UCLouvain

École polytechnique de Louvain

Deep learning-based image classification for autonomous vehicles and training by Unreal Engine.

Author : **Celine Nardi**
Supervisor : **Benoît Macq**
Readers : **Siegfried Nijssen**, **Jonathan Samelson**, **Victorien Sonneville**
Academic year 2021–2022
Master [120] in Computer Science

Acknowledgements

I would like to thank the following people without which this work would not have been possible.

- My supervisor Benoît Macq for his patience and support.
- Jonathan Samelson for the dataset, his availability, support, patience and precious advice.
- The whole team of the project for this great opportunity.
- Finally, my family and close ones for their love and care.

Content

Introduction

Nowadays, there are many reasons to use a game engine other than creating a video game. One of those reason is becoming quite popular: creating a digital twin meant to be used in applying machine learning models.

This master thesis falls within such a project. Our project is to create digital twins of urban scenes. Here we focus on a crossroad with traffic lights. That is interesting in many ways: from the improvement of detection of the cameras at traffic lights, to automatic labelling of satellite-like images. This master thesis focusses on a specific aspect of the project: allowing vehicle in the digital twin to move in an autonomous way, and more precisely to make the decision of braking or not, solely based on driver's frontal point of view, speed and steering angle.

In this digital twin, the world is simpler than the real world. Firstly, vehicles follow a predefined track and do not need to make direction-related decisions. Secondly, speed limit exists but is the same everywhere. Thirdly, vehicles go straight in the intersection, they do not turn. Finally, if the road has two lanes, vehicles do not change lane. Therefore, the only big decision that has to be made is how to adjust the speed, which we have simplified as a binary decision: braking or not.

The research objective of this master thesis is then to develop a model able to predict effectively whether or not a vehicle needs to brake based on the driver's frontal point of view, speed and steering angle. As speed limit is constant, it was not a parameter worth adding if speed is already considered because it is indirectly taken into account by the speed. In order to reach that objective, we have a dataset built from the digital twin of a crossroad with traffic lights in the game engine *Unreal Engine 4*. The labelled vehicles of the dataset make decision in the crossroad based on built-in decision trees in the game engine. The dataset construction is further discussed in the dedicated section.

Because this work falls into a context simpler than reality, it might be interesting to see if a good performing model can be built only using the frontal point of view, speed and steering angle. If a sufficiently effective model can be built in that sense, it would be a first step toward saving on sensors and democratization of autonomous systems.

In the following, we break down our research topic into five fields - image classification, self-driving cars, classification based on multiple types of data, and data class

imbalance – and review literature and related work in each of these fields. Taking inspiration from the literature, we develop the action plan of our research that consists of three steps. The first step is comparing the performance of four convolutional neural networks – VGG16, VGG19, ResNet50 and Xception - with solely the driver's frontal point of view as input. The second step is comparing the same models improved by adding numerical inputs, i.e. speed and steering angle. The third and final step is to boost the prediction performance by aggregating the predictions of the four models of the second step with ensemble techniques. We then describe the technology used. Afterwards, there is a theory section that comes back on all the main specific concepts that are used during the research.

After the theory section, we further describe the construction of the dataset and explore it. Then for each step of the action plan, we discuss modelling, results and we draw conclusions.

Literature and Related work

The research topic of this master thesis can be broken down into four fields: image classification, self-driving cars, multi-input classification, and class imbalance.

Before going any deeper, it might be useful to get back to the basics and quickly review the concepts of artificial intelligence, machine learning and deep learning.

Artificial intelligence (AI) generally refers to algorithms that solve problems that would normally need human intelligence to be solved. Machine learning refers to the subset of those algorithms that can learn without that learning being explicitly programmed. In contrast to traditional machine learning approaches, deep learning refers to the subset of machine learning algorithms that do not need manual intervention to detect relevant features.[1, 2] Deep learning networks try to emulate human intelligence and are capable of learning from a very large dataset [3]. One big advantage of deep learning models is portability. If they are trained on a sufficiently large and diverse dataset, they are easily repurposable to other problems with new classes. [2]

There are two main learning approaches in machine learning: supervised learning where a labelled dataset is used for training, and unsupervised learning where a non-labelled dataset is used for training [1]. In this work, we use supervised deep learning.

Image classification

Image classification has been extensively studied in literature and it turns out that convolutional neural networks (CNN) are outstandingly effective for that task in a lot of different domains such as tumor diagnosis [1], developing a harvesting robot able to detect fruits [4], COVID19 detection via X-ray images [5], or enhancing autonomy of an endovascular surgery robot [6]. We can see an illustration of that in a paper of Sunil et al. (2022) [7] on image classification to identify plant species where classification of VGG16 architecture has outperformed all Support Vector Machine (SVM) classifiers tested with an average f1-scores between 93 and 97%, about 10% higher than the scores of SVM classifiers.

Rashmi et al. (2022) [1] describe a multi-view classification approach to detect a tumor. The idea is to improve accuracy by combining data from different perspectives into one uniform representation. That approach increases the classification reliability [8] and resilience. Multi-view data includes multiple samples from the same target but with different perspective, such as pictures from a same object but taken with different angles [8]. The multiple

perspectives can be merged at different stages of the network (early fusion) or at the classification step (late fusion).

Another concept very close to multi-view classification, is ensemble learning where the different perspectives come from different CNN architectures that derive different features from the same image [5].

Kong et al. (2022) [5] successfully use this approach to detect COVID19 via X-ray images. They use a dense convolutional network (DenseNet) and a visual geometry group network (VGG16) that receive an input preprocessed by a residual network (ResNet). The models are fine-tuned to extract features that are then merged and given as input to an ensemble learning classifier including a multi-layer perceptron (MLP). The average accuracy for this binary image classification task is 98%.

It is not uncommon to combine multi-view classification and ensemble learning to boost the performance, as those are close concepts. [9]

Guerin et al. (2021) [9] successfully use both techniques for an image clustering task, which is similar to image classification but with unsupervised learning. They combine at the same time different angles for the same subject and different CNNs (VGG16, VGG19, ResNet50, InceptionV3 et Xception) to extract features that are used as input in MLP classifiers constituting the multi-view clustering model. The results indicate that the best layer to extract features is the highest level of a pretrained network, i.e. the one just before the prediction layer, because it is the most specific to the task and the most complex.

We have previously established that CNNs perform better than other models in the context of computer vision but the choice of the CNN architecture is crucial for the final results and it is a hard task [9]. For instance, Song et al. (2019) [4] conclude that VGG16 performs better than another CNN, i.e. ZFNet. Combining CNNs allows to reduce the importance of that choice. Information contained in different CNNs can be complementary even if their input is the same [9]. To recognize a car, one CNN maybe looks at the wheels, while another looks at the windows. Both can recognize a car but together they are more resilient.

Ensemble learning is truly considered as quite effective in the literature. What better prove that most of the recent data science competition winners have used ensemble techniques such as the winners of SIIM-FISABIO-RSNA COVID-19 Detection 2021, RANZCR CLiP - Catheter and Line Position Challenge 2021 and Cassava Leaf Disease Classification 2021 on Kaggle [10]. The first one has used ensemble learning on four different CNNs. The second one

has done a twofold classification: a first ensemble with 20 models – a mixture of 3 different architectures – and a second ensemble with 31 models - a mix of many different architectures. The third one has combined four models using stacking and average voting with class probabilities. They all have reached accuracies beyond 90% [10].

Ensemble learning is not infallible however and does not work in all cases as pointed out by the winner of RSNA-MICCAI Brain Tumor Radiogenomic Classification challenge 2021 on Kaggle who owes his first place to good preprocessing choice and training samples selection [10].

Another popular concept that has been used in many works is the K-fold cross validation. It is used to adjust the volatility and stability of the results [5]. The overall performance of the model is reported by average the performances on each fold. K can take different values but five is quite common [5, 10, 11]

Self-driving cars

Autonomous decision in self-driving cars has been extensively covered in the literature and mostly focus on object detection, scene perception and vehicle motion control. The AI used in self-driving vehicles have visual recognition system that encompasses image classification, object detection, segmentation and localization [12]. Research in the field mostly uses a combination of sensors, cameras and other perception tool to develop a decision-making model. We explain in the next section how most of the research around autonomous vehicle is outside of the scope of this master thesis.

Self-driving cars in the real world are considered as intelligent agents capable of building a tri-dimensional representation of their environment by aggregating data provided by various sensors, cameras, RADARs, LIDARs, etc. [3, 12, 13]. The self-driving cars in our work match the definition but the tri-dimensional representation is indirect, and the sensors limited.

CNNs have also been used for different tasks in the context of self-driving cars, for object detection -such as pedestrian detection – and distance estimation for instance [12, 14].

For example, Chen et. al (2022) [13] use CNNs to build a spatial-temporal graph convolution network (STGCN) to predict traffic flow in road network and then build a Digital Twin prediction model.

For pedestrian detection, Li et al. (2020) [15] use their own CNN combined with an AdaBoost classifier based on SVMs, decision trees, and random forests. They reach an

accuracy of about 96%. Wu et al. 2020 [16] has another approach to the task. They convert LIDAR point cloud data to pixels that serve as input to a SVM that does a first classification of the dominant class so that class imbalance is reduced. Then the remaining unclassified data are used as input in a CNN.

For road-marking detection, Hu et al. (2019) [17] modify the VGG16 base network of faster R-CNN algorithm using a data fusion technique, in this case by concatenating all the convolutional layers within a stage to extract image features and have most of the features ever extracted by VGG16 in one feature map. Based on the feature map, a R-CNN identify the regions of interest.

For traffic light detection, Mu et al. (2015) [18] do not use a CNN but use ensemble learning with SVMs and Histogram of Oriented Gradients (HOG). They use a camera-based algorithm that first converts RGB color space in Hue-saturation-value (HSV) color space, then find areas where there might be a traffic light and finally detect the state of the traffic light with a HOGs and SVMs.

Two other methods of deep learning models that are often used for autonomous vehicles are Long Short-Term Memory (LSTM) and et Reinforcement Learning (RL). LSTM is a neural networks architecture mostly used in motion planning, decision making and vehicle control in order to predict the best next action based on past actions. RL consists of developing an agent that learns from its error and experiences. [3]

Multi-input Classification

The multi-input classification problem refers to training a predictive model with multiple inputs.

First of all, everything that has been said about ensemble learning actually applies to the multi-input classification. Indeed, if several models extract parallel features of a same input, they each get a copy of the input. Therefore, there are as many inputs as models.

The case of interest here is the case where the multiple inputs are different, i.e. a multi-modal classification task. The process is very similar to identical inputs, as shown by Zhao et al. (2019) [6], Chen et al. (2022) [19] and Apostolopoulos et al. (2021) [20].

In order to deal with multiple inputs for the development of a surgery robot, Zhao et al. (2019) [6] use a CNN-base framework composed of a multi-input CNN with one dimension that takes the data on operation force obtained from sensors, and a two-dimensional CNN with

a visual input obtained from a camera. The two CNNs are combined thanks to a collaborative algorithm that takes the best decision based on the outputs of both CNNs. They have good results in terms of improving the robot's autonomy and being generalizable to other surgical robots.

Chen et al. (2022) [19] in the context of fault detection for photovoltaic system, use the same CNN architecture for two different inputs: time-domain and frequency-domain. The CNNs' outputs are then combined and classified using a variant of an MLP where the *dense* layers are replaced by *Global Average Pooling* layers in order to reduce the risk of overfitting.

Apostolopoulos et al. (2021) [20], for a classification task in the field of cardiovascular disease diagnosis, have two different types of inputs: some are non-visual and train an MLP or a random forest, while the other type of inputs is visual, which is used to train the CNN model InceptionV3. The features extracted by InceptionV3 are then combined with the predictions of the MLP and the random forest. They reach an accuracy of 79.15%, which matches the expert's accuracy.

Class imbalance

The class imbalance problem has been extensively studied. There is class imbalance when one of the classes that are predicted is under-represented in the dataset. It is often the under-represented class that is the most important one. For instance, the presence of a tumor is less frequent than the presence of a healthy body. It is then crucial that the under-represented class is correctly identified. There exists a myriad of strategies to address the class imbalance problem. Those strategies can mostly be classified into two categories: data-oriented approaches which aim at modifying the dataset before training to restore balance, and algorithm-oriented approaches which adapt the training process to take into account the difference in representation. [21, 22]

The data-oriented approach encompasses all the resampling strategies that alter the dataset to make it balanced. Those strategies include oversampling that either duplicates or simulates sample of the under-represented class, and under-sampling that removes samples from the dominant class. [21, 22, 23]

The algorithm-oriented approach adjusts the learning algorithms to reduce the bias towards the dominant class. That can be done by giving weights to the classes [5], choosing appropriate optimization metrics, choosing the right optimization algorithms and the right loss. Apparently, the *Adam* optimizer, which uses a stochastic gradient descent optimization method

[24], remains the fastest and the best performing compared to other optimizers in a class imbalance context [5, 10]. An example of appropriate loss function is focal loss that down-weights well-classified samples and gives a higher loss value to misclassified samples [10, 25, 26]. Ensemble learning can also be introduced to alleviate the imbalance as it makes the prediction more robust and resilient. [21, 22, 23]

Standard evaluation criteria such as accuracy and error rate, tend to give more impact to well represented classes at the expense of the under-represented classes [23]. Nevertheless, there exist many metrics that suit an imbalanced dataset, such as F-measure which combine precision and recall (also called sensitivity) [7, 13], G-Mean which compute the geometric mean of accuracies of each class, AUC that refers to the area under the receiver operating characteristics (ROC) curve which evaluate the trade-off between true positives (TP) and false positives (FP) (benefits and costs), etc. [23]

Action Plan

In the literature, we have seen that repurposing pretrained CNNs perform great on image classification, as well as ensemble learning. Ensemble learning is also a way of dealing with class imbalance and reducing the importance of choosing the best CNN, which is a hard task. Popular CNN architectures are residual networks (ResNet) and visual geometry group (VGG). Another interesting technique to reduce volatility and increase stability, is the K-fold cross validation algorithm that seems to bring significant added value to model training.

Regarding autonomous vehicles, we could not help but notice that the stakes in the literature go far beyond the scope of this project. The world we work on is much simpler and we have done the choice to not use other data/sensors than the frontal point of view, the speed and the steering angle. In the literature, most research works try to get an explicit tridimensional perception of the environment, control vehicle motion and explicitly detect objects. We do not need that in our task, we simply want to decide whether or not the vehicle should brake.

In order to deal with multi-modal classification, i.e. multiple inputs of different types, it was clear from the literature that the most common way to combine those inputs it to assign a partial or full classification model to each family of input and aggregate their information using another model. The combination of the models can happen at different stages of the classification (early or late fusion). CNNs are generally used to extract features from visual inputs while MLP, random forest, SVM, etc. are used for non-visual inputs and to combine models.

Concerning class imbalance, the algorithm-oriented approach sounds more appealing because it would be hard to ensure keeping diversity of the many urban situations that can happen on the road if we re-sample our dataset, even in our simpler world.

Taking inspiration from the literature, we develop the action plan of our research that consists of three steps. To start, we prepare the dataset and the folds of our K-fold algorithm. The first step is comparing the performance of four convolutional neural networks – VGG16, VGG19, ResNet50 and Xception - with solely the driver's frontal point of view as input. The second step is comparing the same models improved by adding numerical inputs, i.e. speed and steering angle. Both family of inputs are assigned a model – a CNN for the visual input, and an MLP for the numerical inputs – and then the models are combined in another MLP classifier to get final predictions. The third and final step is to boost the prediction performance by aggregating the predictions of the four models of the second step with ensemble techniques.

Regarding the technology, the code that is used in the following is written with *Python3.9* [27] and uses *Keras* [28], a high-level deep learning API written in Python developed by Google running on top of the machine learning platform *TensorFlow* [29]. The interface is approachable and provides essential abstractions and building blocks for developing machine learning solutions with a focus on modern deep learning. *Tensorflow* is a scalable and open-source machine learning platform that efficiently executes low-level tensor operations on CPU, GPU and TPU. [28]

The code is run on a Linux x89_64 machine with twelve Intel® Xeon® CPUs E5-1650 v4 @3.60GHz, 64 GB of RAM and four GPUs GeForce RTX 2080 Ti.

Concepts

This section introduces some of the theoretical concepts that are used in steps 1 to 3 of the action plan.

K-fold cross validation

The K-fold cross-validation is a technique where the dataset is divided into K partitions (cf. Figure 1). The model is then trained with K-1 partitions and tested with the last unused partition. The model validation score is the average of the test scores over the K partitions. [2] This technique is quite popular because it allows to reliably assess the model. If the hyperparameters of a model are tuned for only one test set, there is a risk of overfitting this test set and the model might perform poorly on new data. Researchers in machine learning often face that situation where they obtain outstanding results on the test set but their model does not perform as good once applied on a new dataset. [2, 30]

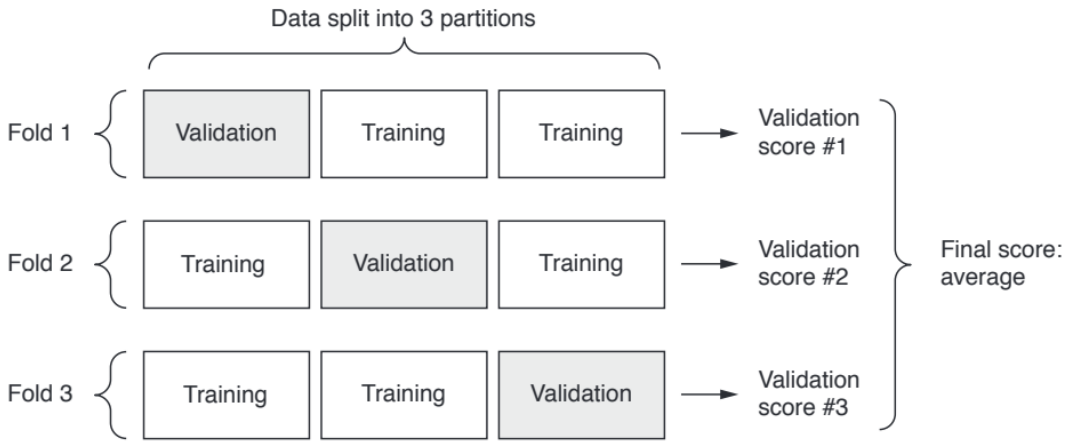

Figure 1: An example of 3-fold cross validation [2].

Convolutional neural network (CNN)

A convolutional neural network (CNN) is generally an artificial neural network trained to interpret visual inputs in deep learning (cf. Figure 2). A CNN derives features, i.e. visual concepts - without human intervention – from the training dataset, which are then used to predict classes [1]. CNNs are actually considered the best tool for image classification [1,2]. They have proven to be outstandingly effective at classifying data based on pixels [6].

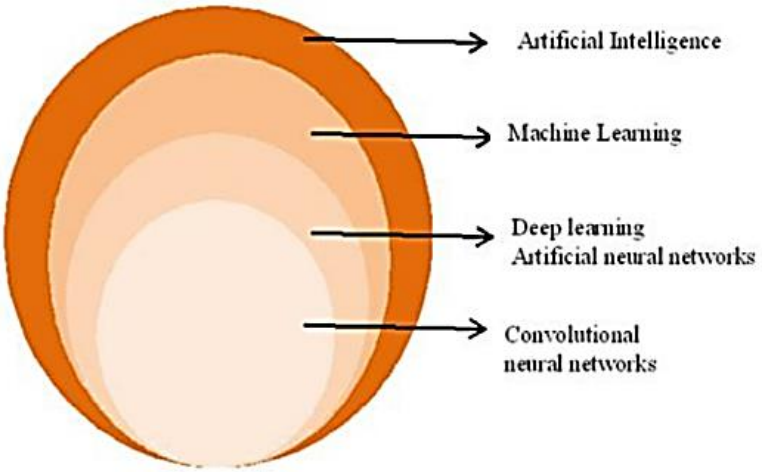

Figure 2: Hierarchical diagram of convolution neural networks [1].

They have two key characteristics. The first one, they learn patterns that are translation invariant. In other words, if a CNN identifies a pattern in the left corner of the picture, it can identify it anywhere on the picture afterwards, even in the middle of the picture for instance. The second one, they learn space hierarchy of those patterns. The first layer learns small local patterns such as edges. The following layers learn increasingly large patterns based on the patterns of the previous layers until they can interpret very abstract visual concepts. Higher the layer is in the hierarchy, bigger is the complexity of the patterns. [1, 2]

A CNN also have two main parts: a convolutional base that consists of pooling and convolution layers, and a top classifier which is a densely connected classifier. Generally, higher the layer is in the model, more specific it is to the classification problem it was trained for. The convolutional base identifies generic concepts, which can be easily applied to other contexts. It also considers information about where are the concepts on the picture whereas the space notion is lost in the top classifier. In the first layers, colors, textures, and edges are identified while in later layers, more specific elements are considered such as rabbit ears or goat eyes. [2]

Deep learning models in computer vision are by nature easily repurposable with minor changes if they have been trained on a sufficiently large and diverse dataset. If the dataset is large enough, the features that are identified are sufficiently generic to be applied to other data from new domains and new classes. Using a pretrained network is, moreover, a very common and effective approach to address an image classification problem because, via the weights acquired with the pretraining, the network already has some knowledge on what features are relevant to extract. A pretrained network is a network that has been previously trained on a

sufficiently large dataset and that can be, therefore, easily re-used with another dataset. [2] The pretrained layers can be used as they are or re-trained. In the second case, pretraining allows to significantly save time because the layers do not start from scratch but already have some knowledge. To draw a parallel, it is quicker to learn how to resolve an equation if you can already count.

Training a neural network is usually done with several iterations (cf. Figure 3). A neural network is composed of layers that each has an activation function, an input and an output. The output of one layer is the input of the following one. An activation function is how the input is transformed into the output, and is parameterized by its weights. The objective of training is actually finding the right weights values of all layers of a network. To find the best weights values, the weights are evaluated by the loss function that compares the predictions with the true labels and compute a distance score (loss score). During the training, each iteration tries to lower that score by adjusting the weights via an optimization algorithm. [2]

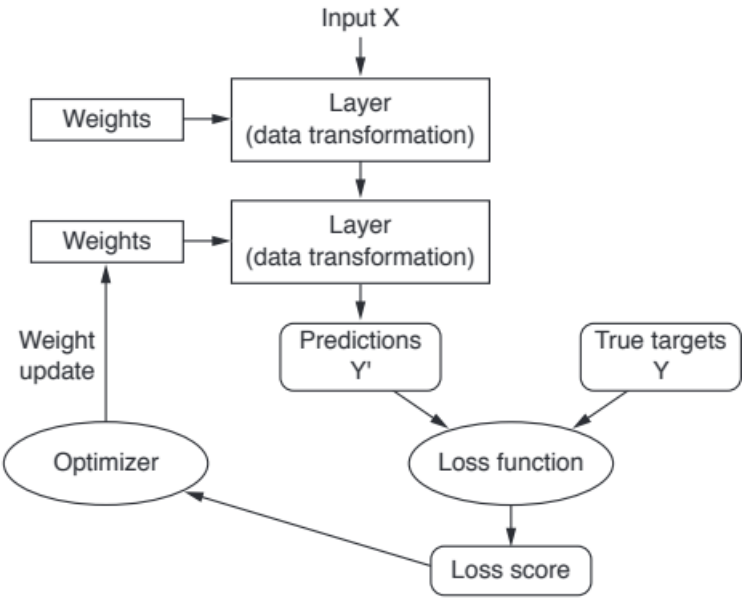

Figure 3: An example of training iteration [2].

In this research, we use CNNs pretrained on a large dataset called *ImageNet* [31], and repurpose them to our classification problem. The implementation of those CNNs is available in *Keras* library.

ImageNet

ImageNet is a Large-Scale dataset used for image classification created by Dr. Fei-Fei Li [31, 32]. The dataset holds more than 15 million images belonging to 22000 categories [33]. The subset used for the data science competition ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) contains 1000 classes – mostly animals and common objects [2] – with roughly 1000 images each. It is subdivided into 3 sets. The training set has 1.3 million images, the validation set 50 thousands and the test set 100 thousands [32].

As *ImageNet* is the only very large public dataset with sufficient diversity to be easily repurposable [9], it is frequently explicitly referenced to as pretraining dataset in image classification and clustering research. [4, 7, 9]

VGG16 and VGG19

Deep learning-based Visual Geometry Group networks (VGG) architecture is a simple and commonly used CNN architecture developed for *ImageNet* Large-Scale Visual Recognition Challenge (ILSVRC) par Karen Simonyan et Andrew Zisserman in 2014 [2, 5, 7, 32]. One of the main characteristics of that architecture is that the convolution layers have a receptive field that is the smallest possible while keeping a notion of left and right, and up and down [32].

The difference between VGG16 and VGG19 is the number of convolutional layers, respectively 16 and 19. (cf. Figure 4)

VGG networks have a convolutional base made up of 5 convolutional blocks composed of 2 or more layers separated by max pooling, followed by a top classifier compose of 3 fully connected layers

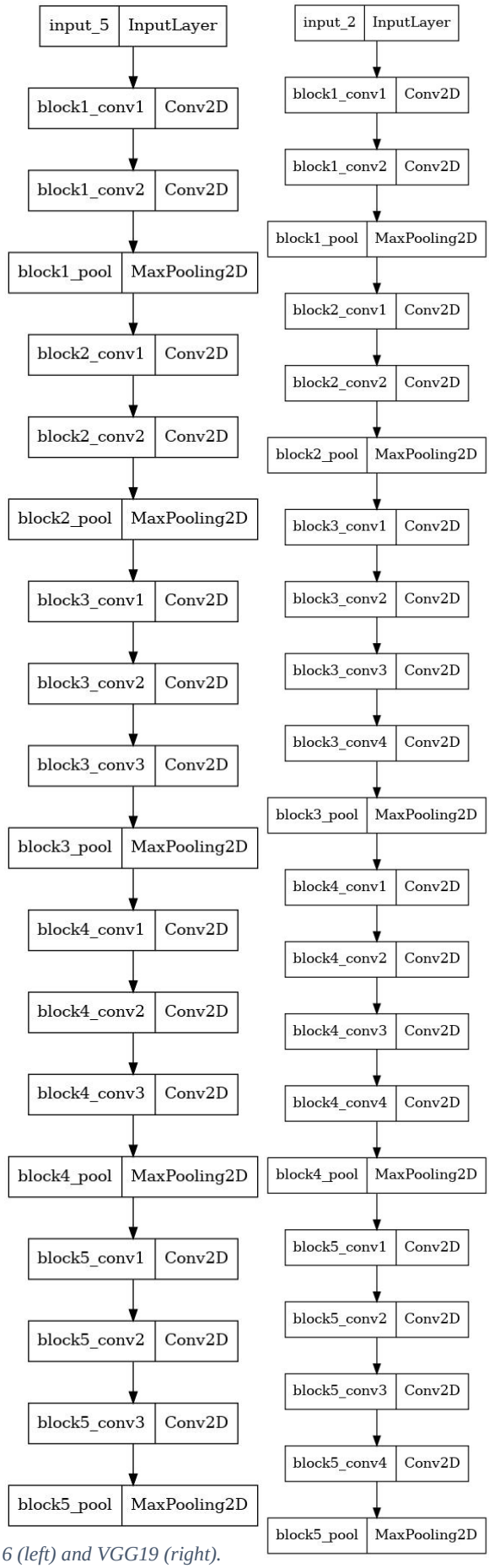

Figure 4: Architecture VGG16 (left) and VGG19 (right).

and an output layer *softmax*, which is the most appropriate activation function for multi-class classification. The activation function used in all hidden layers is *ReLU* that outputs either a positive value, either a zero which makes the computation faster. [5, 32]

This model is already quite old and cannot compete with the latest models in terms of performance. Nonetheless, this model is simple [2, 5, 32], it can easily be repurposed to a new problem [32] and it is implemented in the *Keras* library.

ResNet50

Residual Networks architecture, developed for ILSVRC 2015 by He et al. (2016) [34], is deeper than VGG networks while keeping a low complexity. The main characteristic of that architecture is the introduction of shortcuts which provides the gradient with an alternative road. That allows to resist the vanishing gradient problem that lowers the performances when there is a lot of layers. Residual learning ensures that the next layer performs equally good or better. [34]

ResNet50 has 50 layers structured into 5 steps that each have a convolution block an identity block composed of 3 convolution layers (cf. Figure 5). [34]

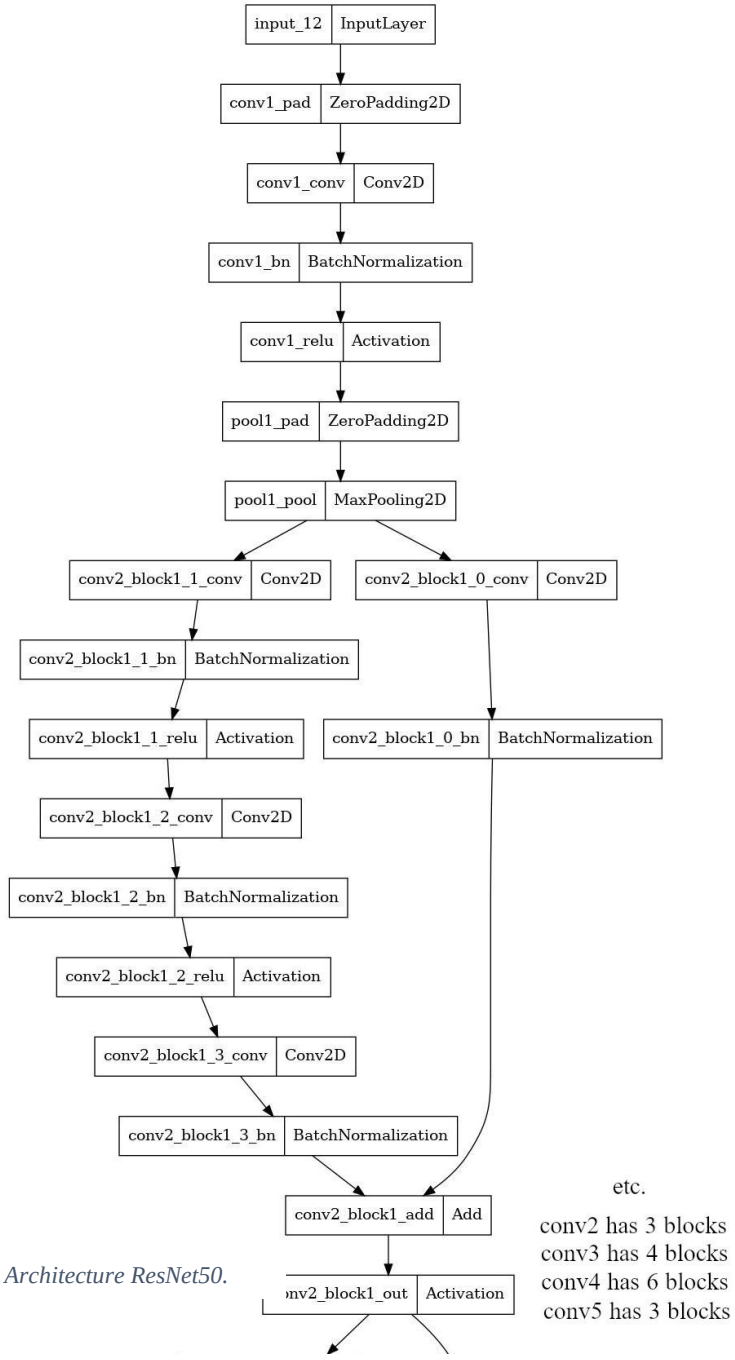

Figure 5: Architecture ResNet50.

Xception

Xception architecture is inspired of Inception architecture and outperforms it on *ImageNet*. Inception modules are replaced with depthwise separable convolutions. The main difference between the two is the presence or absence of non-linearities, respectively. The particular assumption of the Inception architecture is that it is possible to consider cross-feature correlations independently from space correlations and therefore to simultaneously map them (cf. Figure 6). [35]

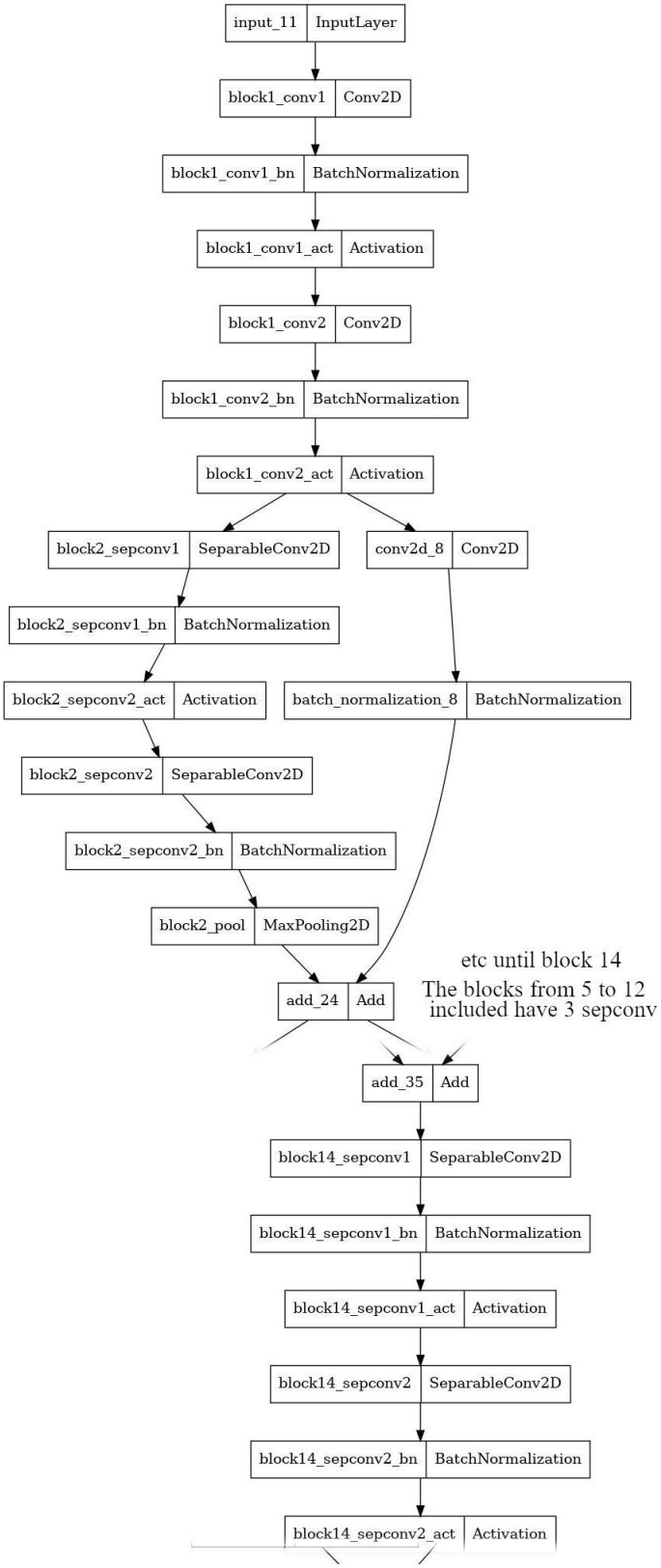

Figure 6: Architecture Xception.

Class imbalance

The class imbalance problem has been extensively studied. It is often the under-represented class that is the most important one. For instance, the presence of a tumor is less frequent than the presence of a healthy body. It is then crucial that the under-represented class is correctly identified. There exists a myriad of strategies to address the class imbalance problem. Those strategies can mostly be classified into two categories: data-oriented approaches which aim at modify the dataset before training to restore balance, and algorithm-oriented approaches which adapt the training process to take into account the difference in representation. [21, 22]

Balanced Classification Rate (BCR)

The most common metrics to measure the performance of classifiers are not always the most appropriate in a class imbalance problem. By using traditional metrics, there is a risk to develop sub-optimal models, i.e. biased towards the dominant class. That is why it is crucial to carefully choose the performance metrics. For instance, accuracy is a very popular metric, but it is not appropriate in a class imbalance context because the impact of an under-represented class on the accuracy value is lower compared to the impact of the dominant class. As an example, if we have a dataset used to classify cats and dogs composed of 80% of dogs and 20% of cats, the accuracy would already reach 80% if the classifier predicts that all samples are dogs. [23]

The 'Balanced Classification Rate' (BCR) is a good metric for a dataset that is not initially balanced. The BCR computes the classification rate of each class and average them. When the classification problem is binary, as in this case, the BCR is the average of *specificity* and *sensitivity*. [36]

Sensitivity measures the proportion of correctly classified positives while specificity measures the porportion of negatives identified as such. They approximate the probability that a positive or a negative respectively is correcty classified. Even though both metrics are relevant in a class imbalance problem, they do not give enough information individually. [37] That is why we use the BCR to measure classification performance.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Specificity = \frac{TN}{FP + TN}$$

$$Sensitivity = \frac{TP}{TP + FN}$$

$$BCR = 0.5 * (Specificity + Sensitivity)$$

Where TP (TN) refers to the number of positives (negatives) classified as positive (negative), i.e. True Positives (Negatives), and FN (FP) represents the number of positives (negatives) classified as negative (positive), i.e. False Negatives (Positives) (cf. Table 1).

	Actual positive	Actual negative
Predicted positive	TP	FP
Predicted negative	FN	TN

Table 1: Confusion matrix.

Let's reuse the example of the dataset with 80% dogs and 20% cats. If the classifier identifies all individuals as dogs, it reaches an accuracy of 80% but a BCR of only 50%. If it is important to correctly classify cats, BCR is a better metric to measure the performance of that classifier than accuracy.

There exist other metrics that suit an imbalanced dataset, such as F-measure which combine precision and recall (also called sensitivity), G-Mean which compute the geometric mean of accuracies of each class, AUC that refers to the area under the receiver operating characteristics (ROC) curve which evaluate the trade-off between TP and FP (benefits and costs), etc. [23]

$$Precision = \frac{TP}{TP + FP}$$

Feature extraction and fine-tuning

Feature extraction

Feature extraction is the process of extracting interesting features of new data from the representations of a network pretrained with other data. These features are then used to train a new classifier (cf. Figure 7). In the case of image classification, features are extracted from the convolutional base of a pretrained network and used to train a new top classifier. [2]

Only the convolutional base is used to extract features because the representations learned by this base are more generic and repurposable whereas the representations learned by the top classifier are more specific to the training problem. The convolutional base in computer vision detects generic concepts of an image, which is easily re-applicable to other contexts. It is also in this base that we consider information on where those concepts are locally while that space notion is lost in the top classifier. Therefore, if that space notion is relevant for the new problem that we are trying to repurpose the model for, there are not many arguments in favor of repurposing the top classifier to that new problem. [2]

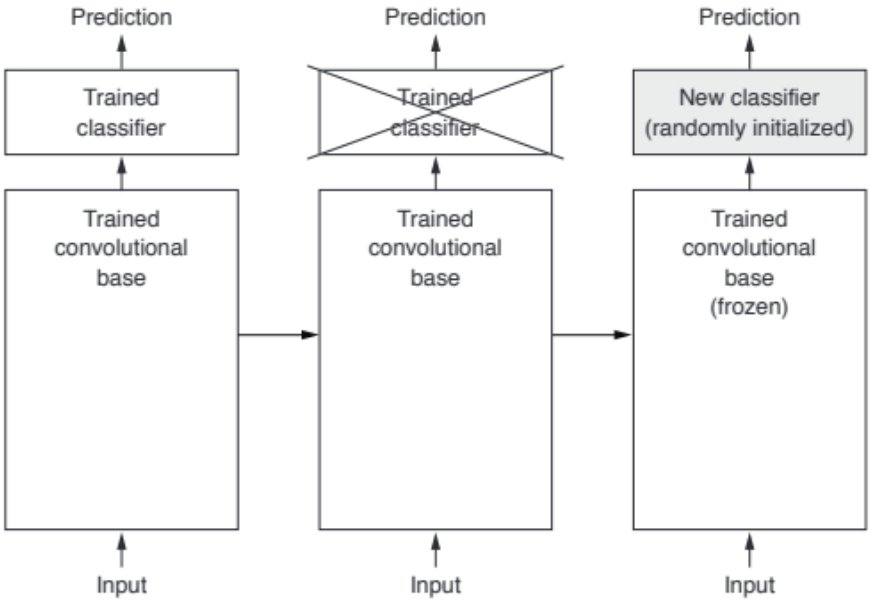

Figure 7: Replacing top classifiers while keeping the convolutional base [2].

In terms of implementation, the most efficient technique is to first extract the features out of the convolutional base and use them as input to train the new top classifier. Another technique would be to simply add the new top classifier on the convolutional base – set as non-

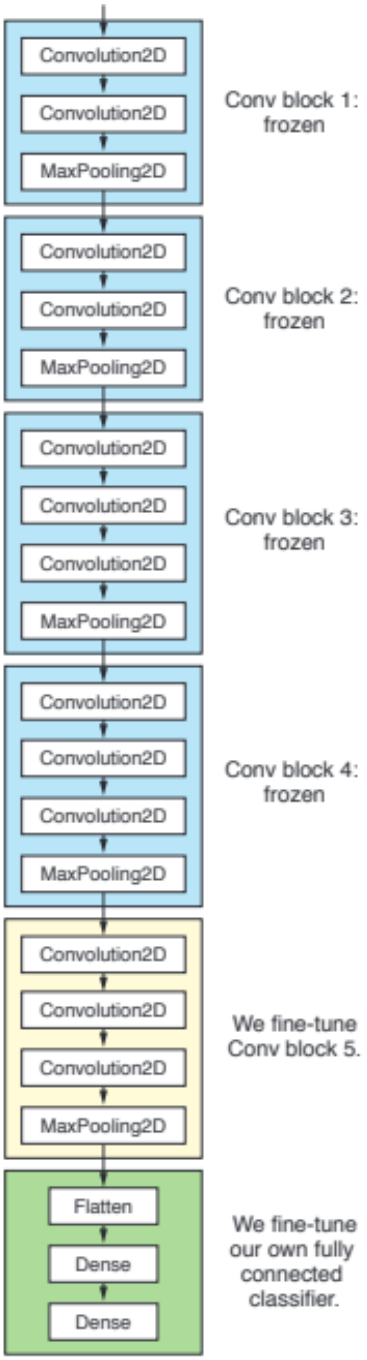

Figure 8: An example of fine-tuning on VGG16 [2]

trainable – and train that new merged model. However, the costliest part of the operation in terms of time and resources, is to extract the features from the convolutional base. The first technique performs that part only once – and then train the classifier with the extracted features as input - while the second has to extract the features at each iteration of the training as the input has to go through the whole convolutional base. [2]

Fine-tuning

Fine-tuning is a complementary approach to feature extraction. This approach consists of unfreezing the last layers of the convolutional base in order to jointly train the base and the top classifier (cf. Figure 8). The point is to adjust the last layer so that they become more specific to the current classification problem. As we have seen before, deeper we go in the model, more specific are the representations. Reciprocally, closer we are to the beginning of the model, more generic is the information. That is why it is more interesting to re-train only the last layers of the convolutional base rather than the first ones. [2]

In terms of implementation, it is necessary that the new top classifier is pretrained with the new data before unfreezing the last layers of the convolutional base, otherwise the error signal propagated during the training might be too big and the model would struggle to reach the global optimum. [2]

Ensemble learning

Ensemble learning is becoming more and more popular in the field of machine learning by giving a boost to accuracy. To illustrate that, it might be interesting to note that most of the recent data science competition winners have used ensemble techniques such as the winners of SIIM-FISABIO-RSNA COVID-19 Detection 2021, RANZCR CLiP - Catheter and Line Position Challenge 2021 and Cassava Leaf Disease Classification 2021 on Kaggle [10]. Ensemble methods combine the outputs of different base estimators in order to improve the prediction accuracy, generalizability and robustness. [2, 30, 38]

Ensemble learning assumes that different good models trained separately learn different things, focus on different aspects, and are then potentially complementary. Therefore, the key to ensemble learning is diversity. [2]

There are three big classes of ensemble learning techniques [30].

The first class refers to **mixing training data** (*bagging*). Instead of training a big classifier on a big dataset, the classifier is duplicated and each copy is trained on a different subset of the big dataset. Next, the classifiers are pooled to get the final result. That approach ensures that each classifier captures sufficient diversity in its subset.

The second class of ensemble learning approaches is **mixing combinations** (*boosting*, *stacking*). The idea is to combine models in various ways. *Boosting* for instance starts like bagging but if a learning model is not as performant as the others, more resources can be allocated to its training. *Stacking* refers to adding a meta learner on top of one or several base learners (cf. Figure 9). Base learners make predictions that are used as input by the meta learner which combines them as best as possible. Without a meta learner, predictions pooling leads to more errors.

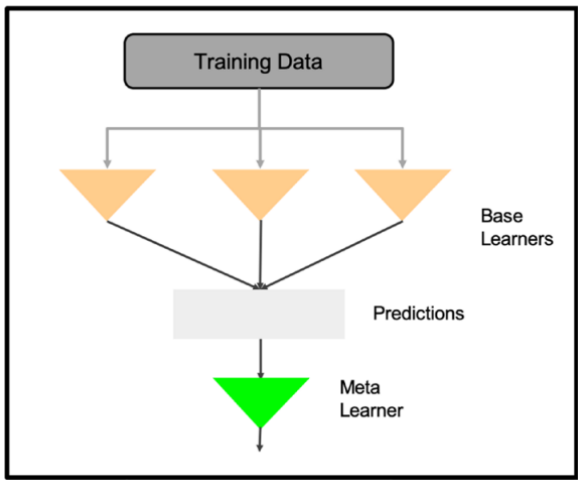

Figure 9: Stacking [30].

The third and last class concerns **mixing models**. Instead of using one model architecture on different subsets of the larger dataset, different types of model or different

hyperparameters are used (cf. Figure 10). The added value is getting better accuracy and lower bias.

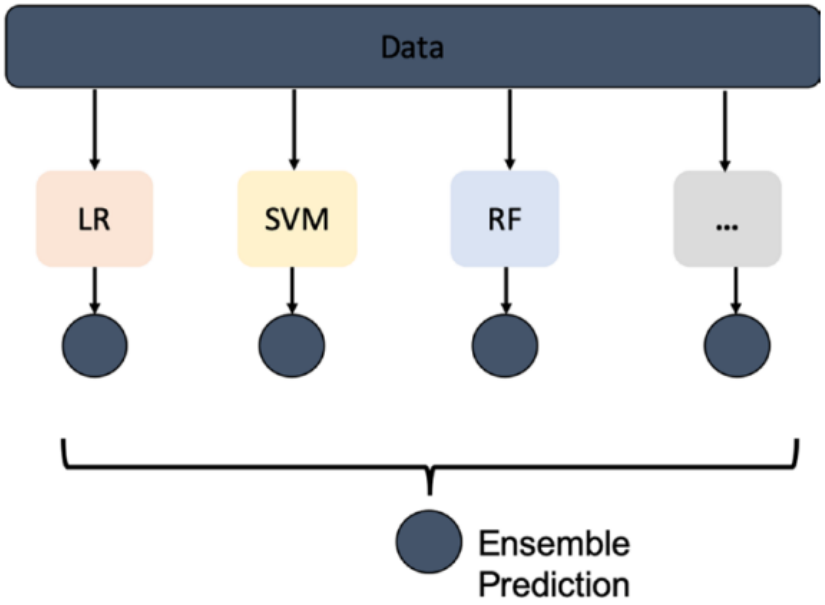

Figure 10: Mixing models [30].

Voting ensembles

When mixing models, one needs at some point to combine the results. Voting ensemble is one way to do the combination. [30]

In **majority voting**, each model votes for a class and the class with the most votes is chosen. [30]

Averaging or soft voting takes the prediction probability of each classifier and average them to get a final prediction probability. It only works if the classifiers are equally good. One can also decide to give different weight to each prediction and that is then called **weighted averaging**. [2, 30]

Bagging

Bagging is short for *bootstrap aggregating* and is a technique that divides the dataset into several samples with replacement which are each used to train a different model (cf. Figure 11). Those models are then combined to reach a final prediction. [30]

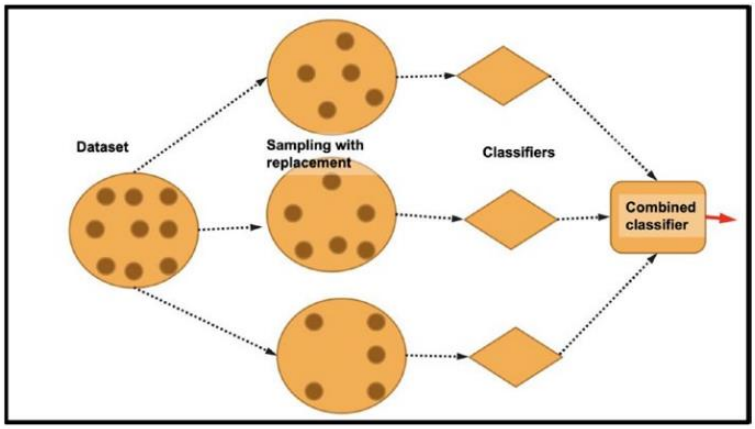

Figure 11: Bagging with replacement [30].

A **random forest** is a bagging technique that combines several decision trees which are trained on different subsets of the dataset. A **decision tree** is a top-down flowchart-like method where each node represents a decision to make based on one or several independent variables (cf. Figure 12). The tree continues until the dependent variable can be predicted or the depth is explicitely limited. Deeper is the tree, better is the accuracy but bigger is the risk of overfitting. To limit that risk, one can use a random forest. [30]

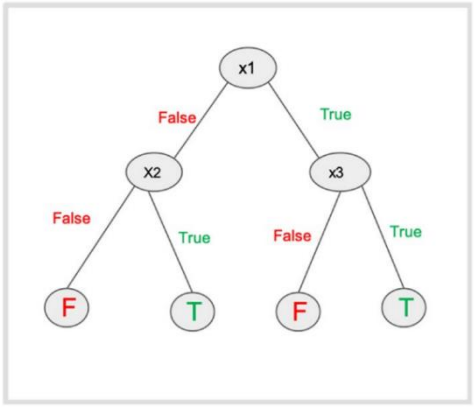

Figure 12: A decision tree [30].

Boosting

Boosting is the idea that more resources can be spent to improve the learning of the weak learner. [30]

AdaBoost increases the weight of the incorrectly classified data and run the training again for *n* iterations. After the last iteration, the classification decision is taken based on a combination of the *n* learners by voting (cf. Figure 13) [30].

Gradient boosting trains a new model based on residual errors made by the previous predictor. Residual error is the difference between the target value and the actual prediction. If 1 is the target value and the prediction is 0.8, the residual error is 0.2. In order to use gradient boosting, one should use a regressor rather than a classifier. [30]

XGBoost is a state-of-the-art algorithm that actually improves the gradient boosting algorithm by adding techniques such as Newton's tree boosting and randomization.[30]

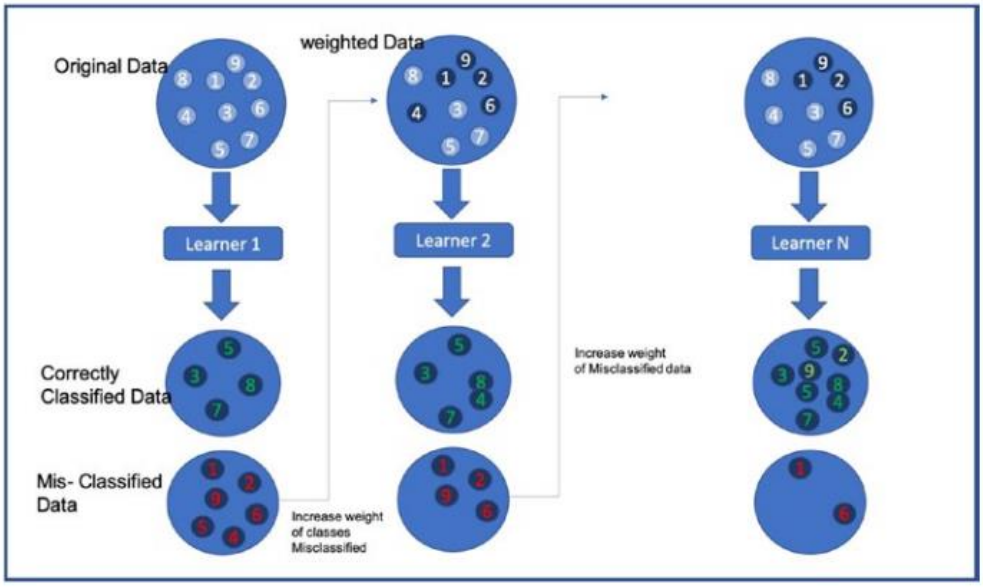

Figure 13: AdaBoost Boosting [30]

Stacking

Stacking firstly trains base learners to get predictions that are then used for classification by a meta learner. It is a way of combining the predictions more intelligently than hard voting. [30]

An example of stacking meta learner classifier is a **logistic regressor** (LR), also known as logit regressor, maximum-entropy classifier or log-linear classifier. Even though regression is in its name, it refers to a linear model mainly used for classification. The model computes the probabilities of belonging to each class with a logistic function. [39] When used in stacking, the predictions of base learners are inputs to the logistic regressor.

An other example of meta learner is a **Support Vector Classifier** (SVC). It uses Support Vector Machines (SVMs) that refers to a set of supervised learning methods. It is supposed to work well in a high dimensional space. It constructs one or several hyperplanes in the data space to separate the data and classify them. The hyperplanes depend on the kernel function chosen for the SVC as you can see in Figure 14. [40]

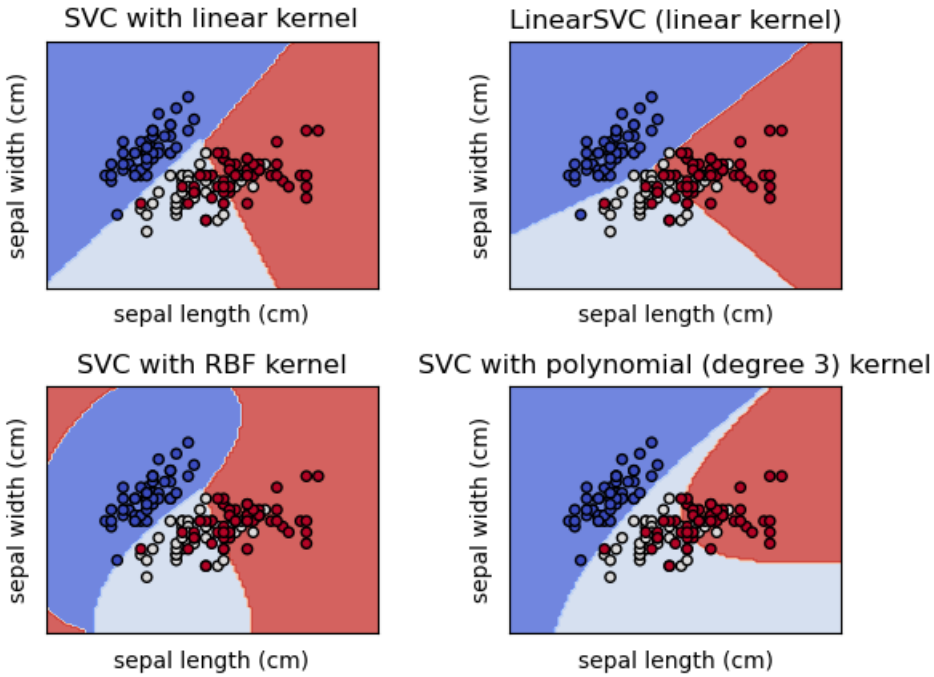

Figure 14: Hyperplanes construct by SVC on iris data depending on the kernel function (SVM).

A third type of meta learner is a **Multi-layer Perceptron Classifier**. It is basically a densely connected neural network. In other words, each neuron in a layer is connected to all the neurons in the next layer (cf. Figure 15) [1]. It uses a supervised algorithm called Multi-Layer Perceptron (MLP). It can be composed of one or more hidden layers and learn a non-linear function approximator for classification given a set of input features and a target output. A disadvantage of this meta learner in that it can remain stuck in a local minimum. [41]

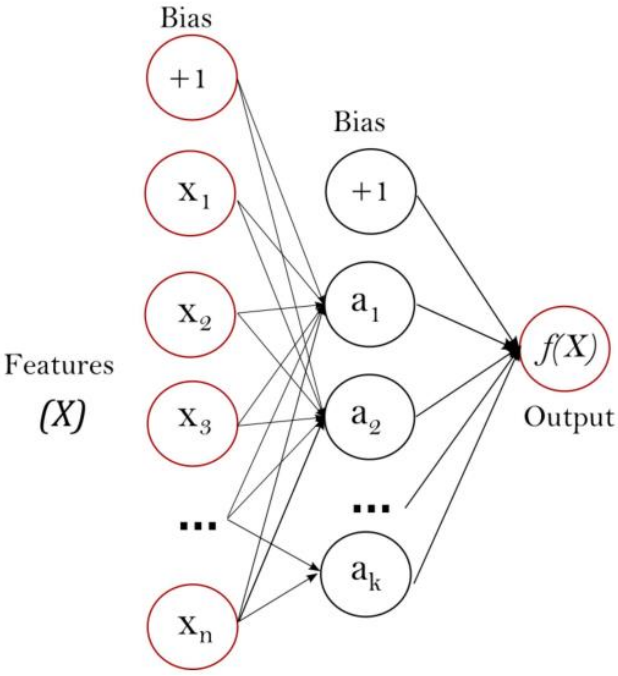

Figure 15: One hidden layer MLP with scalar output (MLP).

Dataset

As a reminder, the dataset comes from the digital twin of a crossroad with traffic lights in *Unreal Engine 4* [42]. In this digital twin, the world is simpler than the real world. Firstly, vehicles follow a predefined track and do not need to make direction-related decisions. Secondly, speed limit exists but is the same everywhere. Thirdly, vehicles go straight in the intersection, they do not turn. Finally, if the road has two lanes, vehicles do not change lane. Therefore, the only big decision that has to be made is how to adjust the speed, which we have simplified as a binary decision: braking or not.

To create our dataset, the driver's frontal point of view of 77 driving vehicles were captured at nighttime at the rate of 10 frames per second. The vehicles make decision in the crossroad based on built-in decision trees in the game engine and register all the information around that vehicle at each frame, such as whether it brakes or not.

The dataset is then reduced and resized. The 20 first frames of each driving vehicle are taken out because it is the warming up period before the vehicle behaves consistently. Because the pictures are temporally very close and does not bring significantly new information, it was not worth keeping the 10 frames per second, so we reduced the stream to 0.8 frame per second. Finally, we reduced the size of the pictures to 381x223 pixels. As it can be seen in Figure 16, the difference is hard to tell.

Figure 16: An example image from the dataset original (left) and resized (right).

The dataset contains now 3543 frames where 2373 are labelled as *not braking* and 1170 are labelled as *braking*, i.e. 33% of the frames are labelled as *braking*. That means that the dataset is slight imbalanced.

Besides speed and steering angle, the dataset contains one more information that gives insight on the types of situations present in the dataset, namely the state of the driving vehicle in the frame: "driving", "stopping for collision", "going straight" or "stopping at traffic light".

"Stopping for collision" refers to stopping for an obstacle such as another vehicle, "Going straight" to crossing the intersection, "Stopping at traffic light" is self-explanatory, and finally "Driving" refers to the rest. An overwhelming 60% of the frames are in the "driving" state, 21% in the "stopping for collision" state, 12% in the "going straight" state, and 7% in the "stopping at traffic light" state.

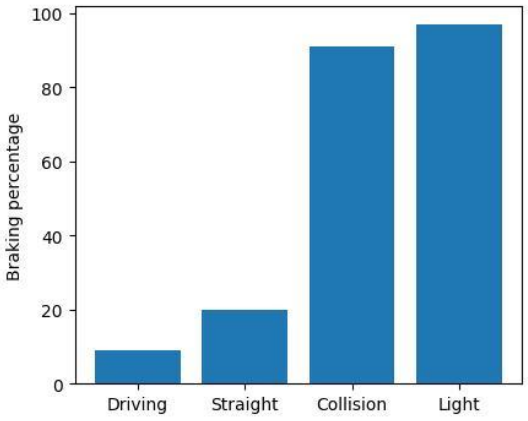

Figure 17: Proportion of braking cars for each state.

One can see on Figure 17 that the states are consistent with the label. Most of the cars that stop for collision or traffic light are actually braking. On Figure 18, one can see the representation of these states between the 77 driving vehicles.

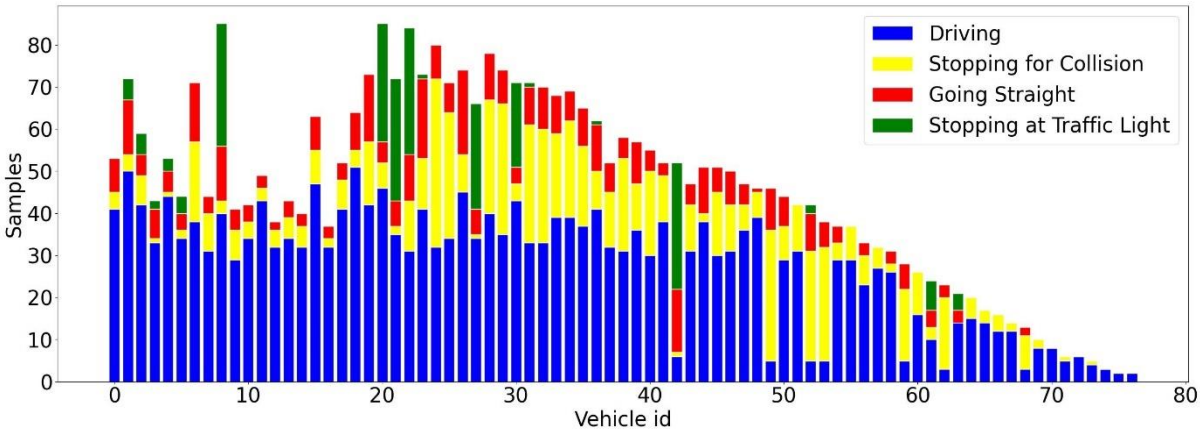

Figure 18: Proportion of each states for each vehicle.

Now we need to divide the dataset into at least a training and test set. To do that, we are using a variant of the K-fold algorithm where we are actually dividing the dataset into 3 subsets: a train set, a validation set and a test set. There are two reasons why we need a validation set. First, while tuning the hyperparameters, the model improvement is evaluated with the validation set so there is a risk of overfitting the validation set but with an extra test set, overfitting can be spotted sooner. The second and main reason is that we can keep a "virgin" training set, i.e. the validation set, for the ensemble learning (step 3 in the action plan) while still having a virgin test set.

We have chosen to do a 7-fold vehicle-wise. The dataset is then divided into 7 sets where each set contains 11 different driving vehicles point of view. The reason why the dataset is divided considering driving vehicles point of view and not simply considering frames is the

frames within a same point of view are pretty close. That means that if the test set shares very close frames with the training set, the model performance evaluation could be biased. As it can be seen on Figure 19, the partitions are of equivalent sample size and all the training sets have samples of the different states that a vehicle can have.

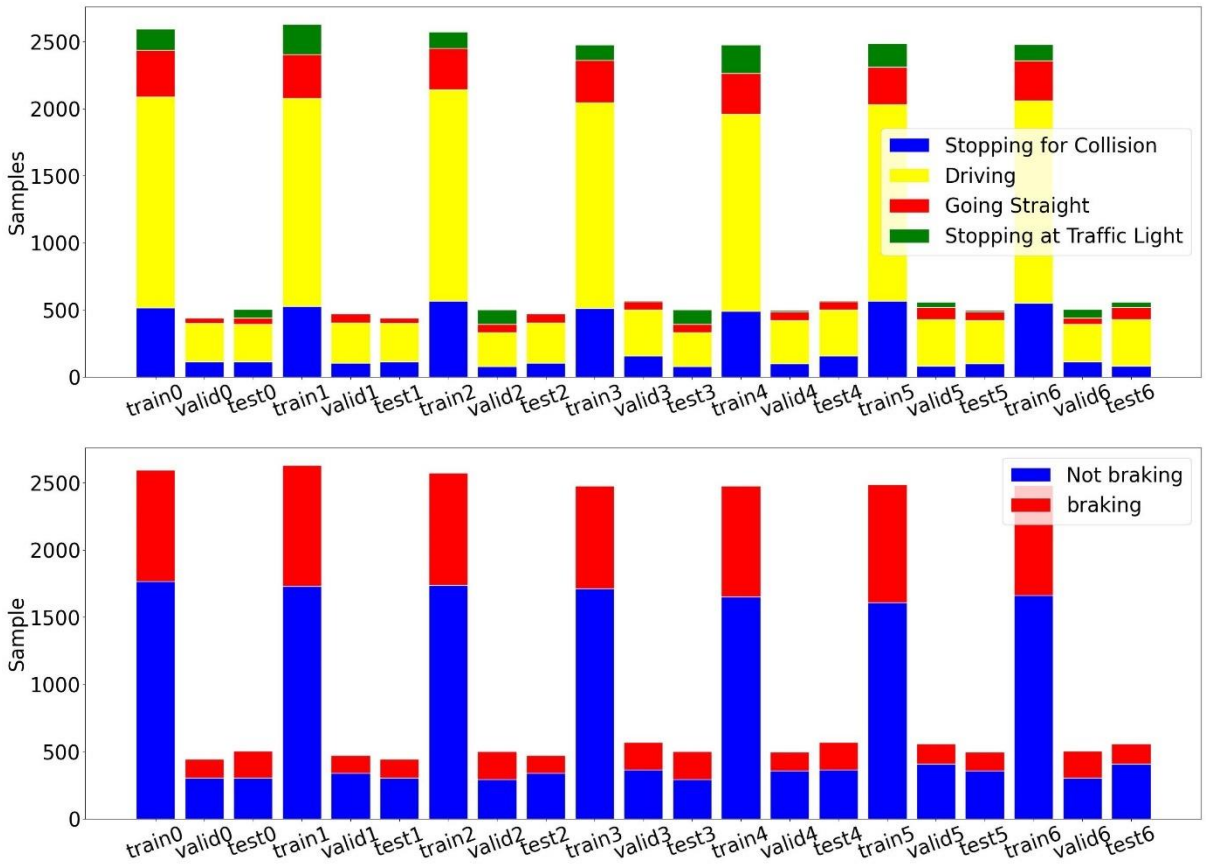

Figure 19: Summary of the proportion of states (top) and classes (bottom) within each fold.

Step 1: CNN with single visual input

The first step is to build a model that takes one single visual input and predict whether the car in this image is braking or not.

As mentioned in the literature section, CNNs are the most effective models for visual inputs. In this section, four pretrained CNNs are compared. The four CNNs that are compared here are all available in *Keras* library and are pretrained with *ImageNet*: VGG16, VGG19, ResNet50 and Xception.

There are two big approaches to repurposing pretrained models: feature extraction and fine-tuning. [2] Those two approaches are also compared in this section.

When exploring the dataset, it appears that there is a slight class imbalance. Indeed, only one third of the samples are labelled as braking. We have then chosen to use a relevant metric to evaluate the performance of the models in the context of imbalanced data. The metrics that we have chosen is the Balanced Classification Rate (BCR).

Models

In order to perform feature extraction, one has to remove the top classifier from the pretrained model. In this case, the function is implemented in *Keras* and we can choose not to add the top classifier when we load the model. We then add a new classifier taking the convolutional base of the pretrained model as input as shown on Figure 21. To have a clear table of features, the output of the convolutional base is flattened – in other words each sample has one feature vector – before it is used as input to our top classifier. We have chosen to build the new top classifier with a prediction dense layer that uses a *sigmoid* activation function, on top of a dropout layer, on top of another dense layer that extracts 256 features with a *ReLU* activation function (Rectified Linear Unit activation function) (cf Figure 20). That actually makes a multi-layer perceptron classifier.

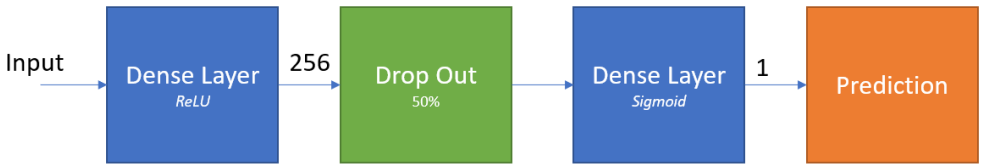

Figure 20: Top classifier architecture.

As a reminder, a *dense layer* is a layer where each of its neurons is connected to every neuron of the previous layer. A *dropout layer* uses a regularization technique that consists of randomly setting to zero a proportion of neurons. That induces noise in the output values of a layer which can help remove patterns that happen to be memorized but actually not significant. [2] A regularization technique tries to prevent the model from overfitting, i.e. focusing on elements outside of the true properties of the dataset. A good dropout rate is usually between 20 and 50%. We arbitrarily use a rate of 50%. An activation function returns the output of the layer based on its input.

The *sigmoid* activation function is the most appropriate for binary classification because it outputs a value between 0 and 1. That is why we use that activation function in the last layer of the model, i.e. the prediction layer. For the hidden dense layer, we have tested two activation functions, the hyperbolic tangent function (*tanh*) that outputs a value between -1 and 1, and the *ReLU* activation function. The *ReLU* activation function outputs a value positive or 0 if the value was negative. When a neuron has a value of 0, it is not activated and does not learn, which makes the computation faster [43]. We have kept *ReLU* for the hidden layer because it was faster and seems to have better results than *tanh* (cf. Appendix 2). We have also tried to add a second hidden layer and a second dropout layer to our top classifier but it did not improve the BCR (cf. Appendix).

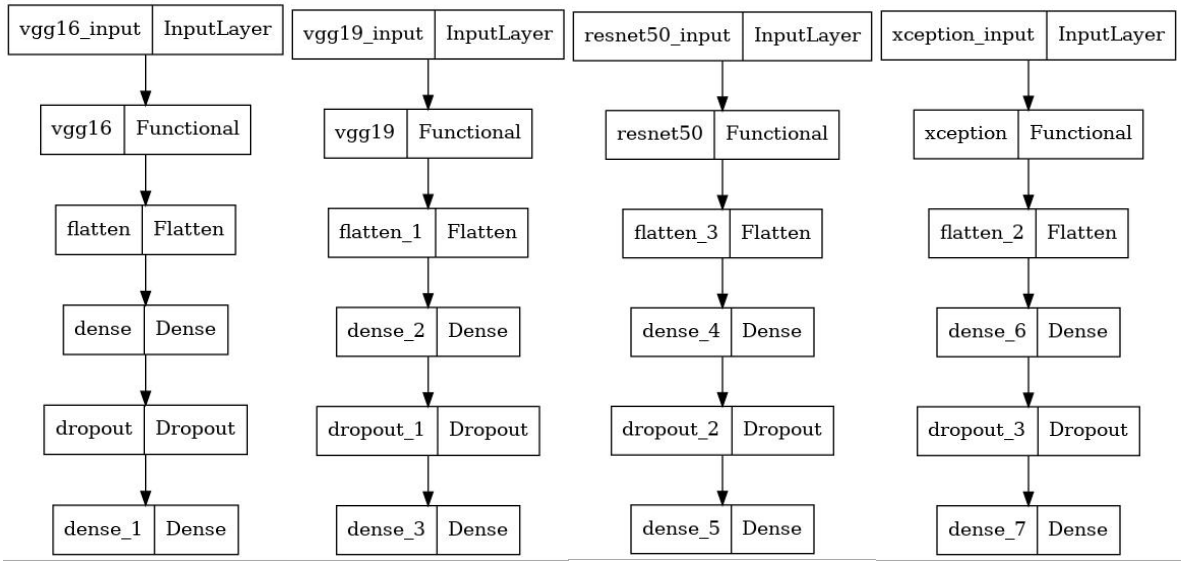

Figure 21: Feature extraction architecture of the four CNNs.

Following the steps of the related work, the models were compiled and trained using an *Adam* optimizer which uses a stochastic gradient descent optimization method [24]. *Adam* actually works well on relatively large datasets in both training time and validation score [44].

The optimizer uses a loss function that gives a loss value to minimize to train the model. We tried several loss functions such as *Sigmoid Focal Cross Entropy* or *Binary Focal Cross entropy* that were supposed to perform better in a class imbalance context. Focal loss is used in a class imbalance context because it down-weights well-classified samples and gives a higher loss value to misclassified samples [25, 26]. Yet, the loss function that has brought the best results is simply *Binary Cross Entropy* that measures the difference between the probability distributions of the predicted labels and the true labels (cf. Appendix 2). [45] We are keeping those two parameters for the rest of the work.

Several combinations of learning rates, number of epochs, i.e. an iteration over the entire dataset provided [46], and batch size, i.e. the number of samples per gradient update [46] have been tested (cf. Appendix 2). We have settled on using a batch size of 30 for VGG19, ResNet50 and Xception, and a 10 for VGG16, 20 epochs for all and a learning rate of 10^{-4}. If the learning rate is too small, the training will take forever before it converges. If the learning rate is too big, the training might be stuck in a local minimum. It is also worth mentioning that the image input is standardized because CNNs prefer to deal with number between 0 and 1.

For fine-tuning, we have tested several possibilities regarding the number of last layers to re-train (cf. Appendix 3). VGG16 and VGG19 models both performs better when re-training from the layer *block4_pool* rather than *block3_pool* (cf. Figure 4). Xception model performs better when we re-trained the layers from *add_11* rather than from *add_10* (respectively *add_35* and *add_34* on Figure 6). ResNet50 is the only model that performs better when more layers were re-trained. The BCR is better when the layers from *conv3_block4_out* were fine-tuned rather than from *conv4_block6_out* (cf. Figure 5). The schema on Figure 21 can help visualize how we have fine-tuned the models.

In order to train the models, we have once again made a choice concerning the number of epochs, and batch size. We have kept a learning rate of 10^{-4} for all. The number of epochs is 20 except for Xception that trains on 10 epochs, and the batch size is 10 except for ResNet50 that is trained with a batch size of 50. The reason why it is important to mention this is that the choice of the combination of learning rate, epochs and batch size influences the accuracy and BCR scores in a significant way. For instance, when tested with different combinations, ResNet50 has mostly scored 50% except with a big batch size. Because we are in a situation of class imbalance, we have also tried to train the models with and without class weights. One can add class weights to the training to give more importance to the class that is under-

represented. It turns out that VGG16 and VGG19 perform better with class weight, while the others do not (cf. Appendix 3).

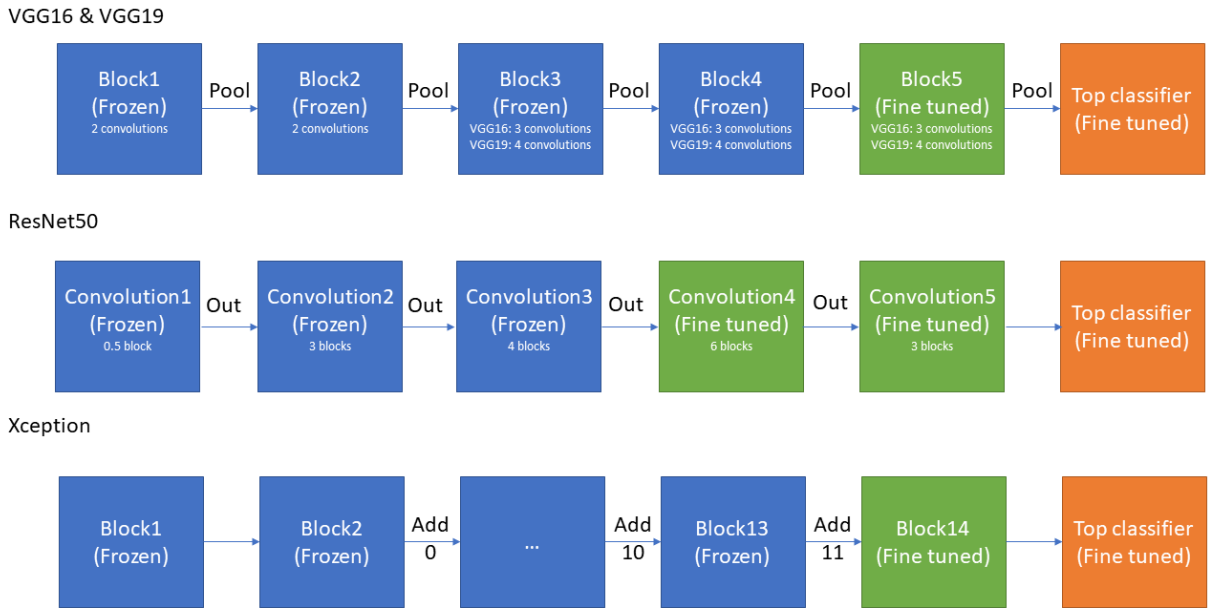

Figure 22: Summary of the fine-tuning strategy.

Results

In the following, we refer to the fine-tuned models with the name of the CNN architecture, even though it is not exactly that model.

Let's first have a look at the performance of the models in a feature extraction context (cf. Figure 23 (left)). Feature extraction performs already pretty well as it reaches a high accuracy of 76.9% and an acceptable BCR of 68.5% with VGG19. VGG16 and Xception are not far behind with an accuracy of 75.7% and 73.9% respectively and a BCR of 66.9% and 63.5%. ResNet50 has the least satisfactory result as it scores an accuracy of 68% with a very high variance between the seven folds, and especially a BCR of 50% with a null standard deviation. That basically means that it gives the same predictions to all the samples. It is interesting to note that by classifying all samples into the same class, ResNet50 still reaches a relatively high accuracy compared to the other models. That is the effect of the class imbalance problem. This is why it is so important to monitor the BCR.

Now if we have a look at the performance of the fine-tuned models (cf. Figure 23 (right)), it is interesting to note that all the accuracies are pretty close. VGG19 and Xception have the best accuracy scores - 75.5% and 75.1% - closely followed by VGG16 and ResNet50

that scores at 74.9% and 73.8% respectively. ResNet50 has clearly the highest variance for accuracy and BCR while VGG16 and VGG19 have the highest BCR scores and the lowest variances. The best BCR score belongs to VGG19 with 71.4% followed by VGG16 with 69.7%, Xception with 67.2% and ResNet50 with 62.9%. It is interesting to note that we have adjusted the class weights to train the two models with the highest BCR scores while that technique could not be successfully applied to the two models with the lowest BCR scores.

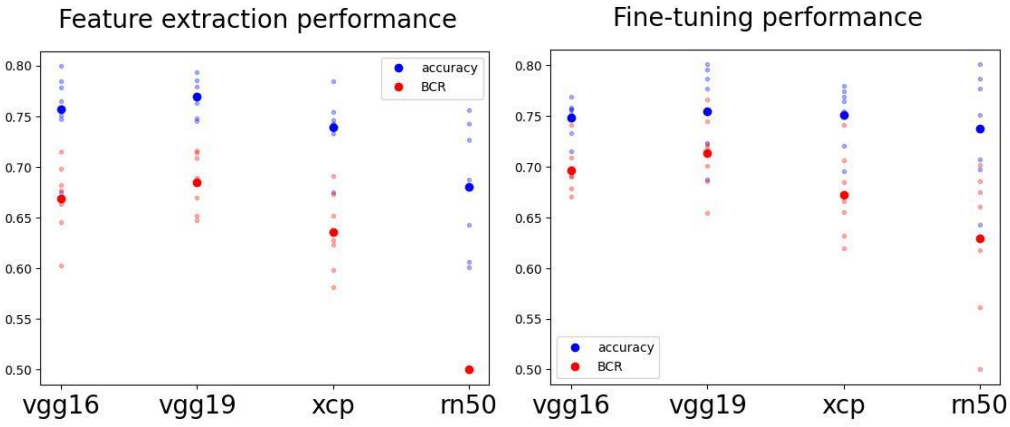

Figure 23: performance of the four CNNs in a feature extraction context (left) and in a fine-tuning context (right). The small dots represent the accuracies and BCR of each fold of the K-fold algorithm while the bigger dot represents the mean accuracies and BCR o of the folds for each model.

When comparing performance of the models before and after fine-tuning, it is obvious that the fine-tuned models perform better (cf. Figure 24). Even though the accuracy of VGG16 and VGG19 is slightly better before fine-tuning, the BCR scores way higher in all models. Besides, as we mentioned previously, BCR is more relevant to monitor than the accuracy because we are in a class imbalance context. It is also worth noting that the models are closer in terms of performance. The accuracies are almost equivalent and the difference between the best and worst BCR scores is clearly smaller than before fine-tuning, as ResNet50 has dramatically increased its BCR score.

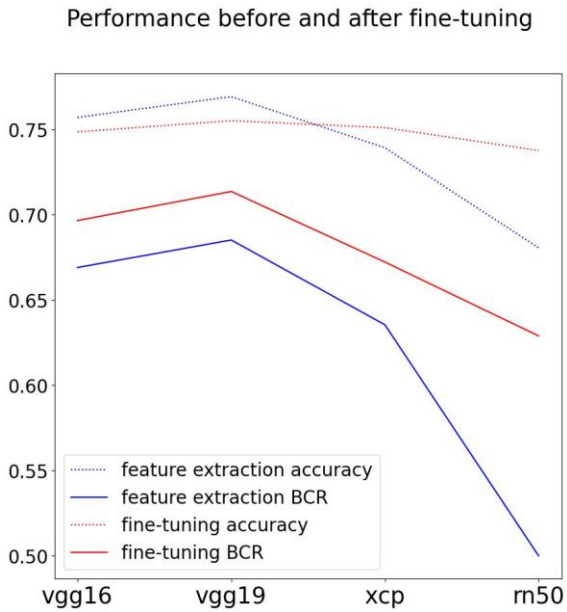

Figure 24: Comparison of the mean accuracies and BCR of four CNNs before and after fine-tuning.

On Figure 25, we can compare the performance of each fold of the K-fold algorithm. The main conclusion that we can make out of those graphs is that no set is easier to predict than another. The set with the highest score for one model can be the set with the lowest score for another.

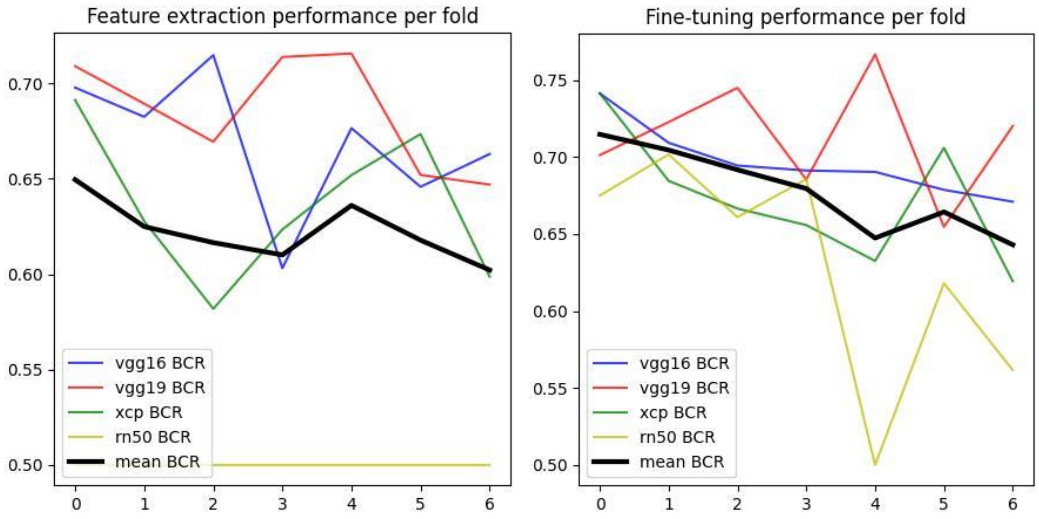

Figure 25: Performance on each fold for the four CNNs before (left) and after fine-tuning (right).

On Figure 26, we can compare the performance scores for states that a vehicle can be in. On both graphs, "Stopping for Collision" scores the highest, followed by "Stopping at Traffic Light" and "Driving". The one that has the lowest scores is "Going Straight". To find an explanation, we can look back at the dataset exploration. "Stopping at Traffic Light" is the least frequent state, but it also is the least ambiguous: almost 100% of the vehicle in that state are labelled as *braking*. That can explain why, despite being the most under-represented in the dataset, its score is not the worst. On the other hand, "Driving" is the most represented in the dataset and more than 90% of the vehicle in that state are labelled as *not braking* and yet, it does not have the best score. An explication to that observation could be that because the *braking* label is under-represented in that situation and in the whole dataset, the models perform poorly in identifying *braking* cars in that situation and as the BCR is *balanced*, the score is lower than expected. The lowest BCR score belongs to "Going Straight". It does make sense because that state is not very well represented in the dataset and is also rather ambiguous in a sense that around 20% of vehicles of that state are labelled *braking*. "Stopping for Collision" scores the best. It is the second state best represented in the models and it has a high percentage of the under-represented label so it can be expected to have a high BCR score. Intuitively, braking for collision or a traffic light seem also easier to predict than slightly braking to turn for instance, which could be labelled "Driving" or "Going Straight".

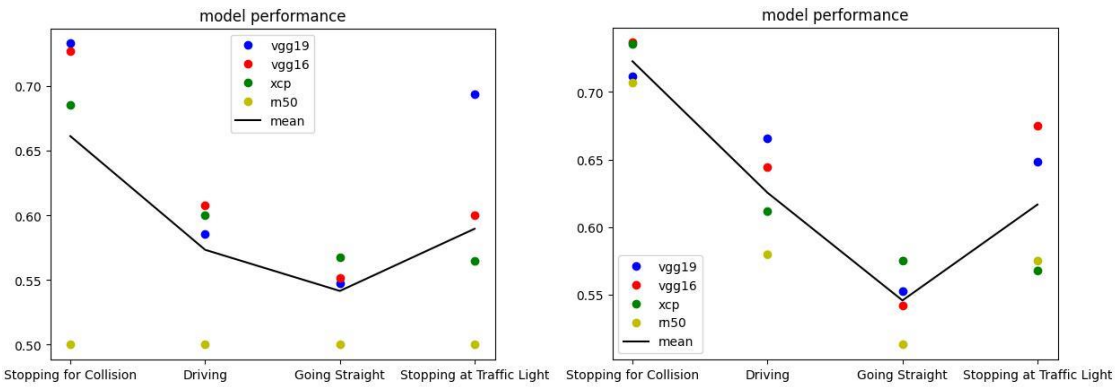

Figure 26: Comparison of the BCR scores of the different states that a vehicle may be in before (left) and after fine-tuning (right).

Conclusion

The models perform way better after fine-tuning than before. It makes sense because fine-tuning adjusts the last layers of the convolutional base to the current problem. It should be expected that the model performs better in a classification problem if it fits more specifically to that problem. However, we have also noticed that fine-tuning more layers in the models did not always improve the scores, which is intuitively odd. A possible explanation is that we have not found the right parameter to optimize the training yet.

Another interesting take is that BCR is more relevant to monitor than accuracy in a class imbalance context and adding class weights to restore balance to the training does not always improve the BCR. This observation should however be tempered because we have balanced the weights automatically while automatic weights are not always the optimal class weights. It would require a lot of trials, but one could most likely find a set of class weights that improves the BCR.

Finally, we can keep in mind that no fold of the K-fold algorithm is easier to predict so there are supposedly no bias in that sense and that the state of the vehicle is somehow correlated with the performance.

Step 2: Introducing numerical input

The next step is to introduce two relevant numerical inputs: the current speed and the steering angle. As we have seen in the literature, a common way to incorporate those inputs is to create a new model and merge it with the CNN. The merge can happen at different stages of the classification process [1]. We could merge the final classification of the fine-tuned model with new input and get a new prediction, but we have chosen to merge it a little earlier in the process (cf. Figure 27): just after the CNN convolutional base and before the top classifier so that the numerical inputs can be considered as a part of the extracted features.

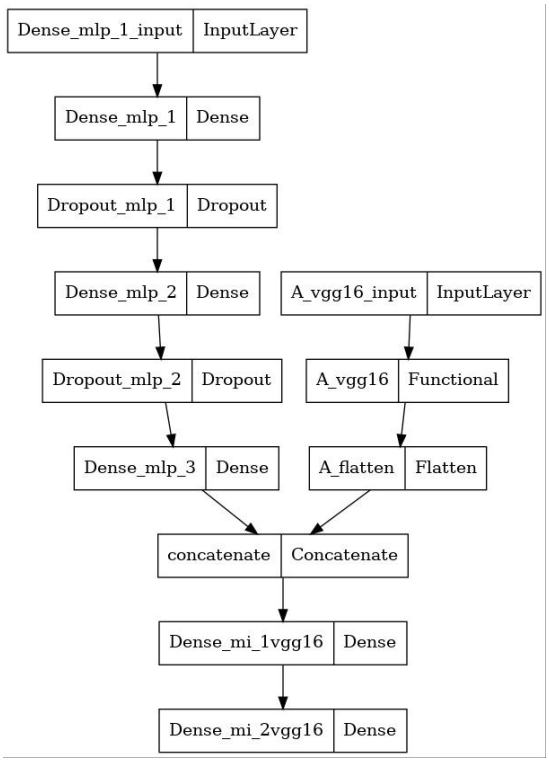

Figure 27: Model architecture when adding the numerical input.

Models

We use the fine-tuned convolutional base for the visual input and a custom MLP for the numerical input. The partial model dedicated to the numerical inputs consists of a dense layer that extracts 256 features out of 2 (speed and steering angle) in order to get all the information possible, then it is followed by a dropout layer and a dense layer that extracts 128 features and finally, after another dropout, the model extract 10 features to be injected in the merged model with the CNN (cf. Figure 28). We could not find a pre-built input generator in *Keras* that allows to combine scalar and visual inputs so we have built our own. The numerical inputs are standardized using a min-max scaler.

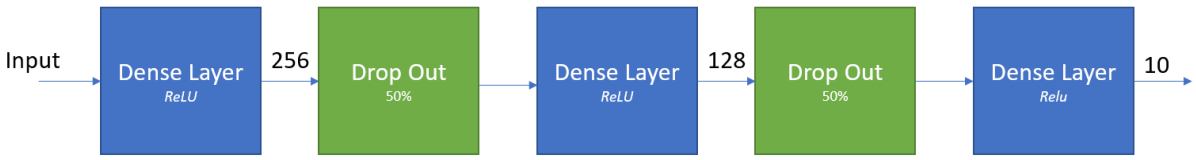

Figure 28: Partial model dedicated to numerical inputs.

After merging the features, the model needs a top classifier. That top classifier consists of two dense layers: one that outputs 100 features with a *ReLU* activation function and another that outputs the prediction with a *sigmoid* activation function. Once again, we have tested to

add hidden layers, i.e. dense layers, as well as other activation functions but that did not improve the score (cf. Appendix 4). In the following, we continue to refer to the models with the name of the CNN architecture, even though it is not exactly that model anymore.

We need to find the best combination of epochs and batch size. We keep the learning rate at 10^{-4}. VGG16 and VGG19 are trained on 10 epochs with a batch size of 30, while ResNet50 and Xception are trained on 11 epochs with a batch size of 10 and 50 (cf. Appendix 4). All the models perform better when balancing the class weights, except for ResNet50.

Results

Let's have a look at the performance of the newly developed models with numerical inputs (cf. Figure 29). VGG16 has the highest accuracy and BCR of 77.3% and 71.2% each, closely followed by VGG19 with 77.1% and 70.8%. Xception and ResNet50 are clearly behind with an accuracy of 73.6% and 73.5% respectively and a BCR of 67.3% and 66.2%.

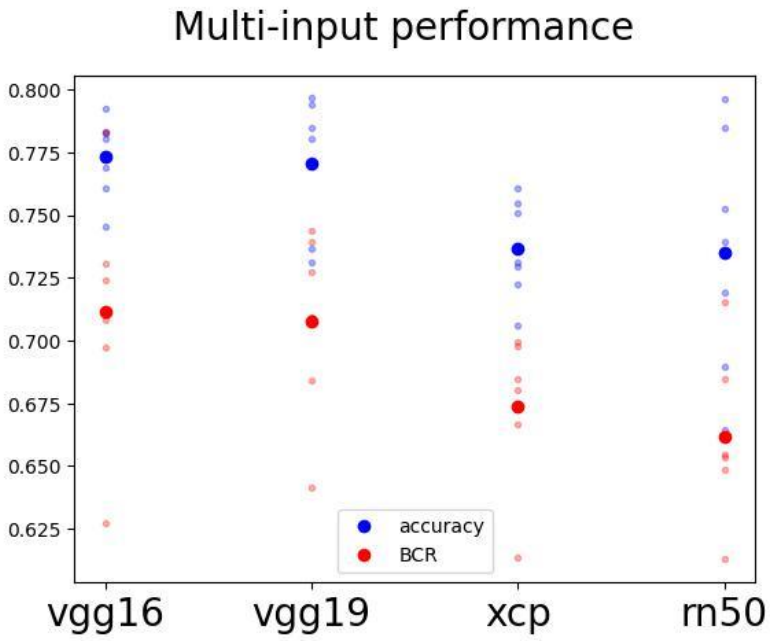

Figure 29: performance of the four CNNs in a multi-input context. The small dots represent the accuracies and BCR of each fold of the K-fold algorithm while the bigger dot represents the mean accuracies and BCR of the folds for each model.

When comparing the models' performance before and after adding the numerical input, it appears that the performance has improved for every model but VGG19 (cf. Figure 30). The improvement in the BCR is significant for VGG16 where the BCR has increased by more than 1 point compared to fine-tuning, and RN50 where the BCR has increased by more than 3 points.

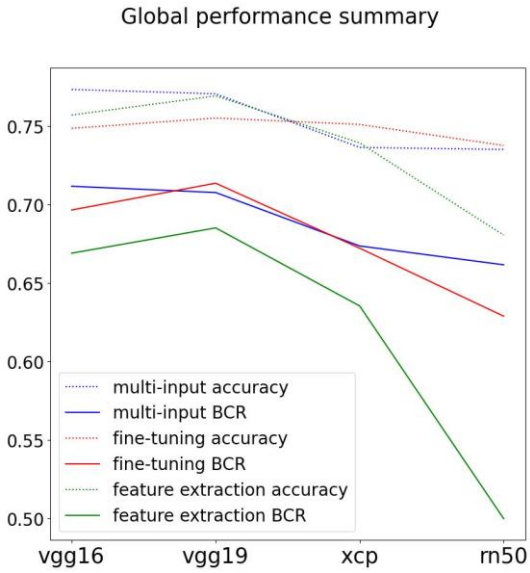

Figure 30: Comparison of the mean accuracies and BCR of four CNNs before and after adding the numerical inputs.

We can draw the same observations and conclusions from Figures 31 as we have done in the step 1 from Figure 26. Adding the speed and steering angle does seem to especially improve the performance on the "Driving" state. We have previously mentioned that, intuitively, braking for collision or a traffic light is easier to predict than slightly braking to turn for instance, which could be labelled "Driving". It is then a possibility that adding the steering angle has helped predicting those cases. Moreover, while 'Driving', a vehicle might brake to not exceed the speed limit, which does not change along the crossroad. That could also explain why the performance on the "Driving" state has improved when adding the speed.

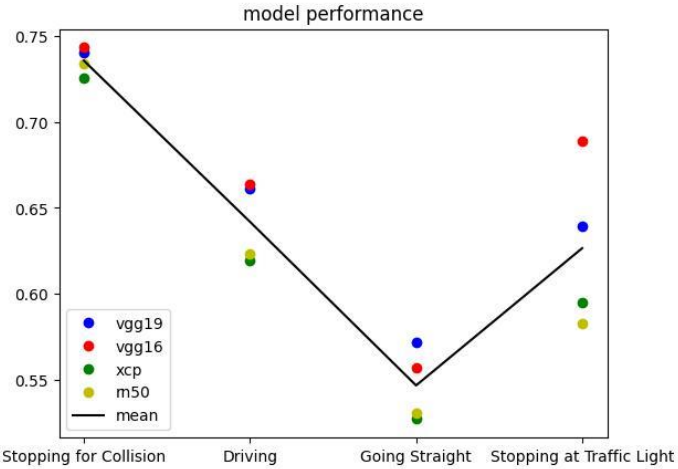

Figure 31: Comparison of the scores of the different states that a vehicle may be in for a multi-input context.

Conclusion

Adding speed and steering angle as numerical input to the model has not improved the performance as much as we would have expected. It could have even created some noise in VGG19 that did not improve at all. Nevertheless, it looks like it helps predicting the case where the car needs to brake before turning or to respect the speed limit. From those observations, we can conclude that speed and steering angle are not more relevant than other features derived from the picture, especially in the learning of VGG19 and Xception.

Step 3: Ensemble learning

Have you ever tried to draw a bike from the top of your head? It probably has the big features of a bike such as two wheels, a saddle, a frame, handlebars. However, some of those features are probably not even close to the reality. I have asked a couple of strangers on the train to draw a bike from the top of their head. You can see the results in Figure 32 and the picture of an actual bike in Figure 33. None of them has drawn a realistic bike. Therefore none of them could remember all the details of a bike. They have *learned* what a bike is by remembering the visual concepts. But even if we could expect them to remember the same concepts, each bike is different from the others. The common represented features are remembered in a way specific for each.

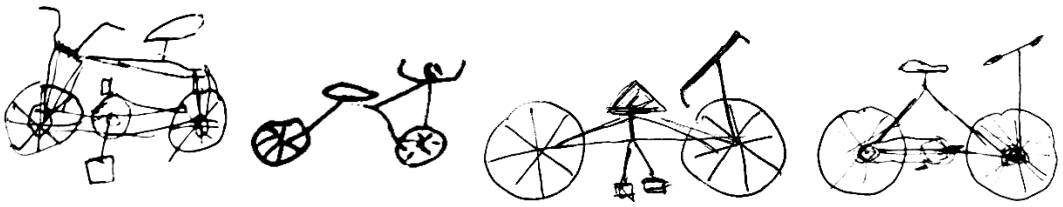

Figure 32: Bikes drawn without any reference.

Figure 33: An actual bike.

You have likely guessed where this is heading. That example illustrates how the models that we have built so far have probably learned the main features of our dataset in a way specific for each. The assumption behind ensemble learning is that models trained separately have learned things differently and could potentially be complementary. In other words, the assumption is that there is nothing to lose to combine them: either they have the same information, and the performance does not improve, either they have complementary information, and the predictions are better. [2]

To see that the models have paid attention to different aspects of the data, it is possible to visualize the visual concepts learned by the models through the channels, i.e the features, derived by various convolution layer and pooling layer. Broken down into layers' outputs, learning can then be visually represented. [2] You can compare the outputs of the first and last convolution layer of the original pretrained models in Figures 35 and 36 for a test image (cf. Figure 34). As one can see in the first convolutional layer outputs, VGG16 and VGG19 are close but slightly different whereas in the last convolutional layer outputs, they are much more differentiated. ResNet50 and Xception are clearly distinct from any other model in both cases.

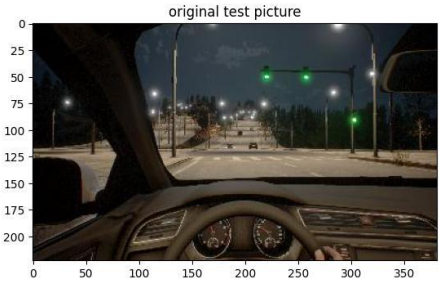

Figure 34: Sample from a test dataset.

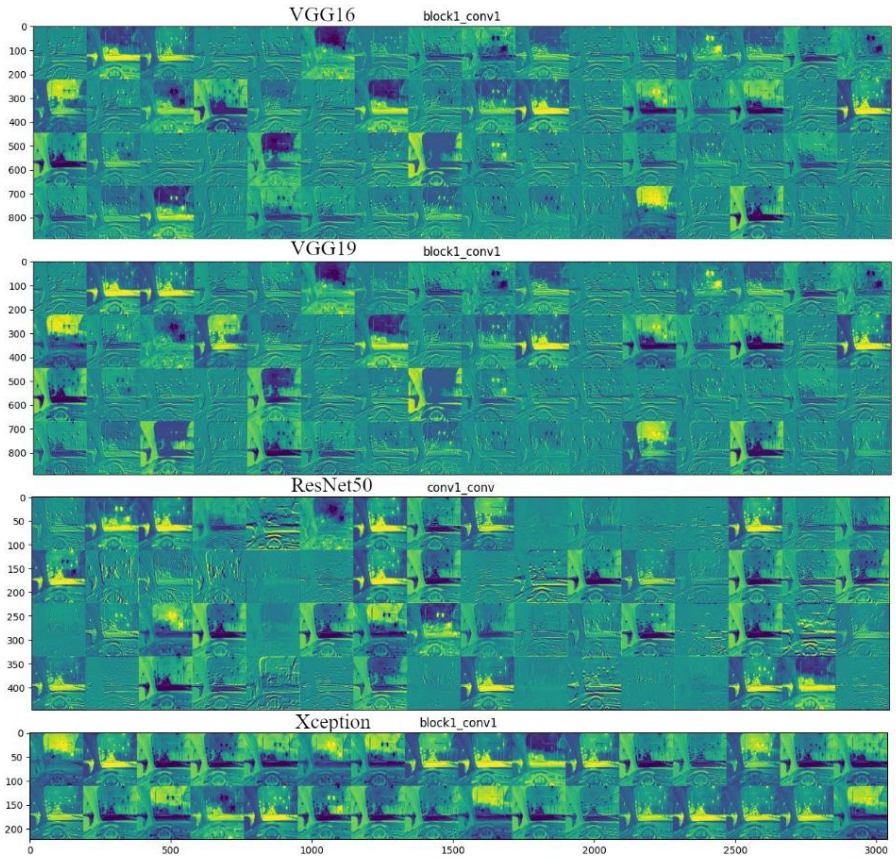

Figure 35: Visual representation of the channels of the output of the first convolution layer of the convolutional base.

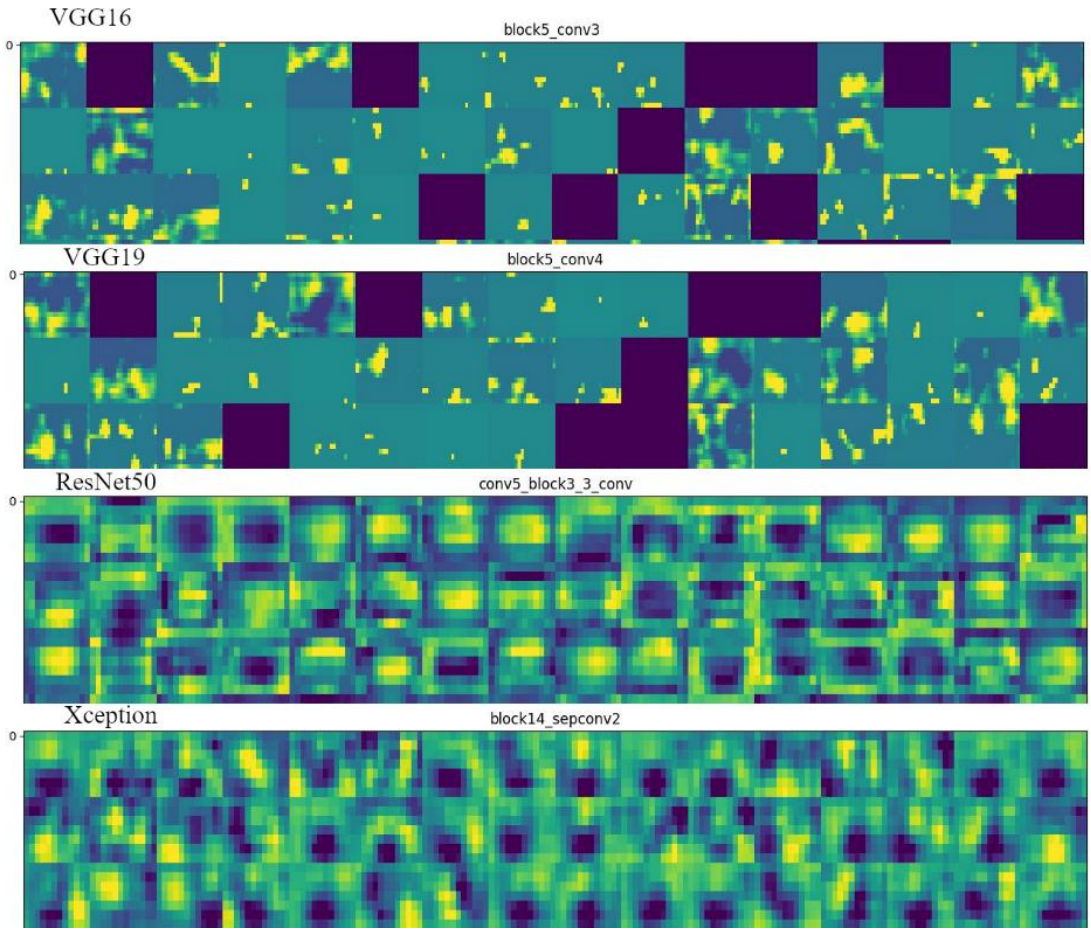

Figure 36: Visual representation of the first 16x3 channels of the output of the last convolution layer of the convolutional base (incomplete representation).

Models

As mentioned earlier, there are many ways to combine predictions. In this section, we are mixing models and use stacking to combine the predictions of the 4 multi-input models developed in the previous section. Thus, we develop a new classifier, i.e. a meta-learner that takes the predictions of the base learners as input. To develop this meta-learner, we use bagging and boosting techniques as well. The new classifiers are trained on the validation sets and tested on the test sets. We have chosen not to train them on the train sets because the classifiers take as input the predictions of our 4 multi-input models. Those predictions are obviously excellent on the training set because the models know and have been trained with those data. However, they are far from what the predictions would be on real data where chances are higher to be misclassified. That is why it is better to use the predictions gotten from the validation set, closer to what the predictions would look like on real data. It is also worth noting that the predictions used by the meta-learner are probabilities and not 0 or 1 and that we use a threshold of 0.5 to discriminate the classes.

The traditional meta-learners in stacking – and most of them have been mentioned in the theory section (cf. Concepts) - are a Multi-Layer Perceptron classifier (MLP), a Logistic Regressor (LR), a Support Vector Classifier (SVC) and a simple decision tree (DT).

Our decision tree classifier (DT) uses the predictions of the 4 models to make a final prediction. There are three functions to measure the quality of a split of a decision tree: *Gini impurity*, *log-loss* and *entropy* that both measures the information gain. The default function for decision trees is *Gini* but *log-loss* is very popular for binary classification. We have tested them all, and apparently the DT performs better with the *Gini impurity* function (cf. Figure 37). We also use balanced class weights.

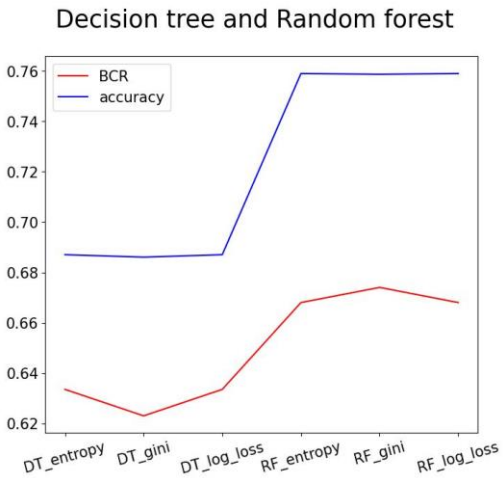

Figure 37: Hyperparameters tuning of Decision tree and Random forest.

We have used two MLPs: one based on a pre-built classifier and another that is added on top of a concatenation of the 4 multi-input models which forms a new model.

The pre-built MLP optimizes the *log-loss* function using solvers *LBFGS* (quasi-Newton methods) or a stochastic gradient descent, and one hidden layer with 100 neurons. We have tested the different solvers and it is the stochastic gradient descent *Adam* that performs the best (cf. Figure 40). *Adam* works well on dataset with a lot of training samples and *LBFGS* performs better on small dataset [44]. We have also tried two different initial learning rates - 10^{-3} and 10^{-4} - and two learning rate schedules: *constant* that applies a constant learning rate and *invscaling* that gradually decreases the learning rate using an inverse scaling exponent [44]. It turns out that an initial constant learning rate of 10^{-3} is better.

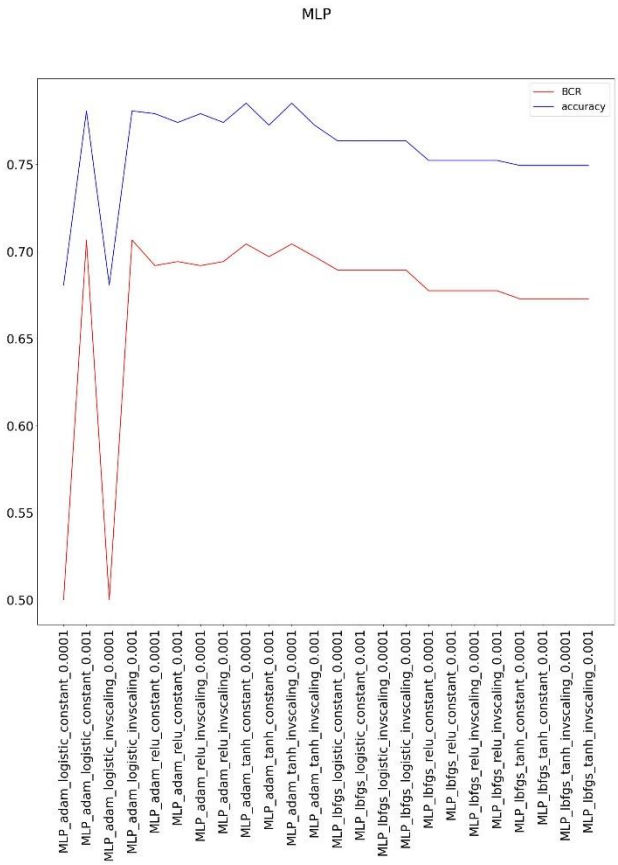

Figure 38: Hyperparameters tuning of MLP classifier.

Finally, we need an activation function for the hidden layer. We have tested three activation functions: *logistic* (*sigmoid*), *tanh* and *ReLU*. The *sigmoid* function is mostly used in binary classification because it outputs a value between 0 and 1 whereas the *tanh* function outputs a value between -1 and 1. As a reminder, the *ReLU* function outputs either a positive value, either a zero which makes the computation faster. The *ReLU* activation function is then more appropriate when there are a lot of layers because the computation time matters more than for smaller architectures. As this MLP only has one hidden layer, it is not so surprising that the logistic function performs better.

The second MLP that we are calling 'Concatenate' is added on top of a concatenation of the 4 multi-input models (cf. Appendix 1). It is composed of one hidden dense layer with 10 neurons and one output dense layer. Because the final model is big and takes a lot of time to compute, we choose the *ReLU* activation function for the hidden layer. For the output layer, the *sigmoid* activation function is the most appropriate as it is a binary classification. We train the MLP with a learning rate of 10^{-3}, a batch size of 10 and 11 epochs. We also balance the weights.

Our logistic regressor is, as we have seen before, more of a classifier because it is mostly used in classification tasks It implements regularized logistic regression using the *liblinear* library, *newton-cg*, *sag*, *saga* or *LBFGS* optimization algorithm. *Liblinear* is rather chosen for small datasets and binary classification while *sag*, *newton-cg* and *saga* suit a large dataset and multi-class classification better [47]. According to Figure 39, *liblinear* performs slightly better. Even though the dataset is not considered small, as it is probably why *Adam* was chosen versus *LBFGS* for the first MLP, it is a binary classification problem indeed and *liblinear* is appropriate for that kind of task. Without much surprise, the regularization parameter *L2* outperforms *L1*. The difference between the two is that *L1* does feature selection, in other words it removes non-interesting features which is great when we have a lot of features, not as in this case, while *L2* keeps all features until the very end [48]. We also balance the weights.

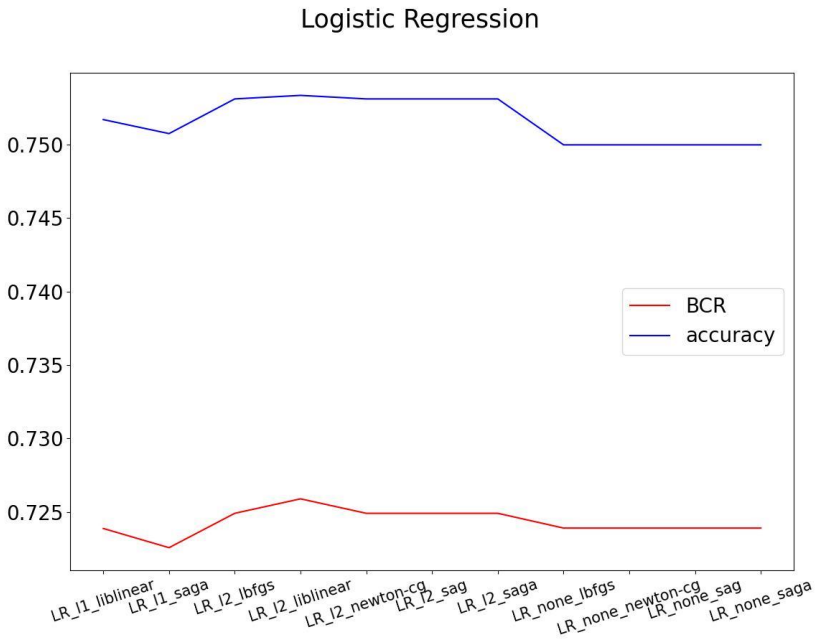

Figure 39: Hyperparameters tuning of Logistic Regressor.

The Support Vector Classifier (that we call SVM) constructs hyperplanes in the dimensional space to divide and classify the samples mapped in that space. The shape of the hyperplanes depends on the kernel function. We have tested 4 of them: *linear*, the most basic, *polynomial*, mostly used image processing, *radial basis function* (*rbf*), the most generic, and *sigmoid*, mostly used in neural networks [49]. It turns out that the best performing one is the *polynomial* function (cf. Figure 40). The class weights are also balanced.

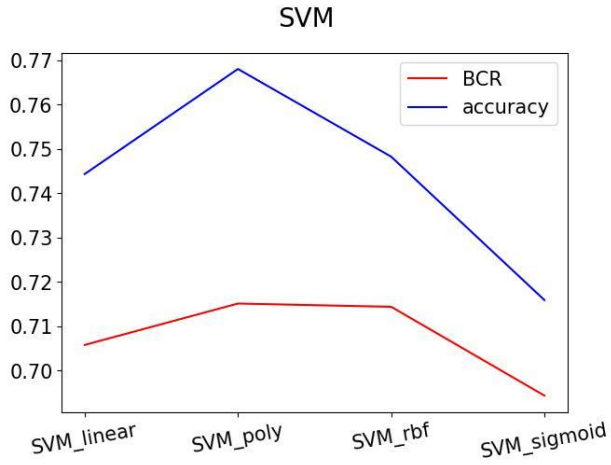

Figure 40: Hyperparameters tuning of Support Vector Classifier.

Among the bagging approaches, we have tested the random forest classifier and a pre-built bagging classifier.

Our random forest classifier (RF) fits 100 decision trees on various sub-samples and average their results. As mentioned with the decision tree classifier, there are three functions

to measure the quality of a split of a decision tree: *Gini impurity*, *log-loss* and *entropy* that both measures the information gain. The default function for RF is *Gini* but *log-loss* is very popular for binary classification. We tried them all, and apparently the random forest classifier performs better with the *log-loss* function (cf. Figure 37). We also use balanced class weights.

The bagging classifier fits ten base classifiers on random subsets of the dataset and then aggregates the predictions. The default base estimator is a decision tree but it performs poorly – a BCR around 65% - so we have tried two other classifiers as base estimators: a LR and a SVM. It turns out that a LR base estimator is the right choice (cf. Figure 41).

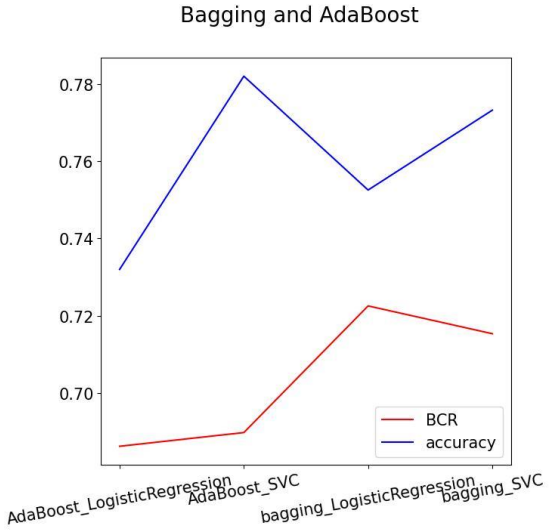

Figure 41: Hyperparameters tuning of Bagging and AdaBoost classifiers.

Then, among the boosting approaches, we have used an AdaBoost classifier and gradient boosting for classification.

The AdaBoost classifier fits a classifier on the dataset and then fits up to 50 copies of that classifier with the same data but with adjusted weights to focus on misclassified individuals from the previous classifier. The default base estimator is also a decision tree but it also performs poorly – a BCR around 66% - so we have tested two other classifiers as base estimators: a LR and a SVM. The SVM as a base estimator has the best performance (cf. Figure 41).

The GradientBoosting classifier fits at each step a regression tree on the negative gradient of the loss function, which can be *log-loss*, same as LR, or *exponential*, same as AdaBoost. To measure the quality of a split in a regression tree, we have tried two functions: *Friedman mean squared error* (MSE) and simply the *MSE* functions. The *Friedman MSE* is

supposed to be better than the simple *MSE* but we can slightly see in Figure 42 that the best performer is the GradientBoosting classifier using *log-loss* and *MSE*.

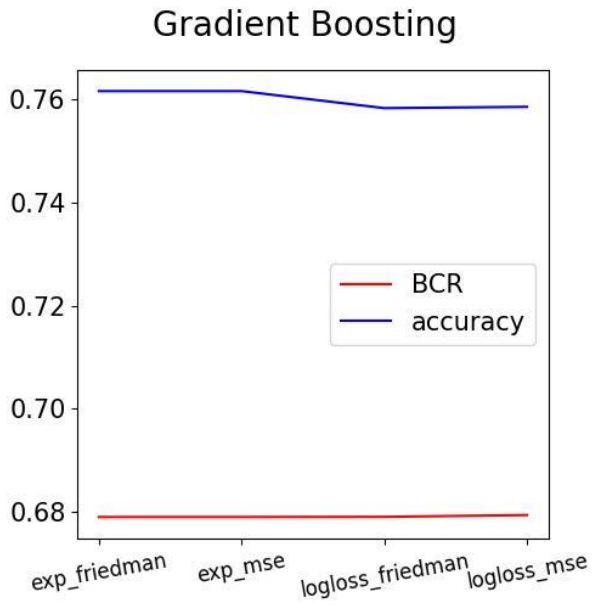

Figure 42: Hyperparameters tuning of Gradient Boosting classifier.

Finally, we have reused all of those classifiers to test two more classifiers: a voting and a stacking classifier that combine the predictions of multiple different classifiers. The voting classifier predicts class labels with majority voting. The stacking classifier uses a logistic regressor to combine the base estimators.

Results

Let's compare our classifiers (cf. Figure 43). The classifiers with the highest BCR scores are LR with 72.6%, Concatenate with 72.5% and Bagging classifier with 72.3%. The ones with the lowest BCR scores are DT with 63%, RF with 67.4% and GradientBoost with 67.9%. In terms of accuracy, the best classifiers are AdaBoost with 78.2%, voting classifier with 78.1%, MLP with 78% and stacking classifier with 77.8%, while the worst ones are DT with 68.7% and Concatenate with 74%. It is interesting to note that good performing models in terms of BCR are not especially good models in terms of accuracy. The metrics do not seem to be correlated. Considering accuracy and BCR, DT is the worst performing classifier but it is impossible to choose a best classifier that is excellent in both metrics.

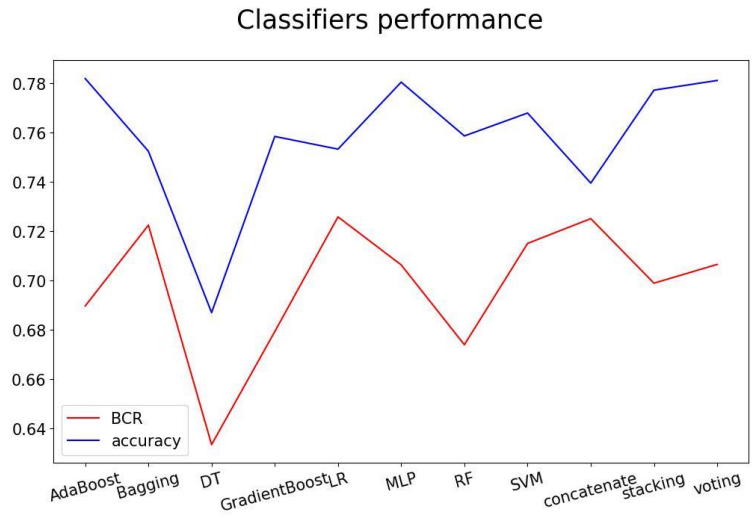

Figure 43: Comparison of ensemble classifiers performance.

We have seen previously that VGG16 and VGG19 are quite close in terms of the visual concepts that they learn, compared to how differentiated ResNet50 and Xception are. It could then be interesting to see if there is a loss of information that would reduce the performance if one of them is missing. We have decided to test the classifiers without VGG16 predictions because it is the smallest model. It turns out that there is a loss of information and a degradation of performance (cf. Figure 44), which means that VGG16 and VGG19 are indeed complementary. The best model LR goes from a BCR of 72.6% to 71.3%, SVM from 71.5% to 69.7%, MLP from 70.6% to 69.9%, etc.

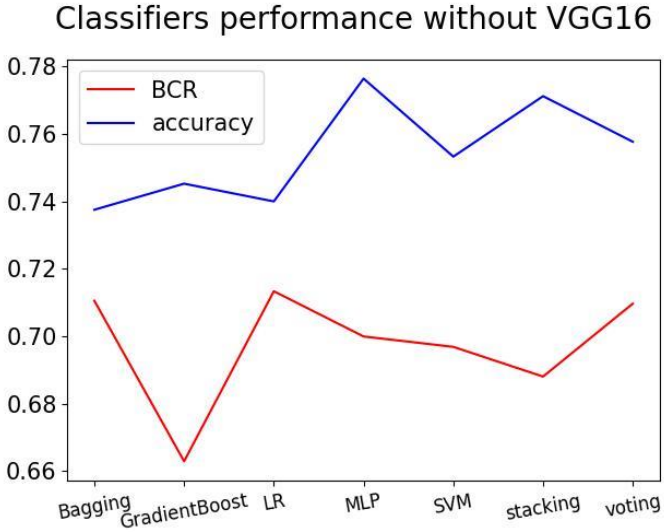

Figure 44: Comparison of the classifiers performance without the input of VGG16.

If we compare the BCRs of the ensemble models and the previous ones (cf. Table 2), four ensemble classifiers out of eleven outperform all the others: LR, Concatenate, Bagging and SVM. Except for DT, all the ensemble models outperform ResNet50 and Xception. Now regarding accuracies, four ensemble classifiers outperform all the others: AdaBoost, MLP, Voting and Stacking. Except for DT and Concatenate, all the other ensemble models outperform ResNet50 and Xception.

	Best BCR score	Best accuracy score
VGG16	71.2% (step 2)	77.3% (step 2)
VGG19	71.4% (step 1b)	77.1% (step 2)
ResNet50	66.2% (step 2)	73.8% (step 1b)
Xception	67.3 (step 2)	75.1% (step 1b)
Ensembles	72.6% (step 3)	78.2% (step 3)

Table 2: Comparison of the best BCR scores among the 3 steps.

Looking at the Figure 45 retracing the performance at the different steps, it is pretty obvious that ensemble learning has better scores than a single model at any of the previous step.

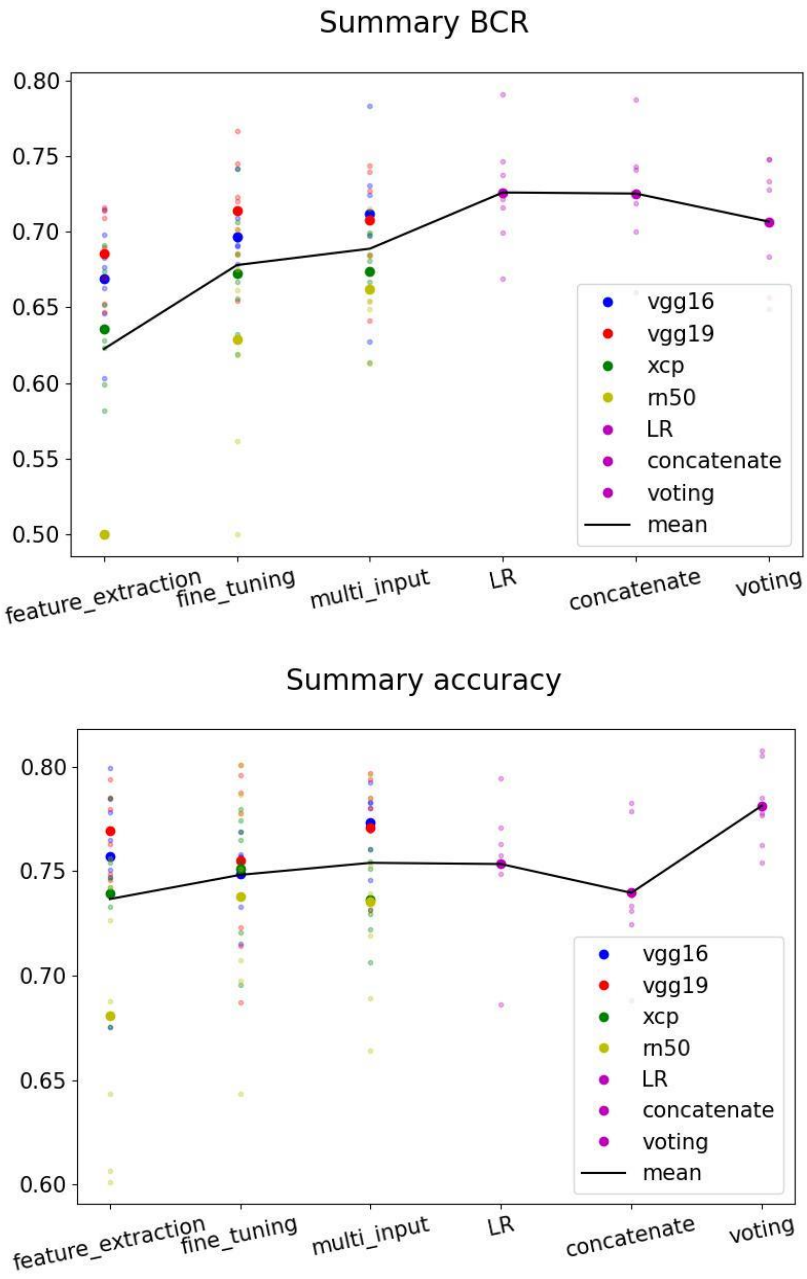

Figure 45: Comparison of the scores of all 3 steps.

Conclusion

Even though VGG16 and VGG19 share the same architecture, they still learn differently and have complementary information. They are not as differentiated as ResNet50 and Xception can be but it still worth bringing them together in an ensemble.

Given the observations, we can conclude that ensemble learning performs better that a single model for our classification task. The best model regarding BCR scores is an ensemble model with a LR classifier on top, and regarding accuracy it is an ensemble model with an AdaBoost classifier. However, it is impossible to choose a best model for both metrics because

none of them are excellent in both metrics, which makes sense because the cost of correctly classifying an extra small amount of under-represented samples might often be to incorrectly classify a bigger amount of samples of the dominant class that were previously correctly classified.

One result that was not surprising is the lower score of DT. It is the simplest classifier, so it was not expected to perform well. By contrast, the voting classifier slightly outperforming the stacking classifier was puzzling for several reasons. First, the LR is the best performing classifier when used on four predictions of the multi-input models, but yet it cannot accurately predict the label given more information, i.e. the predictions of 9 other classifiers. Second, the definition of stacking says that it makes better predictions than simply voting (cf. Concepts). The boosting was quite disappointing as well in terms of BCRs. The fact that GradientBoost does not perform too well is not so surprising because it uses a simple tree regressor. AdaBoost on the other hand reaches an outstanding accuracy with a disappointing BCR. One can draw the conclusion that AdaBoost does not suit a class imbalance problem well.

Conclusion

To conclude this master thesis, we would like to come back on our research objective to rate the success of our research. As a reminder, the point of this work is to develop a model able to predict effectively whether or not a vehicle needs to brake based on the driver's frontal point of view, speed and steering angle. It is a binary classification task including a data class imbalance, image classification and a multi-input classification.

The success in the research objective is moderate. In terms of developing a predictive model, we have reached our goal because the BCR scores are higher than what we would expect if the samples were classified randomly, i.e. a BCR of 50%. However, our reservation lies where the rubber meets the road, in the development of an *effective* predictive model. It depends of course on how *effective* is defined. On average, the BCR and accuracy have increased at each development step (cf. Figure 45). In that perspective, at least the development has been *effective*. On the other hand, if we look at the order of magnitude given by the literature, *effectiveness* can be defined as accuracy or equivalent scores around 90%. We are quite far below as our best BCR score lies at 72.6% and our highest accuracy at 78.2%. We would have expected to reach a higher score with ensemble learning. Nevertheless, there are definitely conclusions to be drawn from trying to explain those disappointing figures step by step.

Feature extraction is the least performing modelling approach. It is to be expected as it is the simplest approach. Even though we have tried to tune several hyperparameters, we could have tried more combinations of hidden layers and their parameters such as the dropout rate or the output size. That may have improved the performance of the models. It is important to note however that training a model with many layers on a large dataset takes so much time that it would take forever to try every combination and every hyperparameter. There is a trade-off between finding the perfect values for the hyperparameters and time. Decisions have to be made regarding what hyperparameters to tune and with which values.

The fine-tuned models perform way better, which makes sense because fine-tuning adjusts the last layers of the convolutional base to the current problem. It should be expected that the model performs better in a classification problem if it fits more specifically to that problem. However, we have tested to deeper fine-tune our model and it did not improve the performance. That has gotten us thinking that we have not found the right parameters to make it improve the performance. But because the literature does not advise to deeper fine-tune the model and claims that it is not worth the time, we have decided to not dig deeper on that side.

We have nevertheless explored other parameters such as adjusting the class weights, which seems to bring significant added value in a class imbalance context. However, class weights can take any values and there is no guarantee that a particular class weights set of value is the optimal one. It could require a lot of training to find this optimal set of value.

Talking about class imbalance, we have also seen the difference between BCR and accuracy, and illustrated the importance of monitoring BCR scores rather than accuracies to ensure that both class are considered in the predictions.

Adding speed and steering angle as numerical input to the model has not improved the performance as much as we would have expected. It could even have created some noise in VGG19 that is less performing with the numerical input. It looks like the speed and steering angle are not more relevant than other features extracted from the visual input. The point of extracting 10 features out of two via an MLP is make sure that they would find their place in the original set of features. However, looking back on my multi-input strategy, I think I may have harmed chances to succeed in this endeavor by creating too much noise in the numerical inputs via an MLP with two dropout layers and hidden dense layer with 256 and 128 outputs for only 2 inputs. Another strategy that could have helped in that objective would have been to select the most relevant features before passing them into the top classifiers so that the set of features would be smaller and less noisy. That could have been done via a principal component analysis for instance.

Ensemble learning is known to boost performance scores and it has been a success. Nevertheless, we would have expected it to give a bigger boost to the performance scores. They have only increased by around 1 or 2%. The best ensemble learning classifier in terms of BCR is a logistic regressor, a form of stacking. We have noticed that given our model, it was not possible to elect a best model in terms of both BCR and accuracy scores. We have also seen that the four models are indeed complementary as an ensemble, despite VGG16 and VGG19 sharing similar architectures. To have bigger boost in performance, the four models should maybe first be further improved individually.

Another thing worth mentioning is that throughout the entire journey, VGG16 and VGG19 have outperformed Xception and ResNet50. However, it does not make sense because Xception and ResNet50 are more recent and more complex and usually score higher than the VGG architecture. But as said previously, there is a trade-off between the amount of tuning

that can be applied and the time it takes. Because they are bigger and more complex model, they do take a lot more time to train so maybe they could have been tuned better.

Regarding the tuning, we are not totally sure that we have made the right decision regarding the choice of the best performing parameters because there was some randomness in the training so no two trainings give the same results. When testing the hyperparameter, the values were very close (cf. Appendix) so it was hard to choose which value was the best as there was no guarantee that it was actually the best choice and not a result from the randomness. We have used K-fold cross validation to alleviate that issue and have the true performance of our models but even so, the values are very close.

Overall, the accuracy is close to 80% and the BCR is above 70%. This is sufficiently better than random predictions to say that we have filled the brief and achieved our research objective. We have a lot of ideas for future work however.

Future work

For future work, a lot of things can be undertaken to improve the predictions.

Firstly, we have restricted ourselves to the driver's frontal point of view, speed and steering angle. It does not have to be that way. *Unreal* allows to add sensors on the vehicles so it would be possible to have new types of data such as LIDAR point cloud, and build for each vehicle a tridimensional representation of its environment for instance. Talking about tridimensional representation, one could also add the driver's side and rear point of views and do some multi-view classification, with potentially some ensemble learning. As seen in the literature, it is not uncommon to combine multi-view classification and ensemble learning as those are close concepts.

Secondly, to improve the models in the future, one can obviously try to tune other hyperparameters or other values. It would also be interesting to try other CNNs, more recent and more complex such as faster R-CNN that we have discussed in the literature. There are also other ensemble techniques that we have not tried yet such as training the same model on specific parts of the dataset. One could train a model on subsets that dominantly belong to one state - *stopping at traffic light*, *going straight*, etc. – and aggregate their knowledge afterwards.

Thirdly, it was mentioned in the literature that we could use temporality for classification with a Long Short-Term Memory architecture. One could also use Reinforcement learning.

And finally, we have worked on a dataset at nighttime, but it might be interesting to see how well the models perform on a dataset at daytime and on a mixed dataset. Regarding the dataset, it might also be relevant to see how the preprocessing impact the performance such as the size of the picture, the color space, etc.

References

[1] Rashmi, B, C., Ramani, D. M., & Harsur, M. S. (2022). CNN based multi-view classification and ROI segmentation: A survey. *Global Transitions Proceedings.* https://doi.org/10.1016/j.gltp.2022.04.019

[2] Chollet, F. (2017). *Deep learning with python.* Manning Publications.

[3] Elallid, B. B., Benamar, N., Hafid, A. S., Rachidi, T., & Mrani, N. (2022). A comprehensive survey on the application of deep and reinforcement learning approaches in autonomous driving. *Journal of King Saud University - Computer and Information Sciences.* https://doi.org/10.1016/j.jksuci.2022.03.013

[4] Song, Z., Fu, L., Wu, J., Liu, Z., Li, R., & Cui, Y. (2019). Kiwifruit detection in field images using Faster R-CNN with VGG16. *IFAC-PapersOnLine*, *52*(30), 76–81. https://doi.org/10.1016/j.ifacol.2019.12.500

[5] Kong, L., & Cheng, J. (2022). Classification and detection of COVID-19 X-Ray images based on DenseNet and VGG16 feature fusion. *Biomedical Signal Processing and Control*, *77*(103772), 103772. https://doi.org/10.1016/j.bspc.2022.103772

[6] Zhao, Y., Guo, S., Wang, Y., Cui, J., Ma, Y., Zeng, Y., Liu, X., Jiang, Y., Li, Y., Shi, L., & Xiao, N. (2019). A CNN-based prototype method of unstructured surgical state perception and navigation for an endovascular surgery robot. *Medical & Biological Engineering & Computing*, *57*(9), 1875–1887. https://doi.org/10.1007/s11517-019-02002-0

[7] Sunil, Zhang, Y., Koparan, C., Ahmed, M. R., Howatt, K., & Sun, X. (2022). Weed and crop species classification using computer vision and deep learning technologies in greenhouse conditions. *Journal of Agriculture and Food Research*, *9*(100325), 100325. https://doi.org/10.1016/j.jafr.2022.100325

[8] Seeland, M., & Mäder, P. (2021). Multi-view classification with convolutional neural networks. *PloS One*, *16*(1), e0245230. https://doi.org/10.1371/journal.pone.0245230

[9] Guerin, J., Thiery, S., Nyiri, E., Gibaru, O., & Boots, B. (2021). Combining pretrained CNN feature extractors to enhance clustering of complex natural images. In *arXiv [cs.CV]*. http://arxiv.org/abs/2101.02767

[10] *Winning solutions of kaggle competitions.* (2022, July 24). Kaggle.com; Kaggle. https://www.kaggle.com/code/sudalairajkumar/winning-solutions-of-kaggle-competitions/notebook

[11] Narin, A., Kaya, C., & Pamuk, Z. (2021). Automatic detection of coronavirus disease (COVID-19) using X-ray images and deep convolutional neural networks. *Pattern Analysis and Applications: PAA, 24*(3), 1207–1220. https://doi.org/10.1007/s10044-021-00984-y

[12] Gupta, A., Anpalagan, A., Guan, L., & Khwaja, A. S. (2021). Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues. *Array, 10*(100057), 100057. https://doi.org/10.1016/j.array.2021.100057

[13] Chen, D., & Lv, Z. (2022). Artificial intelligence enabled Digital Twins for training autonomous cars. *Internet of Things and Cyber-Physical Systems, 2,* 31–41. https://doi.org/10.1016/j.iotcps.2022.05.001

[14] Bachute, M. R., & Subhedar, J. M. (2021). Autonomous driving architectures: Insights of machine learning and deep learning algorithms. *Machine Learning with Applications, 6*(100164), 100164. https://doi.org/10.1016/j.mlwa.2021.100164

[15] Li, G., Zong, C., Liu, G., & Zhu, T. (2020). Application of convolutional neural network (CNN)–AdaBoost algorithm in pedestrian detection. *Sensors and Materials, 32*(6), 1997. https://doi.org/10.18494/sam.2020.2787

[16] Wu, Q., Gao, T., Lai, Z., & Li, D. (2020). Hybrid SVM-CNN classification technique for human-vehicle targets in an automotive LFMCW radar. *Sensors (Basel, Switzerland), 20*(12), 3504. https://doi.org/10.3390/s20123504

[17] Hu, J., Abubakar, S., Liu, S., Dai, X., Yang, G., & Sha, H. (2019). Near-infrared road-marking detection based on a modified faster regional convolutional neural network. *Journal of Sensors, 2019,* 1–11. https://doi.org/10.1155/2019/7174602

[18] Mu, G., Xinyu, Z., Deyi, L., Tianlei, Z., & Lifeng, A. (2015). Traffic light detection and recognition for autonomous vehicles. The Journal of China Universities of Posts and Telecommunications, 22(1), 50–56. https://doi.org/10.1016/s1005-8885(15)60624-0

[19] Chen, X., Gao, W., Hong, C., & Tu, Y. (2022). A novel series arc fault detection method for photovoltaic system based on multi-input neural network. *International Journal of*

Electrical Power & Energy Systems, 140(108018), 108018. https://doi.org/10.1016/j.ijepes.2022.108018

[20] Apostolopoulos, I. D., Apostolopoulos, D. I., Spyridonidis, T. I., Papathanasiou, N. D., & Panayiotakis, G. S. (2021). Multi-input deep learning approach for Cardiovascular Disease diagnosis using Myocardial Perfusion Imaging and clinical data. *Physica Medica: PM: An International Journal Devoted to the Applications of Physics to Medicine and Biology: Official Journal of the Italian Association of Biomedical Physics (AIFB), 84*, 168–177. https://doi.org/10.1016/j.ejmp.2021.04.011

[21] Aguiar, G., Krawczyk, B., & Cano, A. (2022). A survey on learning from imbalanced data streams: taxonomy, challenges, empirical study, and reproducible experimental framework. In *arXiv [cs.LG]*. http://arxiv.org/abs/2204.03719

[22] Krawczyk, B. (2016). Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence, 5*(4), 221–232. https://doi.org/10.1007/s13748-016-0094-0

[23] Branco, P., Torgo, L., & Ribeiro, R. (2015). A survey of predictive modelling under imbalanced distributions. In *arXiv [cs.LG]*. http://arxiv.org/abs/1505.01658

[24] *Adam*. (n.d.). Keras.Io. Retrieved August 18, 2022, from https://keras.io/api/optimizers/adam/

[25] *Tf.Keras.Losses.BinaryFocalCrossentropy*. (n.d.). TensorFlow. Retrieved August 18, 2022, from https://www.tensorflow.org/api_docs/python/tf/keras/losses/BinaryFocalCrossentropy

[26] *Tfa.Losses.SigmoidFocalCrossEntropy*. (n.d.). TensorFlow. Retrieved August 18, 2022, from https://www.tensorflow.org/addons/api_docs/python/tfa/losses/SigmoidFocalCrossEntropy

[27] *Python*. (n.d.). Python.org. Retrieved August 19, 2022, from https://www.python.org/

[28] *About keras*. (n.d.). Keras.Io. Retrieved August 18, 2022, from https://keras.io/about/

[29] *Install TensorFlow 2*. (n.d.). TensorFlow. Retrieved August 18, 2022, from https://www.tensorflow.org/install

[30] Kumar, A., & Jain, M. (2020). *Ensemble learning for AI developers: Learn bagging, stacking, and boosting methods with use cases* (1st ed.). APress.

[31] *ImageNet*. (n.d.). Image-net.org. Retrieved August 18, 2022, from https://image-net.org/about

[32] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. In *arXiv [cs.CV]*. http://arxiv.org/abs/1409.1556

[33] *ImageNet Winning CNN Architectures (ILSVRC)*. (n.d.). Kaggle.com. Retrieved August 18, 2022, from https://www.kaggle.com/getting-started/149448

[34] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[35] Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

[36] Dupont, P. (2022). *Assignment 5* [inginious task]. EPL, UCLouvain. https://inginious.info.ucl.ac.be/course/LINFO2262/A5

[37] Sharma, P., Yadav, U., & Sharma, D. (2009). The concept of sensitivity and specificity in relation to two types of errors and its application in medical research. *Journal of Reliability and Statistical Studies, 2*(2), 53-58.

[38] *1.11. Ensemble methods*. (n.d.). Scikit-Learn. Retrieved August 18, 2022, from https://scikit-learn.org/stable/modules/ensemble.html

[39] *1.1. Linear models*. (n.d.). Scikit-Learn. Retrieved August 18, 2022, from https://scikit-learn.org/stable/modules/linear_model.html

[40] *1.4. Support vector machines*. (n.d.). Scikit-Learn. Retrieved August 18, 2022, from https://scikit-learn.org/stable/modules/svm.html

[41] *1.17. Neural network models (supervised)*. (n.d.). Scikit-Learn. Retrieved August 18, 2022, from https://scikit-learn.org/stable/modules/neural_networks_supervised.html

[42] *Unreal Engine*. (n.d.). Unreal Engine. Retrieved August 18, 2022, from https://www.unrealengine.com/en-US

[43] Kızrak, A. (2019, May 9). *Comparison of Activation functions for deep neural networks*. Towards Data Science. https://towardsdatascience.com/comparison-of-activation-functions-for-deep-neural-networks-706ac4284c8a

[44] *Sklearn.Neural_network.MLPClassifier*. (n.d.). Scikit-Learn. Retrieved August 18, 2022, from https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

[45] *Tf.Keras.Losses.BinaryCrossentropy*. (n.d.). TensorFlow. Retrieved August 18, 2022, from https://www.tensorflow.org/api_docs/python/tf/keras/losses/BinaryCrossentropy

[46] *Model training APIs*. (n.d.). Keras.Io. Retrieved August 18, 2022, from https://keras.io/api/models/model_training_apis/

[47] *Sklearn.Linear_model.LogisticRegression*. (n.d.). Scikit-Learn. Retrieved August 18, 2022, from https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[48] Arora, A. (2021, November 4). *Quickly master L1 vs L2 regularization - ML interview Q&A*. Analytics Arora. https://analyticsarora.com/quickly-master-l1-vs-l2-regularization-ml-interview-qa/

[49] *SVM Kernel Function*. (N.d.). Pythongeeks.org. Retrieved August 18, 2022, from https://pythongeeks.org/svm-kernel-function/#:~:text=Introduction%20to%20SVM%20Kernel%20Function&text=The%20kernel%20performs%20the%20task,to%20surpass%20complexities%20in%20calculations.

Appendices

Appendix 1 : MLP classifier

Figure 46: MLP classifier on top of the concatenation of 4 models.

Appendix 2: Feature extraction

	type	model	mean bcr	mean acc
7	0.0001_10_30	rn50	0.500000	0.655600
15	0.0001_20_10	rn50	0.500000	0.655600
23	0.0001_20_30	rn50	0.500000	0.655600
31	0.0001_50_10	rn50	0.500000	0.673732
51	0.0001_50_30	rn50	0.500000	0.601202
3	0.001_10_30	rn50	0.500000	0.655600
11	0.001_20_10	rn50	0.500000	0.655600
19	0.001_20_30	rn50	0.500000	0.644720
27	0.001_50_10	rn50	0.500000	0.655600
39	adding	rn50	0.500000	0.655600
43	focal	rn50	0.500000	0.655600
47	sfocal	rn50	0.500000	0.655600
35	tanh	rn50	0.500000	0.655600
4	0.0001_10_30	vgg16	0.673955	0.773792
12	0.0001_20_10	vgg16	0.676809	0.764164
20	0.0001_20_30	vgg16	0.671280	0.765305
28	0.0001_50_10	vgg16	0.690596	0.768964
36	adding	vgg16	0.642004	0.757005
40	focal	vgg16	0.651216	0.767287
44	sfocal	vgg16	0.605950	0.750119
32	tanh	vgg16	0.674662	0.761092
5	0.0001_10_30	vgg19	0.646273	0.759211
13	0.0001_20_10	vgg19	0.653237	0.765079
21	0.0001_20_30	vgg19	0.662324	0.768153
29	0.0001_50_10	vgg19	0.661478	0.773186
1	0.001_10_30	vgg19	0.641072	0.764189
9	0.001_20_10	vgg19	0.605154	0.743586
17	0.001_20_30	vgg19	0.615043	0.752962
25	0.001_50_10	vgg19	0.633484	0.760495
37	adding	vgg19	0.633849	0.750179
41	focal	vgg19	0.649421	0.749856
45	sfocal	vgg19	0.588081	0.742099
33	tanh	vgg19	0.661370	0.756149
6	0.0001_10_30	xcp	0.660100	0.752075
14	0.0001_20_10	xcp	0.677737	0.751187
22	0.0001_20_30	xcp	0.696756	0.756885
30	0.0001_50_10	xcp	0.665041	0.737405
2	0.001_10_30	xcp	0.630825	0.737851
10	0.001_20_10	xcp	0.607449	0.719833
18	0.001_20_30	xcp	0.639485	0.727888
26	0.001_50_10	xcp	0.607983	0.708858
38	adding	xcp	0.643043	0.732406
42	focal	xcp	0.650466	0.733273
46	sfocal	xcp	0.623689	0.730900
34	tanh	xcp	0.630054	0.727505

The table on the top refers to the different aspects tested in the step 1-feature extraction.

Sfocal and focal refers to the loss functions Sigmoid Focal Cross Entropy or Binary Focal Cross entropy.

Adding refers to adding a hidden layer.

Class weight refers to the opposite of what was chosen. If the model uses balanced weight, class weight refer to test where the model was trained without it.

Tanh means that *ReLU* was replaced by *tanh* as an activation function in the hidden layer.

0.0001_10_30 refers to a test with a learning rate of 10^{-4}, 10 epochs and a batch size of 30.

To compare, the table below shows the scores of the chosen hyperparameters.

	BCR	Accuracy
VGG16	66.9%	75.7%
VGG19	68.5%	76.9%
ResNet50	50%	68%
Xception	63.5%	73.9%

Appendix 3: Fine tuning

type	model	mean bcr	mean acc
class weight	rn50	0.622883	0.713912
conv4_block6_out	rn50	0.506917	0.682090
0.0001_10_10	vgg16	0.709276	0.765284
0.0001_20_10	vgg16	0.711313	0.764408
0.0001_20_50	vgg16	0.703689	0.767706
block3_pool	vgg16	0.659223	0.734191
class weight	vgg16	0.687172	0.764525
0.0001_10_10	vgg19	0.693028	0.755399
0.0001_20_10	vgg19	0.693377	0.761132
0.0001_20_50	vgg19	0.690646	0.770409
block3_pool	vgg19	0.615029	0.664533
class weight	vgg19	0.688060	0.758598
add_10	xcp	0.653695	0.735903
class weight	xcp	0.670333	0.735730

The table on the top refers to the different aspects tested in the step 1-fine tuning.

Block3_pool refers to retraining from that layer.

Class weight refers to the opposite of what was chosen. If the model uses balanced weight, class weight refer to test where the model was trained without it.

0.0001_10_30 refers to a test with a learning rate of 10^{-4}, 10 epochs and a batch size of 30.

To compare, the table below shows the scores of the chosen hyperparameters.

	BCR	Accuracy
VGG16	69.7%	74.9%
VGG19	71.4%	75.5%
ResNet50	62.9%	73.8%
Xception	67.2%	75.1%

Appendix 4: Multi-input

type	model	mean bcr	mean acc
0.0001_10_30	rn50	0.618144	0.745145
0.0001_11_10	rn50	0.631982	0.751073
0.0001_11_50	rn50	0.638200	0.755260
adding	rn50	0.637827	0.746864
class weight	rn50	0.637621	0.730410
tanh	rn50	0.628318	0.750483
0.0001_10_30	vgg16	0.710902	0.756980
0.0001_11_10	vgg16	0.697458	0.754064
0.0001_11_50	vgg16	0.699441	0.756520
adding	vgg16	0.695636	0.752623
class weight	vgg16	0.654789	0.748121
tanh	vgg16	0.700353	0.752859
0.0001_10_30	vgg19	0.712691	0.759896
0.0001_11_10	vgg19	0.710209	0.759334
0.0001_11_50	vgg19	0.711756	0.757437
adding	vgg19	0.708817	0.755148
class weight	vgg19	0.663124	0.759874
tanh	vgg19	0.711767	0.758281
0.0001_10_30	xcp	0.674091	0.751072
0.0001_11_10	xcp	0.675195	0.749455
0.0001_11_50	xcp	0.670780	0.749572
adding	xcp	0.676006	0.752173
class weight	xcp	0.633826	0.743118
tanh	xcp	0.674661	0.751904

The table on the left refers to the different aspects tested in the step 2.

Adding refers to adding a hidden layer.

Class weight refers to the opposite of what was chosen. If the model uses balanced weight, class weight refer to test where the model was trained without it.

Tanh means that *ReLU* was replaced by *tanh* as an activation function in the hidden layer.

0.0001_10_30 refers to a test with a learning rate of 10^{-4}, 10 epochs and a batch size of 30.

To compare, the table below shows the scores of the chosen hyperparameters.

	BCR	Accuracy
VGG16	71.2%	77.3%
VGG19	70.8%	77.1%
ResNet50	66.2%	73.5%
Xception	67.3%	73.6%

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**