CNN-based Classification of Car Images for Android Devices

Iveta Mrázová[1], Georgi Georgiev[2]

[1]Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic
[2]Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Abstract

The design of efficient yet robust methods for real-time image classification belongs to scorching topics in contemporary AI, particularly in the case of mobile and edge devices. Various types of convolutional neural networks seem to contribute to solving this task. Especially those architectures proposed explicitly for mobile devices, e.g., MobileNet, and EfficientNet, are classified to the least time-consuming ones. This paper thoroughly reviews the structure, performance, and main characteristics of the considered network types. Based on the obtained results, we introduce a mobile-phone application to classify cars we might see on the street and search for nearby car dealerships, e.g., to buy a car similar to that one of interest. The developed application involves the TensorFlow EfficientNet Lite model. Finally, we provide an outlook for a possible enhancement of the application with federated learning.

Keywords

image classification, convolutional neural networks, EfficientNet, TensorFlow Lite models for mobile applications

1. Introduction

Modern convolutional neural networks (CNNs) are known to beat human performance in many tasks. However, their state-of-the-art architectures require substantial computational resources. A pretty natural question thus arises if we can also benefit from CNN image processing capabilities when implemented on mobile devices. Recent Android products range from the Google Pixel 6 mobile phone equipped with the newest Google Tensor processor to mobile devices that use Edge TPU (Tensor Processing Unit) chip. Two examples of Edge TPU devices include Coral Dev Board and the Coral USB accelerator.

Although the CNNs comprise a considerable number of neurons at different layers, the model benefits from weight sharing that keeps down the number of trainable parameters. With local receptive fields (i.e., rectangular filters), the CNNs scan the presented images to look for significant visual pattern features. This information is combined in subsequent layers to detect more complex higher-order features. The neurons' activities form the so-called feature maps representing the extracted knowledge in each layer. Alternating pooling layers blur the exact position of the features and allow for down-sampling of feature maps.

Our ultimate objective is to develop a mobile-phone application to classify cars we might see and search for dealerships to rent or buy a similar car. Fig. 1 presents a snapshot illustrating the function of the application.

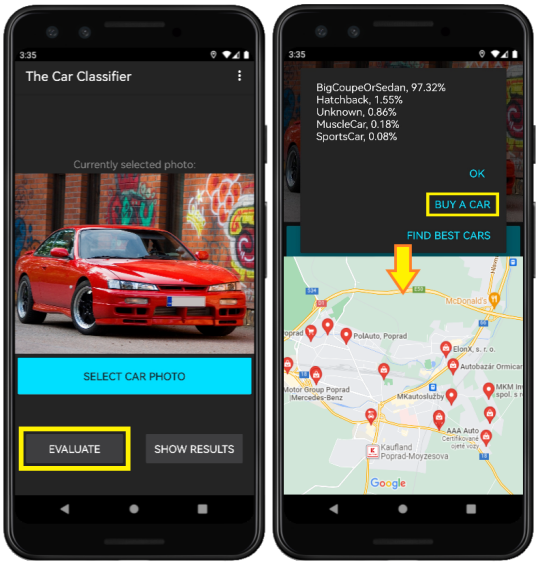

Figure 1: A snapshot of the application running on the Pixel 3 virtual mobile device: the classification of a presented sports car is followed by searching for the closest dealerships offering similar cars in Poprad.

Considering the limited hardware means of mobile devices, a crucial steppingstone in the application design represents the choice of an accurate, robust, and memory/time efficient network model for the CNN-based car classifier.

As a part of our research, we tested the classification and robustness performance of 10 selected CNN models. The results indicate that the EficcientNet models are superior in all cases. More precisely, EfficientNetB5,

ITAT'22: Information technologies – Applications and Theory, September 23–27, 2022, Zuberec, Slovakia
iveta.mrazova@mff.cuni.cz (I. Mrázová);
Georgi.Georgiev.99@seznam.cz (G. Georgiev)
0000-0002-3765-1400 (I. Mrázová)

Figure 2: Examples for car classes from the used dataset. Row-wise from left to right: BigCoupeOrSedan, Hatchback, MuscleCar, PickUp, Van, SportsCar, SUV, Unknown.

with its 108MB TensorFlow Lite version, fits perfectly into an Android application, and its accuracy of 75.7% also outperforms larger models like InceptionResNetV2 (75.5%). Lastly, EfficientNetB5 shows outstanding robustness results. The only model able to beat EfficientNetB5 is EfficientNetB7, which unfortunately does not fit into our application. As a result, we shall prioritize EfficientNetB5.

The Android application we have developed thus uses EfficientNetB5 to classify cars directly on a mobile device. After classification, the application allows the user to search for nearby dealerships that provide vehicles of the same type as the classification result. We see an additional benefit of our research in developing a CNN-based application for Android devices as a proof of concept for solving other image classification and machine learning tasks.

Various CNN architectures target mobile devices, e.g., MobileNet, EfficientNet, and others. Section 2 depicts their main characteristics and briefly overviews the mobile devices under question. Section 3 discusses the design of a mobile-phone application we have developed to classify cars the user sees and might want to rent or buy. Section 4 evaluates the performance of the considered networks concerning their time and space efficiency, accuracy, and robustness to noise and various image transformations. The concluding section summarizes the obtained results, focusing on the TensorFlow EfficientNet Lite model, and provides an outlook for a possible enhancement of the application with the so-called federated learning.

2. Related Work

To find the best model satisfying the above-specified requirements, we have selected 10 candidate network models: MobileNetV2, EfficientNetB0, EfficientNetB5, EfficientNetB7, NASNetMobile, NASNetLarge, InceptionV3, Xception, InceptionResNetV2, and DenseNet121. Their TensorFlow Lite versions are publicly available for all of them and together, they provide a wide variety of architectures (large and small) applicable to image classification. For the chosen models, their size characteristics are summarized in Table 1. Depth refers to the topological depth of the respective model inclusive its activation layers, batch normalization layers etc. [11].

2.1. Network Architectures For Mobile Devices

The historically first model, InceptionV3 [15], introduced the concept of modules consisting of several convolutions with different sizes operating on the same level. The concatenation layer puts together the information gathered from all the convolutions at the end of each module. This

Table 1
Size characteristics of the considered CNNs

Model	Depth	Parameters (M)
MobileNetV2	105	**3.5**
EfficientNetB0	132	5.3
EfficientNetB5	312	30.6
EfficientNetB7	438	66.7
NASNetMobile	389	5.3
NASNetLarge	533	88.9
InceptionV3	189	23.9
Xception	**81**	22.9
InceptionResNetV2	449	55.9
DenseNet121	242	8.1

The data comes from the Keras Application page of the official Keras documentation [11].

model is characterized by the depth of 189 and 23.9 million parameters. InceptionV3 achieved the best results for its time, but today it gets easily outperformed even by smaller models.

The InceptionResNet model replaces the concatenations of InceptionV3 with residual connections skipping the layers [16]. Inserting such shortcuts improves the network's ability to back-propagate errors across multiple layers. InceptionResNetV2 has the depth of 449 layers and 55.9 million parameters which makes it one of the biggest models we dealt with. In this case the size of the model and its residual Inception-like structure result into very good accuracy and robustness results.

The Xception network splits full convolutional operators into depthwise and pointwise convolutions. The depthwise separable convolutions reduce the necessary computational costs almost ten times with only slightly reducing the accuracy compared to standard convolutions [17]. This led to a considerable drop in depth to 81. The number of parameters was, however, reduced just by 1 million (to 22.9 million). Still, despite of a reduced number of layers and parameters, Xception usually performs slightly better than InceptionV3.

To support feature reuse, the DenseNet model embraces an architecture connecting each convolutional layer to all its successors [18]. DenseNet121 belongs to rather smaller models. It has just 8.1 million parameters and a depth of 242. We can clearly see the effect of the added connections from each layer to all its successors. The number of trainable parameters remains low even though the depth of the model is above average. In addition to reducing the number of network parameters, this approach further improves the efficiency of the network.

The austere model of MobileNetV2 [19] exploits the so-called linear bottleneck layers to capture the function of the entire layer. The model also takes advantage of the so-called inverted residuals. In this case, several bottlenecks follow the input within a residual block and are enhanced by an expansion afterward. Utilizing the much smaller input and output dimensions for the shortcuts improves efficiency of the inverted design considerably. MobileNetV2 has one the smallest depths (105) and also the smallest number of parameters (3.5 million) of all the models in our selection. Considering its small size, MobileNetV2 is able to outperform bigger models in some of the tests.

The NASNet approach automatically searches for the best network architecture considering the data at hand [20]. However, the learned image features can be transferred to other computer vision problems. NASNetMobile and NASNetLarge are two variants of the same model. They share the same structure and differ only in their size. They preceded the EfficientNet model family and also achieve worse results. Due to the depth of 533 and 88.9 million parameters, the NasNetLarge may be too large for the small dataset we have and could easily get overfitted.

The state-of-the-art family of the so-called EfficientNets [21] exploits the NASNet strategy. The baseline model of EfficientNetB0 and its variants B1 to B7 upscaled uniformly in all the network parameters (i.e., width, depth, or resolution) belong to the most accurate and memory-efficient CNNs, now. EfficientNetB0 is bigger than MobileNetV2, but has still a very small size compared to the remaining models. The automated construction of EfficientNetB0 aims at finding the best possible network given the predefined operations.

The bigger size of EfficientNetB5 and B7 leads to improved accuracy and noise robustness. B5 is characterized by the depth of 312 and 30.6 million parameters. B7 is with its depth of 438 and 66.7 million parameters the second largest model in our collection (behind NASNet-Large). EfficientNetB7 also performs the best.

2.2. Android Mobile Devices

Recent mobile devices, e.g., Google Pixel 6 equipped with Google Tensor processor [4], can perform complex computations such as image and video processing, real-time evaluation of CNNs, or other machine learning tasks. Except for the traditional CPU and GPU modules, the Google Tensor processor [4] also contains a TPU module. Below, we will provide a brief overview of mobile devices suitable for CNNs, such as mobile phones, accelerators, and micro-computers. Based on their price, we will consider three categories of Android-run smartphones:

- Mobile phones priced at about 100 EUR, e.g., Xiaomi Redmi 9 A with 2GB RAM and 32GB internal memory. It runs Android 10, has an 8-core MediaTek Helio G25 CPU, and supports AI face-scanning [1]. For the tests, we have created a less powerful virtual mobile phone that could evaluate even the Lite versions of the EfficientNet models.
- Middle-priced smartphones at around 470 EUR, e.g., the Samsung Galaxy phone A52s 5G with 6GB RAM, 128GB internal memory, and an 8-core CPU [2]. It runs Android 11 (can be upgraded to the newest Android 12 OS). Without problems, these phones can operate TensorFlow applications.
- Cutting-edge phones at about 1150 EUR, e.g., the Galaxy S21 Ultra 5G phone with 12GB RAM (can be bought even with 16GB RAM), 256GB internal memory (512GB is possible, too), 8-core CPU, and more [3]. It supports Android 11 and 12. These phones are more powerful than some laptops today, and we might even use them to fine-tune small neural network models in the future.

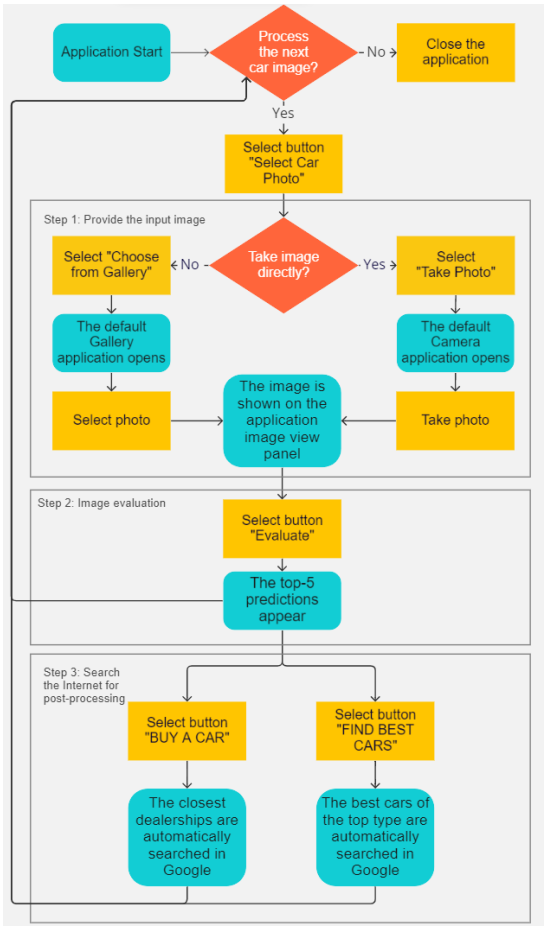

Figure 3: Application flow diagram.

Coral Dev Board is a single-board computer equipped with the Edge TPU coprocessor. Edge TPU is a chip crafted specifically to accelerate machine learning inference (MLI) for mobile CNN models [5]. Another example of a device that uses the Edge TPU coprocessor is Coral USB Accelerator. Its purpose is to enable or accelerate MLI on other external devices. Both Coral Dev Board and Coral USB Accelerator support Tensor-Flow Lite. Further, the accelerator can cooperate with devices that run Debian Linux, macOS, or Windows 10, even with another single-board computer such as Raspberry Pi. Unfortunately, most reviewed mobile devices are not powerful enough to train deep neural networks.

3. Application Design

This section highlights the main design principles for the planned Android classifier of cars' body styles. A viable implementation option would be to gather the input images and send them to a distant server via the Internet. Then, the server shall evaluate the acquired images with a CNN and return the classification results to the Android application afterward to present them to the user. This approach requires a stable Internet connection; without it, the application is out of order.

To overcome this limit, we decided to classify the car images directly within the Android application by a built-in TensorFlow Lite CNN model. A working Internet connection is thus needed only to search for the best scoring cars of the resulting type or the closest car dealerships offering these cars. On the other hand, the chosen neural network model has to be small enough to fit into the Android application. At the same time, the selected CNN must be as accurate and robust as possible (EfficientNetB5, in our case). Figure 3 outlines the application flow diagram.

3.1. The Form of the Employed Data

We used a variant of the Stanford Cars dataset [6] to train and test the respective CNN models. The modified dataset consists of 2560 images of the size 224x224 that belong to 8 different classes (Fig. 2) - BigCoupeOrSedan, Hatchback, MuscleCar, PickUp, Van, SportsCar, SUV, and Unknown. 'Unknown' contains images with no identifiable cars. Table 2 summarizes the class distribution for the involved class labels. For training, we split the dataset into batches of size 64 (i.e., 40 batches in total). 80% of the images form the training set, 20% make up the test set.

For most of the classes, their pattern distributions are comparable. The only exception is the class 'Unknown'. The results we have obtained for the performed tests, however, do not indicate significant overtraining of the CNN models. A reason for this effect could represent extensive data augmentation applied during training. In addition, we can consider the so-called stratified sampling when generating training, testing, or validation datasets in the future.

A critical issue in machine learning consists in adequate data preprocessing. Preliminary experiments indicate, for example, that even the color of the vehicles might strongly affect the classification result. If the data contains specific cars only in one color, the trained model can pick that color as the distinguishing feature. Sometimes, this choice might correspond to particular brand colors, e.g., a red Ferrari or a blue Subaru.

Other factors can also significantly affect the classification results, e.g., the car's angle in the photo. To limit the considerable probability of misclassification in such cases, we decided to augment the training data with car images enhanced by various transformations (e.g., corrupted by noise or taken from different perspectives).

Table 2
Class distributions for the original and validation datasets

Class	Original	Validation
BigCoupeOrSedan	406	42
Hatchback	362	44
MuscleCar	280	35
PickUp	353	38
SportsCar	356	42
SUV	366	42
Unknown	80	9
Van	357	35
Total	2560	287

20% of the original dataset patterns were randomly chosen for testing during the 5-fold CV, the rest was used for training.

During training, CNNs extract features characteristic for the given class and then attempt to detect these features in the images provided for recall. Poorly trained networks can, however, fail to identify representative features from the data. Manufacturers often use, e.g., appealing body parts like headlights or the grille's shape for different types of vehicles they produce. Misguided networks sometimes prefer to choose familiar design elements as vital for classification. We shall thus prepare the training data carefully to encourage an improved classifier performance. In the forthcoming section on supporting experiments, we will describe the employed data set in more detail.

4. Supporting Experiments

We will use the above-specified dataset to test the performance of the considered CNN models: MobileNetV2, EfficientNetB0, EfficientNetB5, EfficientNetB7, NASNetMobile, NASNetLarge, InceptionV3, Xception, InceptionResNetV2 and DenseNet121. While we used Python to write the project for evaluating the experiments, we have implemented the example Android application in Java using Android API and Android Studio version 4.1.3.

To train and test the models, we resorted to the libraries TensorFlow 2.5.0-rc1 [9], TensorFlow Lite [10] and Keras 2.5.0 [11]. Keras can work directly with the ImageNet [8] checkpoints of the selected models. Further, we applied NumPy 1.19.5 [12], and Pandas [13, 14] to process the gathered data (count means, standard deviations, confidence intervals, etc.).

4.1. The Accuracy Test

To test the architectures for the achievable top-1 accuracy, we used the 5-fold cross-validation (CV) over the modified Stanford Cars dataset (see Section 3.1). To enhance the recall capabilities of the trained networks, we added an image augmentation layer to the considered models. This layer automatically adds random noise to the images and is active only during training. Further, the considered augmentations comprise horizontal flip, up to 54 degrees rotation, contrast with a factor set to 0.5, zoom with the height factor set to 0.15 (upper and lower zooming limits), and translation (height and width factors set to 0.15).

During training, the image modifications were performed on-place by means of a set of sequential image augmentation layers from the Keras library. Every training image has thus been randomly modified by all augmentation layers. There also exists a very small probability that the image is left without any modification. We shall, however, highlight that augmentation layers can modify the same image differently in different training epochs. This boosts the involved training dataset several times. In fact, each iteration employs the same number of different training patterns (of the same nature).

We used a Gaussian filter implemented within the SciPy library (scipy.ndimage.gaussian_filter) with the standard deviation (sigma factor) set to 1.5 to create blurry images. The 2D Gaussian kernel we used is defined as:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

x and y denote the distance from the origin (at the center $(0, 0)$ of the filter) in the horizontal and vertical axes.

Image rotation was performed with the Keras RandomRotation layer which involves the respective rotation matrices. In order to make the model noise-robust, we considered random rotations of up to 54 degrees. The newly appeared empty regions near the image borders are filled using a reflection (by reflecting the closest image pixel).

To apply horizontal flips, we used the Keras layer called RandomFlip which performs flipping with a 50% chance. Similarly, using the methods from SciPy, we implemented also the remaining layers such as RandomContrast, RandomZoom and RandomTranslation.

We attached the augmentation layers to the beginning of the models, which allows to modify the images using a GPU acceleration. The augmentation layers also become part of the SavedModel during serialization. The augmentation layers thus do not have to be created separately after loading the model [9].

An alternative option would be to use a TensorFlow Image augmentation pipeline. The augmentation methods are, namely, part of the tf.image library and can be applied to the dataset using the tf.data.Dataset.map function. The advantages of this approach are that the data for the following epoch can be prepared by the CPU in advance during the current epoch and the model itself also

Figure 4: Image of a pick-up truck with random pixels set to random colors with probability equal to 0 (upper left), 0.1 (upper right), 0.2 (bottom left), 0.4 (bottom right).

does not have to be further modified. The CNN would be therefore a little bit smaller, but the image augmentation pipeline should be manually constructed every time before the training of the model starts.

For the beginning five epochs, we trained just the last classifier layers of the networks. Afterward, we kept adjusting the top 10% layers of the networks for additional ten epochs (with fixed last classification layer weights). The results are summarized in Table 3 together with the necessary memory requirements. The table shows that EfficientNetB7, EfficientNetB5, and InceptionResNetV2 belong to the most accurate models.

Some models, however, did not achieve accuracy rates measured on the ImageNet dataset due to the limited number of training epochs, e.g., Xception (with the top-1 accuracy of 79.0% reported for ImageNet). In such a case, (re)training of additional top 20% to 30% of network layers with early stopping and patience set to 10 indicated a significant improvement in the final top-1 accuracy (up to 85.7% for the EfficientNetB7 model and roughly 80% for the other networks).

Table 3
Size and accuracy of the considered CNNs

Model	Size (MB)	Top-1 acc. CI	Lite Size (MB)
MobileNetV2	**19.7**	0.7140 ± 0.0088	**8.5**
EfficientNetB0	31.4	0.7441 ± 0.0067	15.3
EfficientNetB5	210.0	0.8051 ± 0.0067	108.0
EfficientNetB7	455.0	**0.8441 ± 0.0090**	243.0
NASNetMobile	42.0	0.6301 ± 0.0130	16.3
NASNetLarge	503.0	0.7192 ± 0.0071	323.0
InceptionV3	123.0	0.6791 ± 0.0089	83.1
Xception	119.0	0.7057 ± 0.0052	79.3
InceptionResNetV2	317.0	0.8177 ± 0.0081	207.0
DenseNet121	43.0	0.7279 ± 0.0093	26.6

[1] The accuracy is averaged over all 5-fold cross validation steps, CI specifies 95% confidence intervals.

4.2. The Robustness Tests

With these tests, we wanted to assess the resilience of the considered networks to noise corruption and various image modifications. We prepared a unique validation set of 287 images not used previously in training for the experiments. The dataset contains images selected both from the Stanford Cars dataset [6], and from the

Figure 5: A positive blurred image test performed with EfficientNetB5. The upper image is the original one, yet misclassified as a sports car. The bottom one was blurred via a Gaussian kernel with a standard deviation of 1.5 and correctly classified as a hatchback.

Figure 6: A positive cropping test done with EfficientNetB5. The upper image is the original misclassified as a muscle car. The bottom one was cropped from all sides with a factor of 12 (cropping amount: width and height divided by 12), and correctly classified as a sports car.

Table 4
95 % confidence intervals for the considered CNN models accuracy (in %) on noisy images

Model	Original	Blurred	Grayscale	Graysc. Blurred	Cropped	RGB[2]p=0.1	RGB p=0.2	RGB p=0.3	RGB p=0.4
MobileNetV2	71.0 ± 2.5	67.9 ± 1.8	67.0 ± 1.7	66.5 ± 3.0	72.5 ± 3.3	53.2 ± 1.2	29.7 ± 2.5	13.3 ± 4.2	6.3 ± 3.1
EfficientNetB0	72.1 ± 0.6	66.3 ± 0.8	64.0 ± 0.3	65.8 ± 0.1	70.5 ± 0.1	63.2 ± 1.6	57.5 ± 0.7	49.5 ± 0.0	39.4 ± 0.0
EfficientNetB5	75.7 ± 1.2	72.5 ± 1.2	70.7 ± 1.4	67.9 ± 1.4	76.5 ± 1.6	70.0 ± 2.1	68.2 ± 3.0	60.4 ± 0.4	52.8 ± 0.3
EfficientNetB7	**78.4 ± 2.7**	**75.2 ± 0.8**	**72.2 ± 2.6**	**72.3 ± 2.3**	**78.6 ± 1.0**	**74.1 ± 1.6**	**69.3 ± 0.7**	**62.4 ± 1.2**	**56.5 ± 0.0**
NASNetMobile	69.2 ± 2.3	68.2 ± 1.3	64.3 ± 2.8	62.2 ± 1.4	68.7 ± 1.6	57.8 ± 2.4	51.1 ± 1.6	43.0 ± 1.2	36.2 ± 1.7
NASNetLarge	72.6 ± 2.3	66.5 ± 2.4	61.2 ± 2.6	60.3 ± 2.7	73.5 ± 1.7	60.2 ± 1.8	56.9 ± 3.5	50.5 ± 1.8	46.1 ± 1.3
InceptionV3	70.0 ± 2.1	68.4 ± 1.9	64.0 ± 1.2	65.3 ± 1.2	70.7 ± 1.5	60.8 ± 2.3	54.4 ± 2.2	51.1 ± 1.8	44.5 ± 1.1
Xception	70.2 ± 3.2	68.6 ± 3.2	59.2 ± 3.1	61.7 ± 3.5	70.7 ± 2.8	58.7 ± 3.2	51.9 ± 0.8	48.6 ± 2.0	39.5 ± 2.9
InceptionResNetV2	75.5 ± 1.6	71.3 ± 2.4	65.0 ± 0.7	63.5 ± 1.3	74.2 ± 1.1	68.0 ± 2.9	57.4 ± 1.6	53.0 ± 1.4	46.9 ± 0.7
DenseNet121	72.0 ± 3.3	68.8 ± 1.7	59.6 ± 2.0	65.1 ± 2.5	73.3 ± 4.2	57.4 ± 0.8	49.0 ± 0.5	40.8 ± 0.5	32.4 ± 1.9

[1] The top-1 accuracy of 5 different checkpoints trained on the original data set with early stopping was considered, its mean was calculated together with the corresponding 95 % confidence intervals (in %).
[2] RGB stands for RGB noise. Random pixels were set to a random color value with probability p (%).

Figure 7: A negative cropping test done with EfficientNetB5. The upper image is the original one correctly classified as a hatchback. The bottom one was cropped from all sides with a factor of 12 (cropping amount: width and height divided by 12), yet misclassified as an SUV.

DVM-CAR dataset, [7] (63 of them to better simulate the reality).

Further, we created multiple variants of the 287-image validation dataset (see Table 2). Each variant contains all of the images from the original dataset modified in a different way: changing random pixels to a random color with a certain probability, blurring the images, cropping them, and making them grayscale and/or blurring them at the same time. This way we were able to test separately the behavior of each model on modified images.

We trained the networks by employing early stopping with patience set to 10 and GPU acceleration. The training ran for all models five times in a row and used always the same validation data. Table 4 shows the results averaged over all five runs.

Regards the robustness to random pixel color changes (see, e.g., Fig. 4), EfficientNetB7 and EfficientNetB5 are the most robust but, at the same time, memory-intensive models. The smallest network, MobileNetV2, demonstrates, on the other hand, the worst results in this test. As an acceptable compromise, we can thus pick the EfficientNetB0 model with just 31.4MB memory requirements and results outperforming many more extensive networks, e.g., Xception, NASNetLarge, and even InceptionV3 (except for highly noised images).

On blurred and grayscale image sets, the models performed better than in the RGB noise test, and the MobileNetV2 model achieved even higher accuracy than EfficientNetB0. The other two EfficientNet models are significantly more accurate than both the MobileNetV2 and EfficientNetB0. Yet as the top-1 accuracy fell significantly for grayscale images compared to the original validation dataset, color proves to play an essential role in the classification process. Also, classification is more accurate for blurred images than for grayscale images.

Although blurring does not improve the overall accuracy of the networks, it can sometimes emphasize significant image characteristics and improve the classification. For example, let us consider the case illustrated in Fig. 5. The EfficientNetB5 model misclassified the shown vehicle as a sports car but correctly classified the blurred one as a hatchback. Blurring emphasized the edge separating the back of the vehicle from the background for the model, thus better indicating a hatchback.

For many CNN architectures, cropping of images results in a higher top-1 accuracy compared to the original validation set. During training, the augmentation layer prepares the network for this test scenario, and cropping removes the image's noisy edges, thus focusing better on the main object, see, e.g., Fig. 6. On the other hand, cropping can also impact unwanted results, see, e.g., Fig. 7 of a hatchback misclassified as an SUV after image cropping caused the car to fill the whole image and appear to be more spacious.

4.3. CNN Models And Mobile Android Devices

From the robustness tests, we see that the most noise-robust models are EfficientNetB7 and EfficientNetB5 followed by InceptionResNetV2. These models may be, however, too big for mobile devices. MobileNetV2 and EfficientNetB0 have the best accuracy-to-size ratios, so we consider them suitable for CNNs to be implemented in mobile devices. If we look for a higher accuracy, then EfficientNetB0 is a better choice. But what if we seek a maximum accuracy?

Further, we will focus on EfficientNetB5, EfficientNetB7, and InceptionResNetV2, which are the most accurate and memory-demanding CNN models. A bigger size means slower evaluation, more challenging learning, and more energy consumption. To see whether the usage of these three models slows us down, we have run a speed test in which we did 100 evaluations of the 287 validation images. For every model, just one checkpoint from the robustness test discussed above was considered. During the assessment, the images were loaded as a dataset object to TensorFlow and split into batches of size 64.

As we can see from Table 5, EfficientNetB5, EfficientNetB7 and InceptionResNetV2 are among the slowest models. Yet, we want to use these models to evaluate a single image at a time (as we are not working with videos). So even the biggest models are sufficiently fast. As an example, EfficientNetB0 is just two times faster than EfficientNetB5. At the same time, EfficientNetB0 evaluates one image in less than approximately 0.005 *s* when considering our current test settings and the specifications of our computer listed below. So both this computation and the computation of the two times slower EfficientNetB5 are indistinguishable from this point of view.

This test ran on the DellG5-15 laptop, which has NVIDIA GeForce RTX 2070 Max-Q Design GPU, Intel Core i7-9750H 2.60GHz CPU, and 16GB RAM. The mobile devices planned for the considered models have much less computational power. Therefore, we decided to focus on the most accurate and robust models for an actual Android application to be created via Android Studio 4.1.3.

An Android Studio project can contain only files smaller than 200MB. So it was impossible to upload directly (without any size optimization) EfficientNetB7 and InceptionResNetV2 to the application as the sizes of their Lite versions are 243MB and 207MB. The most accurate and robust model smaller than 200MB is EfficientNetB5. Therefore, we decided to upload this model into the developed Android application as part of the experiment.

Then, we compiled the application on a virtual mobile phone. Intentionally, we made this device computationally as slow as possible. The aim was to replicate the characteristics of the cheapest mobile devices one can find on the market. Our virtual machine had 1536MB of RAM, and its CPU was set to Google Play Intel Atom (x86), which has only four cores. Further, the device had the Android 7.0 operating system, which is the minimum system our application supports (Android 7.0 was released in 2016, so meanwhile it is considered to be an old system).

Table 5
Mean evaluation time over 100 classifications performed in a row over 287 images (the images were split into batches of size 64) and a summary of the results

Model	Eval. time (s)	Fits in mobile	Accurate enough	Fast classification
MobileNetV2	**1.081**	**YES**	NO	**YES**
EfficientNetB0	1.203	**YES**	NO	**YES**
EfficientNetB5	2.236	YES	YES	YES
EfficientNetB7	3.346	NO	**YES**	NO
NASNetMobile	1.335	**YES**	**NO**	**YES**
NASNetLarge	3.618	**NO**	NO	**NO**
InceptionV3	1.307	YES	**NO**	**YES**
Xception	1.709	YES	NO	YES
InceptionResNetV2	2.121	NO	YES	YES
DenseNet121	1.516	**YES**	NO	YES

YES / **NO** denotes very good / very bad. YES / NO denotes good / bad.

As discussed above, even the cheapest mobile phones are more powerful today than the simulated mobile phone. It is also important to note that when writing this article, the newest Android operating system was 12. Despite of that, it was possible to evaluate EfficentNetB5 successfully on our virtual device. So EfficientNetB5 can be used in combination with the most affordable smartphones.

By converting EfficientNetB5 to TensorFlow Lite, we did not lose its accuracy as TensorFlow Lite ensures a stable conversion without modifying the structure of the model. Only the network format is changed to become more compact and easier to access and evaluate [10]. Table 3 shows that the resulting model size can fall up to two times without losing accuracy. Also, it is already possible to apply on-device training to improve the Lite models. The only limitation is that not all neural network operators are available in Lite, so theoretically, not all models are convertible to Lite. Anyway, we succeeded in converting all CNN models we have considered to Lite.

5. Conclusions And Further Research

Contemporary Android devices are powerful enough for real-time image processing based on neural networks. In this paper, we studied the accuracy, evaluation speed, robustness, and size of 10 considered CNN models and selected the best-performing ones to upload to the developed Android smartphone application.

Table 5 summarizes the results obtained for the car dataset. According to top-1 accuracy, the most accurate models are EfficientNetB7 (84.4%), InceptionResNetV2 (81.8%) and EfficientNetB5 (80.5%). These models are the biggest ones for their TensorFlow Lite size (243, 207, and 108MB, resp.) yet remain easy to train.

After only 15 training epochs, the networks achieved adequate accuracy in the 5-fold CV test. The aforementioned networks are robust against random RGB noise and various image distortions. All of them can be converted to the Tensor-Flow Lite format, although EfficientNetB7 and InceptionResNetV2 do not fit into an Android application. Due to its size, the most accurate and noise-robust model suitable for an Android Studio application seems to be EfficientNetB5.

Should the model be as small and as fast as possible, EfficientNetB0 might pose a better choice. It achieves satisfiable accuracy and robustness results and it is the second smallest model among the considered ones. Further, it can achieve better results than many bigger models like Xception, NASNetLarge, and even InceptionV3. The main contribution of this study thus consists in:

- the development of a mobile Android application that facilitates the classification of car images according to the car's body style.
- the choice of the EfficientNetB5 model for the developed smartphone application. Extensive testing of the CNN models in question justifies this decision that constitutes an acceptable compromise for all the criteria, particularly concerning the model's accuracy, robustness, and the required time and memory costs.
  Only for EfficientNetB5, we obtained good results (although not the very good ones) conforming to all three considered criteria. None of the other models meets all of them. The other candidate models, EfficientNetB7 and InnceptionResNetV2, achieving acceptable accuracy results, do not fit into the mobile application.

While working with the standard TensorFlow library, we did not encounter any significant problems. But to assess the viability of the networks for future on-device fine-tuning, we also measured the memory requirements of EfficientNetB5, EfficientNetB7, and InceptionResNetV2 during training (see Table 6; we averaged the obtained results over five training sessions). EfficientNetB5 consumed 5.3GB of GPU memory and 2.5GB of RAM during each training session.

Our computer needed 251.5MB of GPU memory to store EfficientNetB5 and its training metadata. The other two models were more demanding. Yet, even if we focused only on the EfficientNetB5, we would need a cutting-edge category smartphone like the Galaxy S21 Ultra 5G phone equipped with 12GB of RAM to launch on-device training. The conversion to Lite reduces the models' size up to two times without reducing their accuracy.

Table 6
Memory usage during training

Model	GPU memory used (GB)	Max. RAM[1] used (GB)	Model size[2] (MB)
EfficientNetB5	5.3	**2.5**	**251.5**
EfficientNetB7	5.4	2.7	556.7
InceptionResNetV2	**3.0**	2.6	425.3

[1] A possible bias can be caused by programs running on the background.
[2] The size of the model located on the GPU during training (including the training metadata).

The TensorFlow Lite library was, on the other hand, built to operate on portable devices with low computational power. Originally, the Lite library did not allow on-device training of Lite models. Meanwhile, this limitation has been removed and on-device training is already supported. Despite of a well-written TensorFlow documentation, training of Lite models still remains quite cumbersome, at least from the programming point of view.

Another limitation for our research comes from the Android Studio that we have used to implement the trained CNNs in mobile applications. It has an inbuilt size limit of 200MB for external files to be uploaded to a project. As a result of this restriction, we were not able to upload Lite models bigger than 200MB to mobile applications..

The last limitation is that Android Studio does not officially support uploading of TensorFlow models saved in formats different from TensorFlow Lite. On the other hand, TensorFlow Lite supports also other operating systems such as iOS, so the developers are not limited to writing their applications just for Android.

Further research could enhance the developed application both with on-device training and with federated learning. Federated learning enables robust training across several decentralized edge devices or servers holding local data samples without sharing them. This way, the inbuilt CNN classifier could be easier retrained on new data to keep the implementation up-to-date. Other intriguing options for future research comprise the area of architecture optimization for the trained networks and the involvement of nature-inspired heuristics in the process of CNN design.

Acknowledgments

This research was supported by SVV project No. 260 575.

References

[1] Xiaomi Czech, "Xiaomi Redmi 9A", link https://www.xiaomi.cz/xiaomi-redmi-9a-2gb-32gb-sky-blue/, Accessed: 27 Jan. 2022

[2] Alza.cz, "Samsung Galaxy A52s 5G", link: https://www.alza.cz/samsungu-galaxy-a52s-5g?dq=6667487, Accessed: 27 Jan. 2022

[3] Samsung, "Samsung Galaxy S21 Ultra 5G 256GB", link: https://www.samsung.com/cz/smartphones/galaxy-s21-5g/buy/, Accessed: 27 Jan. 2022

[4] Monika Gupta, "Google Tensor is a milestone for machine learning," 19 Oct. 2021, Accessed: 14 Nov. 2021, link: https://blog.google/products/pixel/introducing-google-tensor/

[5] Google LLC, "Edge TPU performance benchmarks", 2020, Accessed: 16 Nov. 2021, link: https://coral.ai/docs/edgetpu/benchmarks/

[6] J. Krause, M. Stark, J. Deng, and L. Fei-Fei, "3D object representations for fine-grained categorization", 4th IEEE Workshop on 3D Represent. and Recogn., ICCV 2013.

[7] J. Huang, B. Chen, L. Luo, S. Yue and I. Ounis, "DVM-CAR: A large-scale automotive dataset for visual marketing research and applications", 2021, arXiv: 2109.00881

[8] J. Deng et al., "ImageNet: A large-scale hierarchical image database", CVPR, 2009, pp. 248–255.

[9] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous systems, 2015", Software available from tensorflow.org.

[10] TensorFlow, "Deploy machine learning models on mobile and IoT devices", Accessed: 6 Feb. 2022, link: https://www.tensorflow.org/lite

[11] F. Chollet et al., "Keras.", 2015, link: https://keras.io

[12] C.R. Harris et al., "Array programming with NumPy." Nature 585, 2020, pp. 357–362.

[13] The pandas development team, "pandas-dev/pandas: Pandas 1.3.4", 2021, doi: 10.5281/zenodo.5574486

[14] W. McKinney et al., "Data structures for statistical computing in python", Proc. of the 9th Python in Science Conference Vol. 445, 2021, pp. 51–56.

[15] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens and Z. Wojna, "Rethinking the Inception architecture for computer vision", CVPR, 2016, pp. 2818–2826.

[16] C. Szegedy, S. Ioffe, V. Vanhoucke, and A.A. Alemi, "Inception-v4, Inception-ResNet and the impact of residual connections on learning", AAAI, 2017, pp. 4278–4284.

[17] F. Chollet, "Xception: Deep learning with depthwise separable convolutions", CVPR, 2017, pp. 1800-1807.

[18] G. Huang, Z. Liu, L. van der Maaten and K. Q. Weinberger, "Densely connected convolutional networks", CVPR, 2017, pp. 2261-2269.

[19] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and L. -C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks", CVPR, 2018, pp. 4510-4520.

[20] B. Zoph, V. Vasudevan, J. Shlens and Q. V. Le, "Learning transferable architectures for scalable image recognition", CVPR, 2018, pp. 8697-8710.

[21] M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks", Proc. of the 36th International Conference on Machine Learning, PMLR 97:6105-6114, 2019.