to train a traditional classifier. In order to achieve the highest possible accuracy on the test set, different traditional classifiers were trained providing, for each training image,

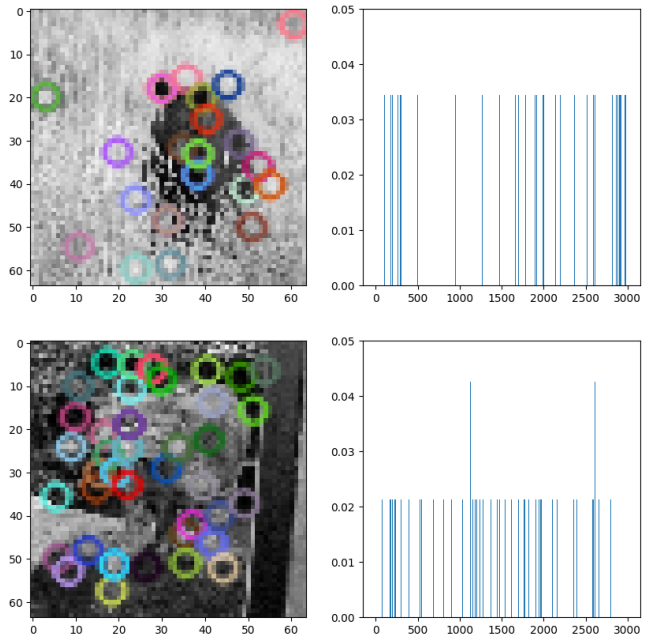


Figure 4: Example histograms associated to different images. Note that the heights of the bars are not between 0 and 1 only for sake of comprehensibility of the figure.

the generated histogram and their label. All the models reported here are taken from the open source library "scikit-learn". The tested models and their parameters are reported in the following list:

- $k-NearestNeighborsClassifier$ (kNC), with 10 as number of neighbors
- $StochasticGradientDescentClassifier$ (SGDC), with loss set to "log_loss" and "l1" penality
- $SupportVectorClassifier$ (rSVC), with radial basis function kernel
- $DecisionTree$ (DecTree), with Gini index
- $RandomForest$ (RandFor), with 300 as number of estimators
- $AdaBoost$ (Ada), with 500 as number of estimators
- $Bagging$ (Bag), with 10 as number of estimators

The obtained accuracies are summarized in the following table:

Classifier	Accuracy (Test)
kNC	1.310 %
SGDC	4.671 %
rSVC	6.791 %
DecTree	2.010 %
RandFor	3.690 %
Ada	2.590 %
Bag	2.260 %

As one can clearly see, no classifier is able to reach a desirable accuracy. There are several possible reasons of such low performance:

- The searched clusters doesn't exist. As already anticipated in the section 3.3, the absence of a knee in the inertia curve while constructing the visual vocabulary may indicate that the chosen clustering solution is not optimal. The construction of an inadequate visual vocabulary may of course lead to not representative histograms and thus to an ineffective classifier.
- Training the classifier on the histograms constructed with the same dataset used to build the visual vocabulary may lead to overfitting.

Out of all the classifiers, however, we can see that the Support Vector Classifier with a radial basis function kernel performs best. When we look at the time it took to train and test the different classifiers, however, we can see that having higher accuracy is not always the only useful metric for choosing a model.

Classifier	Time (Train/Test)
kNC	00h:00m:11s 00h:00m:35s
SGD	00h:04m:14s 00h:00m:01s
rSVC	04h:09m:52s 01h:36m:35s
DecTree	00h:05m:26s 00h:00m:01s
RandFor	00h:10m:49s 00h:00m:31s
Ada	00h:03m:40s 00h:00m:05s
Bag	00h:58m:11s 00h:00m:05s

Clearly, from this table we can see that the Support Vector Classifier takes much much longer than all the other models combined, both in Training and Testing times.

We can conclude that while SVC has a slightly better accuracy (around 7%) than the other models, the amount of time it takes for it to complete both training and testing means that it is not at all a good choice. Burn it.

4. CNN

The Convoutional Neural Network is the second method proposed in this project. This section is structured as follows: in the first subsection 4.1 a quick overview of the workings of the network is given, while in the following sections 4.2, and 4.3 we will dive into more details on how the network performs. In section 4.4 we compare different pre-trained models, and finally, in section 4.5, we will provide an outline on the hyper-parameters we decided to change to obtain a better classifier.

4.1. Convoutional Neural Network Walkthrough

Convolutional Neural Networks are biologically-inspired architectures broadly used in computer visions tasks, such as image classification or object detection. Intuitively, they stack multiple feature extractors as neural layers, where the deeper ones compute more global and invariant features. For example, the first layers may try to extract features relative to a specific part of the image (remember the concept of local connectivity), such as difference in intensity between neighboring pixels, while higher stages may focus on high-level features, such as entire objects segmentation.

For the purpose of our project, we decided to utilize what is called a "Transfer Learning" approach. In short, this means that we used a pre-trained model, changed the output classifier in order to adapt it to our specific problem, and only trained this last addition on our data-set.

The entire process consists of:

1) Choosing a pre-trained model:
   in our case we chose the $efficientnet\_b0$ model, since it gave great results in a previous lab session in terms of accuracy, while still having a low number of parameters (around 5.3 millions). Since it was trained on the ImageNet dataset, and ours was a subsection of it (TinyImageNet), we didn't need to do anything too complex to use it for our problem.
2) Changing the output classifier:
   since the original model was trained to predict 1000 classes, we had to change its classifier, since we are only interested in 100. We decided to add a small Multi-Layer Perceptron classifier composed of two layers with 512 and 256 neurons respectively.
3) Freezing the main model:
   we only needed to train the newly added classifier, since the new dataset is large and similar to the original one (it being a smaller version of it). In order to accomplish this, we just froze the parameters of all neurons that are not in the classifier so that the backpropagation algorithm does not reach them.
4) Data pre-processing:
   compared to the BoW approach, we needed to handle the data differently for optimal network input. The transformations we applied to each image were the following:
   - Resizing the image (256×256 pixel for $efficientnet\_b0$)
   - Center-cropping the image (224×224 pixel for $efficientnet\_b0$)
   - Converting the image to Tensor (for CUDA input)
   - Normalization of the image, using mean
     $$[0.485, 0.456, 0.406]$$
     and standard deviation
     $$[0.229, 0.224, 0.225]$$
     for each channel respectively. This is a standard technique in the literature that has shown to be effective for training basically any model.
   - Batch the images for faster training (64 images per batch for $efficientnet\_b0$)

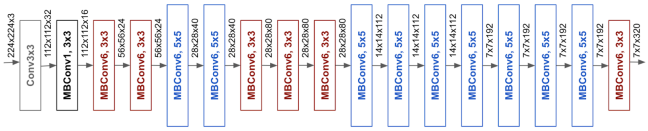


Figure 5: Architecture of EfficientNet_b0

Now our network is ready to be trained on the new dataset.

4.2. Training and Validation

In order to train our model for classification, we need to:

1) Feed it input images with the corresponding labels
2) Try to predict the label of the current image
3) Compute the loss, in terms of difference between predicted and ground truth label
4) Backpropagate the error through the network, updating the model parameters
5) Validate the model on a subset of data to test its performance in terms of accuracy
6) Repeat the process until the accuracy on validation remains the same or gets worse
7) Save the model with best accuracy

In our case, we used the following hyper-parameters for training:

- Number of Epochs: 10 (number of training epochs)
- Patience: 3 (max number of iterations with no improvement, after which we stopped the training early)
- Loss function: $CrossEntropyLoss$ (widely used for multi-class classification problems, since it considers the logits outputs of the network as a probability distribution over the different classes, and compares them as a whole to the ground truth label).
- Optimizer: $Adam$ (a variant of the Stochastic Gradient Descent optimizer, used for computing Backpropagation through the network)
- Learning Rate: 0.001 (learning rate used by the optimizer to regulate how much to tune the model parameters after each batch)
- Scheduler: $CosineAnnealingLR$ (used to dynamically reduce the learning rate over the course of the training process)

The entire process took about 1 hour and 15 minutes between Training (average of 9min, 40s per epoch) and

Validation (average of 1min, 10s per epoch). Note that Validation is much faster both because of the absence of the backpropagation process, but also because we decided to increase the Batch-size for this phase four fold.

In figure 6 we can see how the Training and Validation loss decreases in subsequent epochs. Note that the model does not reach the 10 training epochs, since it lost patience and it stopped early, after 3 epochs with no increase in validation loss. We can also see that during this time (where Validation loss increases), Training loss keeps decreasing: this is a clear example of overfitting.

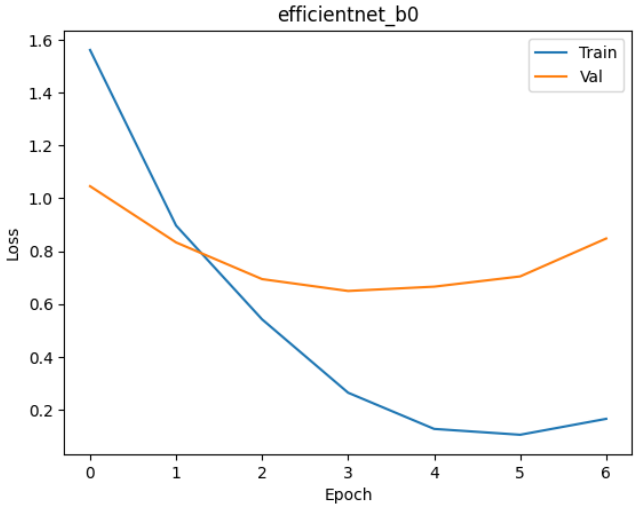


Figure 6: Loss curves for training and validation dataset

We thus save the model with the best Validation loss, which in this case corresponds to the one in epoch 4.

4.3. Testing

After having trained the network, we tested it by feeding it the testing images and seeing the predicted label. We kept track of the True Positives, False Positives, and False Negatives across the images.

The evaluation metric we used is accuracy, defined as the number of correctly predicted labels over the total number of labels, and divided in:

- Accuracy per class, by computing the average accuracy for each class
- mean Average Accuracy, by computing the average accuracy separately for each class, and then averaging over all classes.

In figure 7 we can see the average accuracy for each class reached by $efficientnet\_b0$ on the testing dataset. Note that the mean Average Accuracy as well as the lowest and highest performing classes are highlighted.

We can see that the model performs really well, reaching a mean Average Accuracy of 0.841.

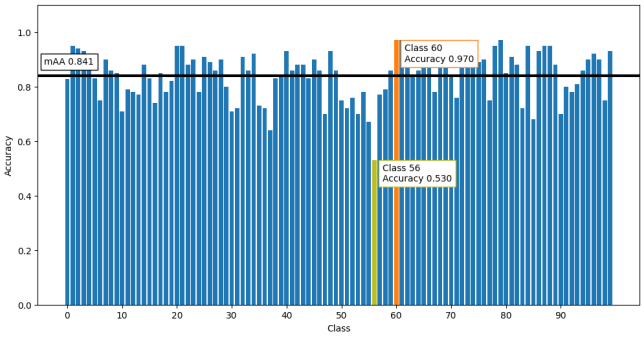


Figure 7: Accuracy of $efficientnet\_b0$

From 7 we can see that one class performs much worse compared to others: class 56, with an average accuracy of 0.530. We decided to try some data augmentation for this specific class, as previously mentioned in section 2.3.

Repeating the entire training and testing process with this new dataset, we got the accuracy plot shown in figure 8 As we can see from the plot, class 56 did indeed see a jump

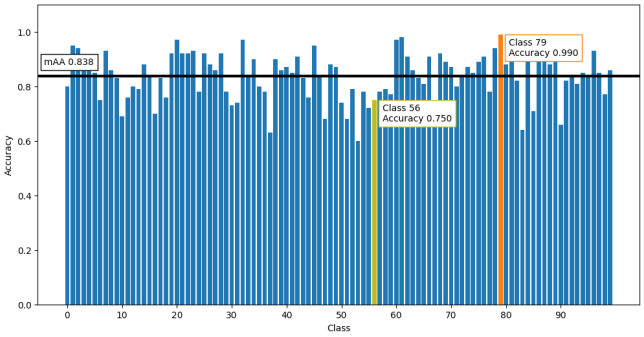


Figure 8: Accuracy of $efficientnet\_b0$ on the Augmented dataset

in average accuracy, meaning that our data augmentation worked. On the other hand, the mean Average Precision did not change much (actually, it got lowered a bit), probably meaning that the network, focusing more on the augmented class, lost some accuracy in the other classes.

4.4. Different Models

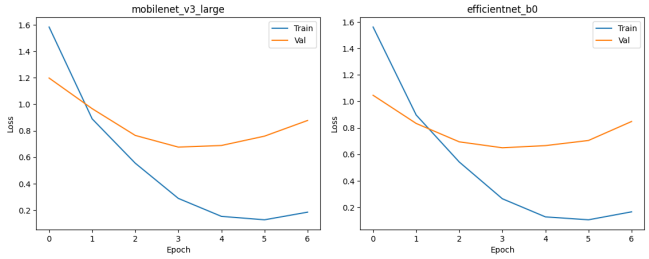
Before diving into the actual hyperparameters tuning process, we wanted to compare the difference between different Neural Network Architectures.

The model architectures we tested are:

- $mobilenet\_v3\_large$: the first one we tested, since it was very light (only $5.5M$ parameters), while still having acceptable accuracy (around 74%)
- $efficientnet\_b0$: the ones we ended up using, since it has great accuracy (77%), while having a very low number of parameters ($5.3M$).

- $vgg16\_bn$: an older model we tried, just to see how it compared to newer ones; unfortunately, due to its very high number of parameters (around $138M$) and lower accuracy (73%), we decided to discard it. Another issue was the insane amount of time it took for training (around 2 hours per training epoch).

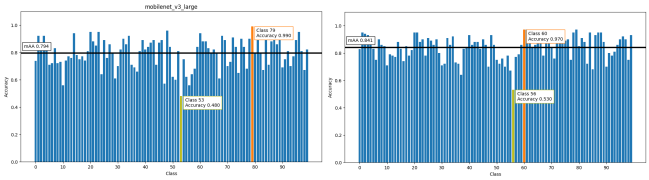
For the first two models listed above, we decided to compare their Training and Validation Loss, as seen in figure 9, which shows that both models followed pretty much the same process: Validation loss decreases for the first 3 epochs, after which there is a period of overfitting and subsequent early stopping of the training process.



(a) (b)

Figure 9: Training and Validation Loss

Additionally, in figure 10, we can see the difference in terms of accuracy between the two models. We can clearly see that $efficientnet\_b0$ reaches a higher mean Average Accuracy of 0.841, while $mobilenet\_v3\_large$ reaches a considerably lower value of 0.794.



(a) (b)

Figure 10: Testing Accuracy

Following these comparison, we decided to keep using $efficientnet\_b0$, for the above-stated reasons.

4.5. Hyperparameters Tuning

An important process in any learning algorithm is Hyperparameters Tuning: the problem of finding an optimal set of hyperparameters to run the model with. The parameters to tune can be numerous, and often it is not feasible to think to try all combinations of them. Some examples of hyperparameters are the number of hidden layers of a neural network, the number of neurons in each layer, the type of optimizer used, the learning rate, the batch size, etc. Even if we wanted to try a couple of values for each parameter, it would take a very long time to compute all possible combinations.

To tackle this issue, we tried using the Optuna Python library, an automatic hyperparameter optimization software framework which exploits the parallelization of the training process to test different hyperparameters sets at the same time. It works by running a number trials with different hyperparameter values, and choosing the best ones. We decided to run the following configuration of the framework:

- Sampler: $TPESampler$, which samples a set of parameters randomly at each run, but also remembers the history of past hyperparameters which it uses to suggest future set of values.
- Direction: $minimize$, since we want to find the set of hyperparameters values that minimizes the Validation Loss.
- Number of Trials: 30, since we want to try a reasonable amount of hyperparameters values before deciding on the best set.

With this configuration, we tried to tune the following different parameters:

1) Classifier:
   - Number of Neurons in first hidden layer: between 512 and 896
   - Number of Neurons in second hidden layer: between 128 and 512
   - Dropout chance after each layer: between 0.0 and 1.0
2) Training process:
   - Training Epochs: between 5 and 10
   - Learning Rate: between 0.0001 and 0.1
   - Optimizer: $Adam$, $RMSprop$, or SGD algorithms

Unfortunately, we were not able to run any of the above code, due to having numerous issues with Colab's CUDA. The reported errors were generic and gave little hints on how to solve them. We tried debugging the problem over and over, asking the professor and searching everywhere online for solutions, but we were not able to solve the problem.

We think that it was still worth it to put this part in the report, since it should theoretically work, and a certain set of hyperparameters would come out on top, in terms of lowest Validation Loss. Unfortunately, we are not able to actually test it.

However, we tried to manually change different parameters to see how they each effected the model performance. Due to the amount of time this took, however, we don't have enough data to provide a clear and conclusive conclusion (in terms of hyperparameters). Thus, we just report the hyperparameters we used for our best model:

- Model: $efficientnet\_b0$
- Hidden Layers in Classifier: 2
- Neurons in first Hidden Layer: 512
- Neurons in second Hidden Layer: 256

- Dropout chance: 0.1
- BatchSize: 64 for Training, 256 for Validation
- Num of Epochs: 10
- Learning Rate: 0.001
- Optimizer: $Adam$
- Learning Rate Scheduler: $CosineAnnealing$, with T_max as $\#TrainingBatches * \#Epochs/2$

5. Conclusion

As we expected, the Convolutional Neural network works much much better compared to the traditional classifier, both in terms of overall accuracy, and training times.

While we did not exhaust every possible optimization possibility (hyperparameters tuning, for example), we can confidently say that we are proud of this project, and that we learned a lot from it.

And them's the facts.

6. Take home message

Figure 11: Actual image of me and Marco during the many training sessions (ca. 2023, colorized, then decolorized again).

Acknowledgments

The authors would like to thank themselves for the hard work done.

References

[1] PyTorch: machine learning framework for Python [https://pytorch.org/vision/stable/models.html]

[2] scikit-learn: tools for predictive data analysis [https://scikit-learn.org/stable/modules/classes.html]

[3] Optuna: automatic hyperparameter optimization software framework https://optuna.readthedocs.io/en/stable/]

[4] Out Past Lab Sessions: much of the code used in this project was taken (and obviously re-adapted) from our past weekly assignments [https://drive.google.com/drive/folders/1cjYkmQqQOqEh25NsYcwi2EQarrK3AXtJ?usp=sharing]

We confirm that this report is entirely original and does not contain any form of plagiarism. We did not make us of ChatGPT or any other Natural Language Processing models.