Real Time Image Recognition with Deep Learning and Android application (December 2019)

A. Fodor, *TU Budapest*

Abstract—Machine learning allows computers to solve more and more complex tasks; object recognition, speech synthesis, or time series prediction are good examples. The advancement of technology allows these models to be available on multiple devices. In parallel, mobile phones have become part of our everyday life. With the help of their sensors, data can be collected about us and our world. The combination of Machine Learning and mobile devices can contribute to discover latent connections yet unexplored. It can open up new opportunities that help us in our daily lives.

The aim of this paper was to have a look at this process through a Machine Learning development life cycle. The steps necessary for this, such as the preparation of the data, the construction of the model, and the completion of teaching, are described in detail. In addition, I explain the process of connecting the Neural Network and Android module to give the reader a complete picture of what is needed to bring such a system to life.

The outcome is an application that can identify cat and dog breeds in real-time with the device's camera. Its functions help to identify breeds unknown to the user, to determine the breed of mixed animals and to select the pet that suits the lifestyle of the keeper.

***Index Terms*—Machine learning, Neural networks, Image recognition, Real-time systems**

I. Introduction

Machine learning enables us to solve tasks that are becoming increasingly complex; recognition of objects and emotions, speech synthesis, time series prediction, even automated planning. Technological advances allow the achievements of this discipline to be displayed on more and more devices and to solve complex tasks that would be very time-consuming to manually program. Machine learning is currently the most efficient in the field of image recognition. Although ML is popular in other fields of applications as well, image recognition was the first where it could produce practical benefits[1].

By combining mobile devices with the potential of machine learning, we can discover relationships and solve everyday tasks that are difficult to achieve with traditional methods.

The purpose of my task is to create an image recognition model and use it with a mobile platform that can identify different dog and cat breeds with the camera of the device in real time.

I chose this topic because there is a constant need for information on pets. Many people always plan to keep a dog or cat, but you can't expect to know the characteristics of hundreds of breeds. Therefore, it would be useful to have an application that provides real-time information to recognized animals and makes it easier to select a suitable cat or cat for a person's lifestyle. Because keepers often have emotional attachments to their pets, this information may be important to them. The application could also be used to determine parent breeds of mixed dogs and cats.

A. Benchmarking

The currently available applications do not fully meet the previously aimed needs. The Apple App Store has only one popular (2,100 rated) dog recognition application that cannot run in real time. The most popular animal recognition app on the Google Play Store (12,000 ratings) cannot run real-time and needs Internet connection to evaluate. This generates high data traffic and makes the application slow. Based on the interest of these applications, the user demand is real, but current solutions do not take advantage of the platforms.

So, I decided to create a Convolutional Neural Network that can identify dog and cat breeds and compose an Android application that can use the model in real time. I present the construction process in the following sections.

II. Data operations

The following section describes the process of collecting, cleaning, and building up the data needed to build the system.

A. Data collection

An important consideration in obtaining the data was to cover as many dog and cat breeds as possible. I expected that it would not be enough to have only one dataset, it would require creating my own database. The data was collected by Kaggle and Google Dataset Search. I used four public datasets to construct the final one.

The first is the Stanford Dogs dataset[2], which contains 20,580 images for 120 different breeds. It is a good starting point, but it lacks basic breeds (such as akita and dalmatian) too.

The second was the Cats and Dogs Breeds Classification

Oxford dataset[3]. It only has 37 different classes and 7,400 pictures but covers well the common cat breeds.

The third was a dataset called Cropped dog breeds[4]. While there is a lot of overlap with the first set of data, it does complement the nearly 200 different breeds of dogs it contains. However, there are significantly fewer images for each class, moreover, in an unequal distribution. So, I only added to my database the missing varieties that had enough pictures.

Finally, I used a celebrity dataset[5] to make the model recognize people. It contains celebrity portraits of which I put a total of 200 images in my database into a general person category. Because this dataset well covered genders (equalized male to female) and possible races, I created the class using the same number of images from each celebrity. My goal was to eliminate the possibility of underrepresented subclasses in the general human class. This way, people of different genders and skin colors are equally likely to be recognized.

B. Data cleansing

The merging of the databases revealed that different numbers of images are available from different types. This is not good because if there are 400 images of Beagle and just 100 of Labrador, the system may tend to predict Beagle more because of the first few samples.

An extreme example: if only two classes need to be distinguished and there are 19,000 images in the database and only 1,000 images in the database, a model that always predicts a can achieve 95% accuracy on a randomly constructed evaluation set from the same source. And building a system that always says a does not require machine learning. Using another scoring metric, looking at the hit rates for each class separately, it will be 100% for a and 0% for b; and the two are equally weighted with accuracy of 50%, which better represents the true knowledge of the model.

The example above illustrates the importance of applying appropriate metrics and balancing the classes in the database. I decided that the ratio of classes with the most and least elements should be a maximum of 2: 1. First, varieties that had extremely high images than average had to be thinned out. The average number of pictures in the classes was 172, so I decided to reduce classes of 250 or more to 200. And I leave out those with a cardinality of 125 or below. It may happen that a class of 125 and a class of 249 will remain, but that is already within the prescribed limits.

C. Generate teaching, evaluation, and test sets

After collecting and cleaning the classes, I divided the database into three subsections.

The teaching set is the part with which the model is taught. Of the three, this unit contains the most images.

The validation set is used to periodically check how well the model is performing. This should be done on images that are separate from the training set, because our goal is not to memorize images in the model, but rather to generalize what they represent to completely unknown images. Therefore, the evaluation must be done with data unknown to him.

The third is the test set. Essentially, it is used to validate a model that performs well during the evaluation with newer images. If a system is often evaluated with the same validation set, it may eventually become implicit to fit its features[6]. This is because during teaching, we change it so that the measured result is as good as possible, which we evaluate on the validation set. The test set can be used to test well-performing models, that is, whether they are good or against the phenomenon described above.

Also, in order to avoid the phenomenon outlined in the previous paragraph, it is important that our database is balanced and, if it is met, that all three sets contain equally general images. Therefore, the generation was done by applying the 70-20-10 rule (70% teacher, 20% validation, 10% test image) for each breed, and the order of the order was random for each class image. Applied separately to each breed, 70-20-10 ensures that the class ratios of the entire database are maintained in the subsets, with the randomness that diverse images are placed everywhere.

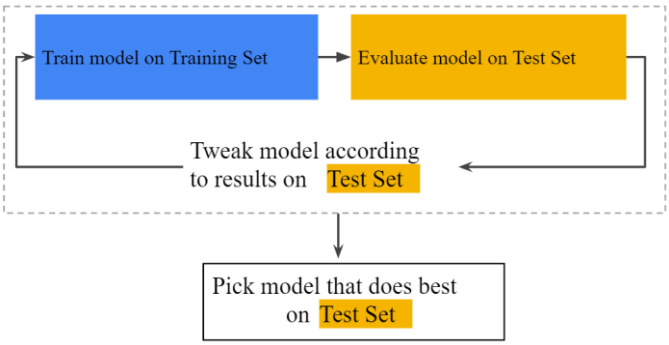


Figure 1: workflow with teaching, validation and test sets[6]

D. The database

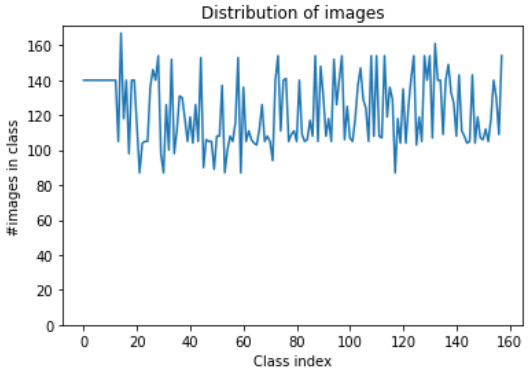
The database was generated using a Python script made by myself. The completed dataset is 1.53 GB and contains 27,699 images (19,290 teachers, 5587 validations, 2822 tests). There are 158 different classes including 1 human, 12 cat and 145 dog breeds (this inequality is not a problem because we treat each breed completely separately). An eight-digit unique identifier was generated for each group to resolve name conflicts (because dogs and cats of the same name may occur).

```
Class Dachshund (generated name: n03000053-Dachshund) - training: 103, validation: 30, test: 15 image(s)
Class Dalmatian (generated name: n03000054-Dalmatian) - training: 112, validation: 32, test: 16 image(s)
Class Dandie_Dinmont (generated name: n03000055-Dandie_Dinmont) - training: 126, validation: 36, test: 18
Class Dhole (generated name: n03000056-Dhole) - training: 105, validation: 30, test: 15 image(s)
Class Dingo (generated name: n03000057-Dingo) - training: 108, validation: 32, test: 16 image(s)
Class Doberman (generated name: n03000058-Doberman) - training: 105, validation: 30, test: 15 image(s)
```

Figure 2: Excerpt from the output of the database generator script with information of six classes

From the point of view of the directory structure, the root contains three subfolders (training, validation, test), each with 158 additional breed folders, each containing images of the given set. The root also contains a file called labels.txt, which contains the ID and name bindings.

TRAINING DATASET



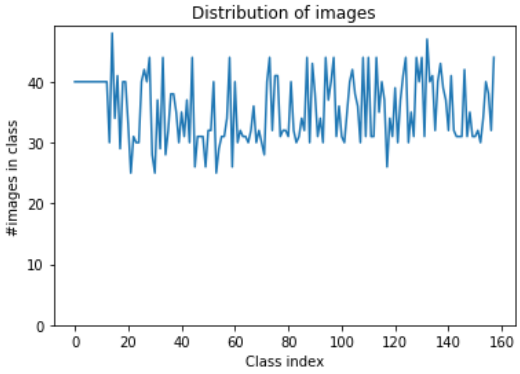

```
---
# images: 19290
# classes: 158

Class with most images: ('Afghan_hound', 167)
Class with least images: ('Anatolian_shepherd_dog', 87)

Avg. per class: 122
Mean: 122
```

Figure 3: Data distribution of the teacher set

VALIDATION DATASET




```
---
# images: 5587
# classes: 158

Class with most images: ('Afghan_hound', 48)
Class with least images: ('Anatolian_shepherd_dog', 25)

Avg. per class: 35
Mean: 35
```

Figure 4: Data distribution of the validation set

III. Preparing for Modeling

Before implementing the system, my goal was to create a reusable frame in which different networks can be quickly built and taught. This includes developing a data pipeline (input preprocessing), a specific model building method, a form of teaching, and evaluation components.

A. Input Preprocessing

Knowing the database available, I will first introduce the preprocessing steps of the images.

1) Generating the expected format

Convolutional networks basically work with fixed-size inputs. In order to support more than just one, it is advisable to resize the images to the same size and format. I did this with Keras ImageDataGenerator class when reading from the database. Scanning was done in RGB format and normalized between 0...255 on each channel to 0...1.

```
# Normalize input
imgDataGenerator = ImageDataGenerator(rescale=1./255)
# Flow images in batches of 32 using generator
imgGenerator  = imgDataGenerator.flow_from_directory(
      # This is the source directory of images
      sourceDir,
      # All images will be resized to dimensions[0] x
      dimensions[1]
      target_size=(dimensions[0], dimensions[1]),
      batch_size=32,
      color_mode='rgb',
      shuffle='True',
      class_mode='categorical',
      data_format='channels_last',
      dtype='float32')
```

Teaching, evaluation, and test sets are all scanned using ImageDataGenerators in random order. However, the training set generator also performs data enrichment operations.

2) Data augmentation

The database contains nearly 28,000 images. Although this value does not seem small, it is in fact because so many samples are distributed over 158 classes. So, on average, 177 images can be farmed per breed. Since 20% of the images in each class are used for validation and 10% for test purposes, 124 items remain for teaching purposes. This is very little and carries the risk of over-learning the model.

Over-learning is when a system performs better on test data than evaluators. By this time, the generalization ability is already deteriorating as the network begins to memorize the teaching data. Because the goal is to successfully identify unknown images, this phenomenon should be avoided.

If we teach with too little data, there is a great risk of over-learning because the characteristics of a given class may appear differently in the teaching data than in reality. This could mislead the model. For example, if we teach a car class with only 4 pictures, of which 3 are blue cars, the system may reasonably believe that there is a relationship between the blue color and the car's existence in the picture. This is a false assumption and the model will probably not recognize a red car anymore.

To prevent over-learning, I applied data enrichment to the teaching samples. This means random transformations (rotation, vertical reflection, stretching, darkening) that slightly change the original image. This way, reading the same image never produces the same result.

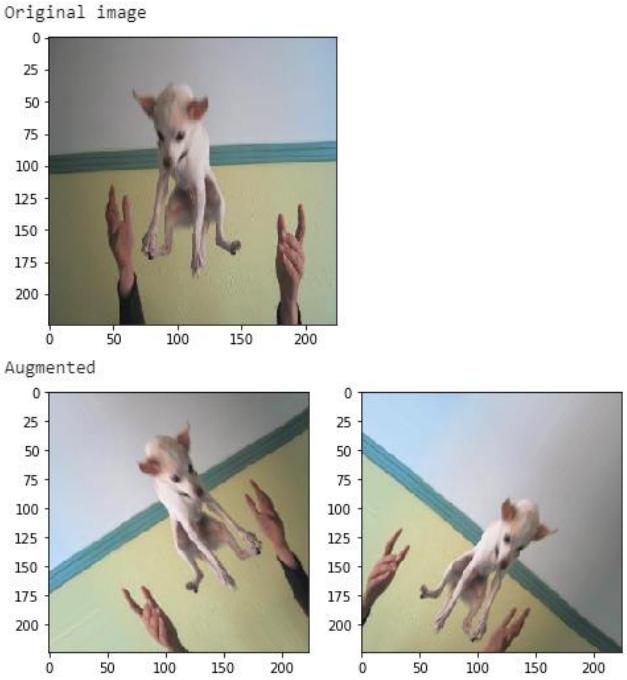


Figure 5: Original image (above) and augmented samples (below)

B. The Model Building Process

The different image recognition networks are constructed in different methods, which have the same signature when viewed from the outside. Returns a Model object based on the function inputs. The following code snippet illustrates the implementation of a simple convolutional network.

```
def buildModel(dimensions, classes):
  """Builds up a convolutional neural network from scr
atch

  dimensions -
- Dimensions of the input 2D image - [0] and [1] are u
sed
  classes -
- Number of classes to choose from - output size of th
e network

  Returns the compiled Model
  """

  # The input feature map is dimensions[0]           x
dimensions[1] x 3:
  # dimensions[0] x dimensions[1] for the image pixels
, and
  # 3 for the RGB color channels
  inputImage = layers.Input(shape=(dimensions[0], dime
nsions[1], 3))

  # First convolution extracts 16 filters that are 3x3
  x = layers.Conv2D(16, (3, 3), activation='relu')(inp
utImage)
  x = layers.MaxPooling2D(2, 2)(x)
  # Second convolution extracts 32 filters that are 3x
3
  x = layers.Conv2D(32, (3, 3), activation='relu')(x)
  x = layers.MaxPooling2D(2, 2)(x)
  # Third convolution extracts 64 filters that are 3x3
  x = layers.Conv2D(64, (3, 3), activation='relu')(x)
  x = layers.MaxPooling2D(2, 2)(x)
  # Use Global Average Pooling 2D which flattens the o
utput to 1 dimension
  x = layers.GlobalAveragePooling2D()(x)

  # Add a fully connected layer with 2048 hidden units
 and ReLU activation
  x = layers.Dense(2048,activation='relu')(x)
  # Add a dropout rate of 0.2
  x = layers.Dropout(0.2)(x)
  # Add a final softmax layer for classification
  prediction = layers.Dense(classes, activation='softm
ax')(x)

  # Configure and compile the model
  model = Model(inputImage, outputs=prediction)

  optimizer = SGD()
  model.compile(loss='categorical_crossentropy', optim
izer=optimizer,
  metrics=[metrics.mean_absolute_error, 'acc',
  metrics.categorical_accuracy])

  model.summary()

  return model
```

C. The teaching and assessment process

Once the network has been set up, it will be necessary to determine how it will be taught and how it can be evaluated.

1) Teaching

The model is taught using the fit_generator() method, which uses the training data generator as a data source. This generator delivers images in batches of 32 elements. To potentially cover the entire data set during a training iteration step (epoch), I selected the steps per epoch so that the point is the number of elements in the training set divided by 32.

```
history = model.fit_generator(trainingGenerator,
      steps_per_epoch=trainingSize/32,
      epochs=1,
      validation_data=validationGenerator,
      validation_steps=validationSize/32,
      verbose=1,
      callbacks=[tensorboardCallback])
```

At the end of each epoch, the validation generator can be used to monitor the current prediction capability of the model.

2) Evaluation

Since it is of little importance to the identification of animal species, which type of error is more frequent (first or second order) for a class, and the number of classes of the database used is approximately correct, I used the accuracy metric. For the final evaluation of the model, I also used a confusion matrix to filter out the very weak / overly dominant categories.

A thorough representation of the results is provided using TensorBoard. It is a visualization tool where, in addition to basic metrics, the weights of each layer can be monitored during teaching. In addition, for the evaluation, I made three different methods that were applicable to all models.

The first option is to run prediction on specific images.

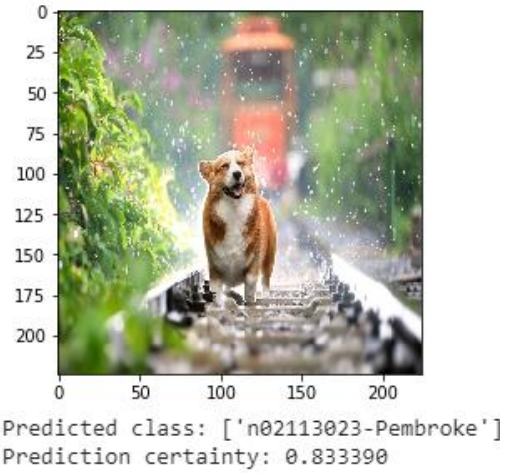


Figure 6: Prediction on a specific image

The second method is to draw graphs from the history file generated during model teaching. During the development I observed the development of accuracy and loss on the training and evaluation data.

The third option is the drawing of the confusion matrix, which is an illustrative representation. Assigning the values of the matrix elements to a color scale also shows some tendencies based on color (for example, where the model tends to go wrong). However, it has the disadvantage that it is difficult to see in 158 different classes.

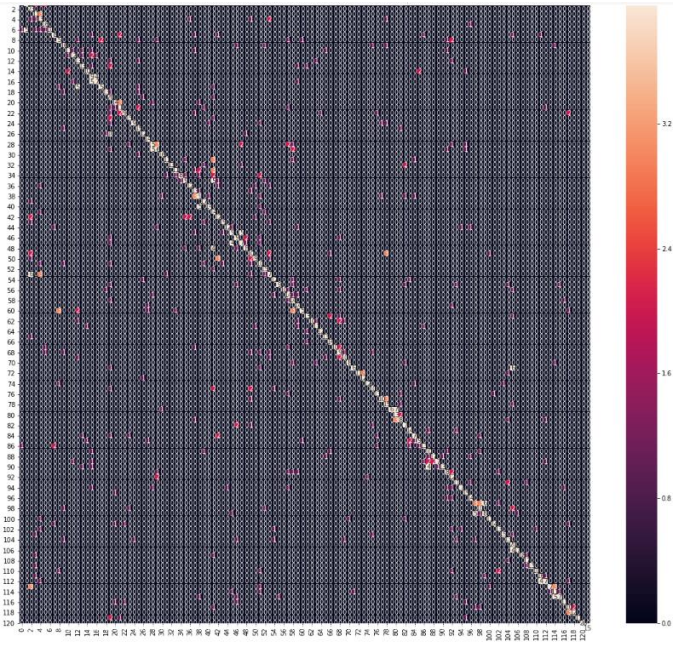


Figure 7: Confusion matrix of a network - the clearer a cell, the higher its value. The main diagonal indicates correct results.

IV. Selecting the Initial Network

After developing the frame for the development process, the model was developed. I decided to try three options, and whichever works best, I will continue to work on it.
To make the comparison correct, these networks differed only in their convolutional part, with dense layers at the end (2048 perceptron layer with ReLU and then an output layer with 158 elements with SoftMax activations). I used SGD as the optimization algorithm, with the same learning rate (0.01). I chose categorical cross entropy as a cost function. In each case, the teaching lasted for 10 iteration steps, with 2000 steps per epoch.

A. *Custom convolutional model*

First, I created a simple network whose convolutional structure was inspired by a Keras pattern[7]. This model was able to distinguish dogs and cats with 80% accuracy after 50 training cycles, teaching at 2,000 images. My own task is much more complicated because there are 158 classes instead of 2. My basic idea, however, was that once the image recognition part can distinguish features that can be used to separate dogs / cats, it can do a little more with more features. Therefore, instead of the original 3 convolutional layers, I used 4 and then placed the dense grading layers.

The network had a total of 720,000 parameters and the training took 24 minutes. The accuracy for the 10 epochs for both the teaching and assessment samples is around 1% (a completely random result would be 1/158 = 0.63%).

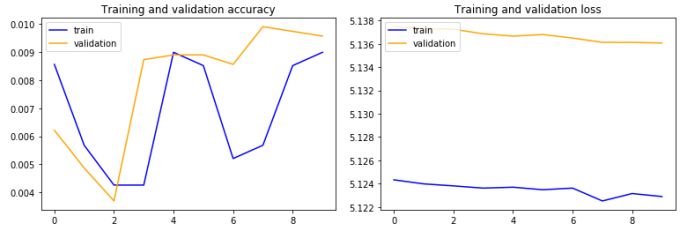


Figure 8: Results of teaching the custom convolutional network

B. *Inception V3*

Next, I tried a model that incorporates convolutional layers of the Google-developed Inception V3[8] network, pretrained with the ImageNet dataset[9]. Because convolutional layers provide features, a model taught in generic images can be reusable. If we cut off the classifying part from the end, we get a network that, at its output, clearly separates the features shown in the images. If we put our own classifiers on it and teach only these, we will save the teaching of the convolutional part. This method is called feature extraction.

This model is already larger, with almost 11 million parameters. Of this, 1.9 million should be taught (layers at the end of the network). This is almost three times the total weight of the previous network. However, teaching is expected to be faster: slow learning conv layers do not need to be changed.

Teaching was completed in 11 minutes, significantly faster than the previous case. Accuracy is about 1% for the training set and 1.2% for the validation samples.

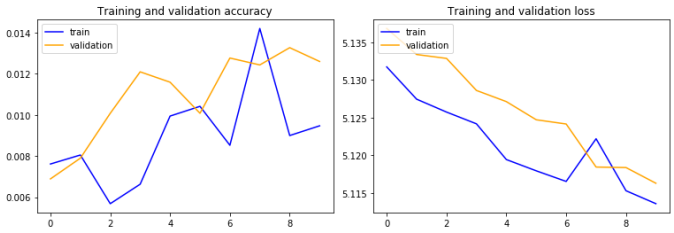


Figure 9: Results of teaching Inception V3

C. *MobileNet V2*

Finally, I tried the MobileNet V2[8] pre-taught network. As its name suggests, this model is optimized for running on mobile devices: its size is much smaller than Inception V3 because it is made up of less complex components. There is a total of 5,225,704 parameters, of which we will teach 2.9 million (grading layers at the end of the network). There is so much of this because the last convolutional layer of MobileNet is nearly twice the depth of Inception, on which we build the dense layers.

The duration of this teaching was 10 minutes. Already at the end of 10 epochs, the model achieved 35% accuracy on the validation set and 20% on the teacher. This is an order of magnitude greater than the previous two cases. The training samples are likely to perform much worse than validation due to data enrichment. If this trend persists, there is no need to fear over-fitting.

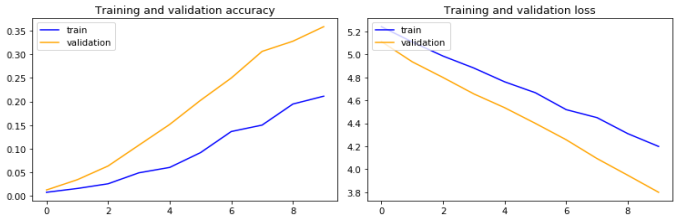


Figure 10: Teaching results of MobileNet V2

D. *Choice*

Of the three models, the first option was the most difficult to use due to the slowness of teaching, so I will compare the two pre-taught networks below. Teaching them requires similar resources (10 epochs~10 minutes), but MobileNet learned much faster.

The better results of MobileNet may be since my final classification structure used for all models is connected to the end of pre-trained networks with a Global Average Pooling 2D layer. The output of this system has 1280 5x5 output characteristics. Inception has only 768 but 12x12 sizes. The GAP loses the feature matrix size. This is higher for Inception (12x12), resulting in more information loss. That's why we had to train less than Inception, and probably did so much worse. This can be eliminated by using a flatten layer instead of GAP, but in the case of Inception, the 2 million teachable parameters would be in the order of billions. This solution would be very slow to run on a mobile device, so I'm not going to try it.

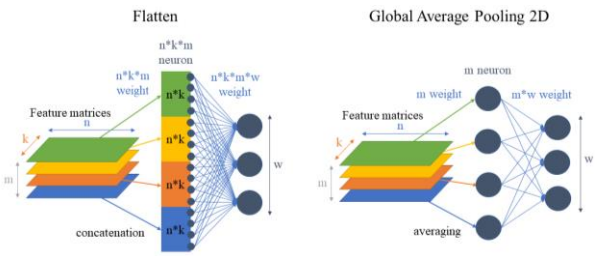


Figure 11: Comparison of Flatten and GAP 2D layers. It is clear how large the difference in the number of weights is already for 2x2 property matrices.

Due to the faster learning and smaller size of the networks examined, I chose MobileNet.

V. Teaching and Fine Tuning

In this section I will describe the optimization and teaching process of the selected network, which resulted in the final model.

A. *Hyperparameter optimization*

By changing hyperparameters (such as learning rate, optimization function, number of layers and neurons), the goal is to increase the learning capabilities of the model. Finding the optimal combination of these is a difficult task. The following is an intuitive introduction to what changes I have tried.

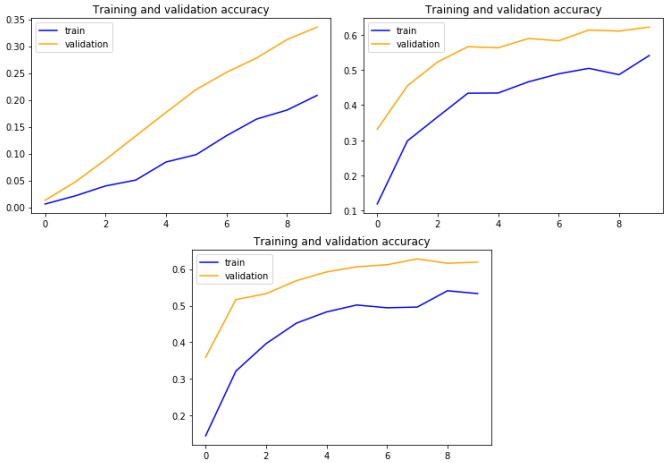


Figure 12: Comparison of the SGD (left), RMSprop (right) and Adam (bottom) algorithms

1) *Choosing an optimization algorithm*

I tried three options for the selected network. I used the default initial value for each algorithm because, for example, the optimal learning rate for each function may vary. In each of the three cases, teaching the models for the same amount of time I examined which model was the most promising.

SGD achieved 20.83% on the teacher and 33.55% on the evaluation samples. This algorithm is usually efficient but slow. On the accuracy curve, after all this teaching, there is no convergence and it could be continued for too long without the risk of over-learning.

After 10 epochs, the RMSprop trained network performed 54% on training samples and 62.15% on validation samples. This is a much faster algorithm towards its optimum, accuracy has started to converge (the curve begins to flatten).

Adam is experiencing a similar phenomenon. 53.31% is teacher and 61.92% is evaluation pattern accuracy. Learning is also fast, and convergence is beginning to be seen here.

Out of the three cases examined, I chose Adam. Very similar results were obtained with RMSprop, but the difference in accuracy between the validation and training samples was more significant for Adam. Therefore, it can be assumed that we can continue teaching without the risk of over-learning. For a more accurate comparison, we could have run the SGD version until it started converging. However, since the goal of the teaching

process is not to last too long (time and resources), I rejected this algorithm.

2) Determining the learning rate

The next step was to test Adam with more learning rates. The default value for this algorithm is 10 ^ (- 3). It is worth changing this parameter on a logarithmic scale, so I compared 10 ^ (- 2), 10 ^ (- 3), 10 ^ (- 4) and 10 ^ (- 5).

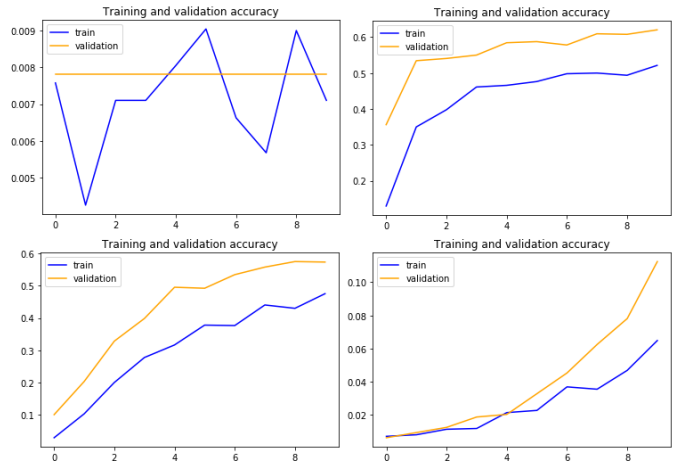


Figure 13: Teaching Process with10 ^ (- 2) (top left), 10 ^ (- 3) (top right), 10 ^ (- 4) (bottom left), and 10 ^ (- 5) (bottom right) learning rates

Of the four options, the default value (10 ^ (- 3)) performed best. Increasing the parameter significantly impaired prediction ability and learning at lower values slowed down. In the third case, the network began to converge with slightly lower accuracy and closer to the results for the training and validation sets than for 10 ^ (- 3). With the smallest parameter value, learning would be slow, and no sign of convergence could be seen. So, I chose the default learning rate.

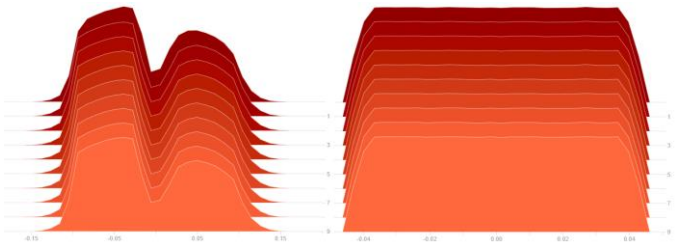


Figure 14: Distribution of gradients for highest (left) and lowest (right) learning rates. The horizontal axis shows the possible values, while the vertical axis shows the number of gradients with this value. Depth denotes each epoch.

It is worth looking at what was behind the drastic deterioration of the first case. If the graph on the right has gradient distribution, it is close to most extremes on the left. So, in every iteration, the changes are too big due to the high value of the learning rate, so the system could not learn.

3) the number of neurons in a layer

# neurons	Teacher acc.	Val acc.	Difference
512	53,31%	61,92%	8,61%
1024	54,26%	59,69%	5,43%
2048	51,94%	60,78%	8,84%

The comparison provided the following results. All three cases performed best by some metric, but the evaluation accuracy is the most significant, so I chose 512. This carries a lower risk of over-training due to its smaller size.

4) Change the number of layers

In the last comparison I varied the number of layers. In the first case, I connected the GAP output to the output of the entire network, without using a dense hidden layer. In the second case, I tested a level with the 512 neurons selected above. For the third, I added another layer with 512 neurons between the output and the output to increase the complexity. This added 300,000 parameters to the system (the network became more resource intensive).

To make the decision here, I taught the models using the early stopping technique. The point is that the process does not stop until there is improvement. Meanwhile, weights are constantly saved. If the system fails to improve within a certain grace period (3 epochs), the operation stops, and the previous best version is restored.

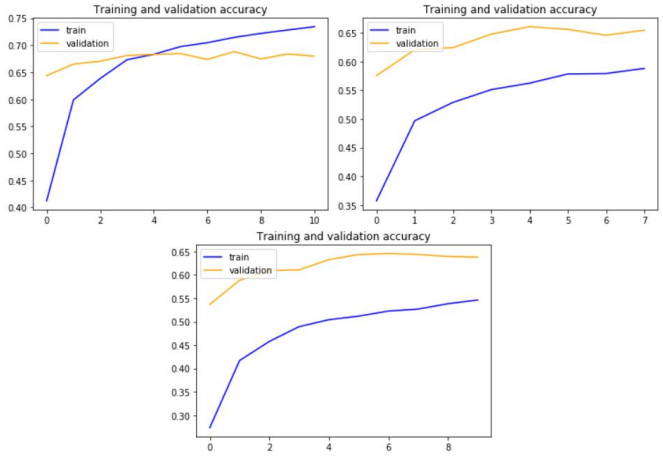


Figure 15: Improvement of accuracy for epochs with 0 layers (left), 1 layers (right), and 2 layers (bottom). Early stopping was used there.

The zero hidden layered solution performed best among the three, reaching 68.82%, so I chose this as the final structure.

B. Fine tuning of convolutional layers

So far in the model I have only taught the classifier placed on the pre-taught network. Once this process has been completed, it is also possible to dissolve and teach the last layers of convolution.

What happens then is that the general characteristics of these layers are slightly changed in order to better fit our classes. This is worth doing after the end-of-network module itself has been taught and its improvement rate has slowed down. Otherwise,

the large changes in weight in the classifier, which was still making a mistake at the beginning of the teaching, would ring up to convolutional layers, which would obliterate their previous knowledge.

The last layers of the pre-taught model are only worth unlocking. It is true for convolutional networks that the later a layer has, the more specific it is. The first ones encode very simple features, generally applicable to any image, so you shouldn't change them. The goal is to tailor the more specific features to our own departments.

The network contains a total of 157 layers, from the 140th level I released weights. In this way, 1.25 million teachable parameters were generated. On the model, I ran early stopping instruction with two orders of magnitude lower (10 ^ (- 5)) learning rates so that convolutional features would only adapt, and large changes would not lead to forgetting.

Teaching was able to improve for 4 epochs, with a final score of 70.5%. The system also performed with 70% accuracy on test samples, which validated the result.

VI. Component Packaging

The finished network needs to be formatted for use on an Android device. Here are the steps I have taken to do this.

A. TFLite Conversion

The model had to be converted to tflite format. Here's how I made it:

```
converter = tf.lite.TFLiteConverter.from_saved_model(
      modelsPath + '/' + modelDirNameToConvertFrom)

tflite_model = converter.convert()

with open(tflitePath + '/' + modelDirNameToConvertFrom
 + '.tflite', 'wb') as f:
  f.write(tflite_model)
```

During the process, it was important to note that not all operations implemented in TensorFlow are also implemented in TFLite. In order to prevent conversion problems, I checked the compatibility of the operations in my model (activation function, average pooling) with a compatibility chart. At the end of the process, the file with the tflite extension can be run on an Android device using an interpreter.

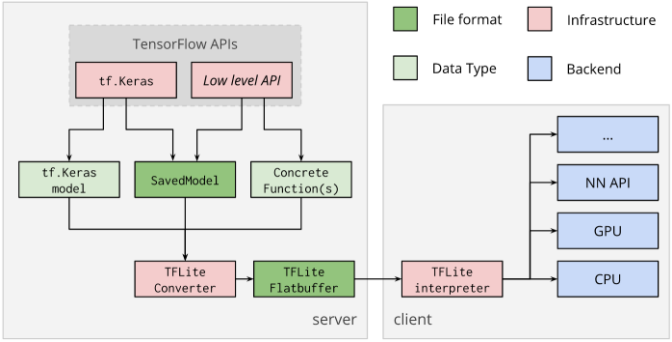


Figure 16: TensorFlow Model conversion to TFLite (Server) and Device usage (Client)[10]

B. Input and Output Data Formats

When teaching the network, I used 224x224 images with RGB color coding. Therefore, this is also required for the use of the model. The input is an array encoded by channels_last (index order: height, width, RGB channels). The value of one channel of a pixel is encoded in 32 bits (4 bytes). Values are interpreted as float32 (32-bit floating point) numbers on all three RGB channels, which should be between 0 and 1.

The output is an array of 158 float32 elements, each with a value between 0 and 1. Array elements represent the probabilities of the input image, the sum of the values of the elements is always 1.

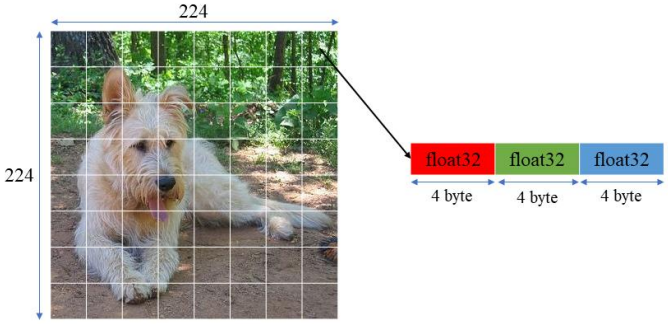


Figure 17: An illustration of the expected input of the model

C. Auxiliary Structures

I also needed to use an auxiliary structure to interpret the network's output. To do this, I created a label.txt file that has as many lines as the output of the model. The n^{th} output neuron information (unique identifier and name) is the one numbered in the n^{th} line of the file. That is how the results and classes can be mapped.

```
n02000009-Ragdoll
n02000010-Russian_blue
n02000011-Siamese
n02000012-Sphynx
n03000001-Affenpinscher
n03000002-Afghan_hound
n03000003-African_hunting_dog
n03000004-Airedale
n03000005-Akita
```

Figure 18: Few lines from the auxiliary structure

The model also has a file named model_info.txt. It contains general information about the network, including defining and interpreting the input and output data formats needed for use.

VII. Convert images on Android

The neural network running on the device does not accept any input, only RGB encoded 224x224 sized inputs. Whether an image is generated from the device's camera or loaded, it must be transformed. Here are the steps:

1)

Convert the input to the same width / height. First, the center of the image is calculated, where half of the smaller dimension is mapped in all directions, and the outside pixels are discarded. This produces an *N*x*N* output that retains the original aspect ratio.

2)

Resizing an image to produce a 224x224 image using matrix operations.

3)

From this, a ByteBuffer conversion is needed, which is a byte array containing image information. Array indices are in the order of height, width, RGB channels (due to the expected channel_last encoding). The size of the structure should be 224x224x3x4 bytes. The image size is 224x224, RGB coding requires 3 channels, and one pixel per channel requires 4 bytes, because we store 32-bit float values (1 byte = 8 bits; 4x8 = 32 bits). The bitmap's getPixels() method facilitates the creation of this structure.

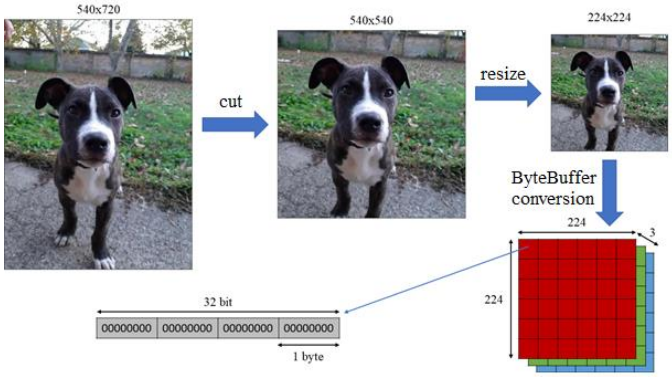


Figure 19: Image preprocessing steps during on-device inference

VIII. Summary

In this paper, I showed how I created an image recognizer model with transfer learning, how I converted it into TFLite, and used in an Android application.

Since I haven't dealt with Python and Keras / TensorFlow frameworks before, this was a good opportunity to get to know them.

I learned a lot during the task. I liked the fact that theory could be put to test right away. It was interesting to see how open the community was in deep learning. I think the attitude of developers / researchers sharing their results is a major contributor to the current development of the industry.

There are many ways to improve the image recognition model. The coverage of dog and especially cat breeds is far from complete, and additional databases could be used to expand the range of recognizable animals. Other combinations could have been tried to optimize the hyperparameters, or the Keras Tuner, which automatically does this, because in my opinion it would be better than 70%.

Overall, I enjoyed the preparation of my task. I feel I have achieved my goals and demonstrated the potential of merging the Android platform with machine learning tools through a usable application.

References

[1] LeNet5: LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). *Gradient-based learning applied to document recognition.* Proceedings of the IEEE, 86(11), 2278- 2324.

[2] Kaggle.com. (2019). *Stanford Dogs Dataset.* [online] Available at: https://www.kaggle.com/jessicali9530/stanford-dogs-dataset [Accessed: 12 Dec. 2019].

[3] Kaggle.com. (2019). *Cats and Dogs Breeds Classification Oxford Dataset.* [online] Available at: https://www.kaggle.com/zippyz/cats-and-dogs-breeds-classification-oxford-dataset/data [Accessed: 22 Nov. 2019].

[4] Kaggle.com. (2019). *cropped_dog_breeds.* [online] Available at: https://www.kaggle.com/maks23/cropped-dog-breeds [Accessed: 22 Nov. 2019].

[5] Kaggle.com. (2019). *face-dataset-celebrity.* [online] Available at: https://www.kaggle.com/tanvirshanto/facedatasetcelebrity [Accessed: 22 Nov. 2019].

[6] Google Developers. (2019). *Validation Set: Another Partition.* [online] Available at: https://developers.google.com/machine-learning/crash-course/validation/another-partition [Accessed: 23 Nov. 2019].

[7] Blog.keras.io. (2019). *Building powerful image classification models using very little data.* [online] Available at: https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html [Accessed: 1 Dec. 2019].

[8] Keras Documentation. (2019). *Applications.* [online] Available at: https://keras.io/applications/ [Accessed: 1 Dec. 2019].

[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei. (2009). *ImageNet: A Large-Scale Hierarchical Image Database.* In: IEEE Computer Vision and Pattern Recognition (CVPR).

[10] TensorFlow. (2019). *TensorFlow Lite converter. [online] Available at:* https://www.tensorflow.org/lite/convert [Accessed: 1 Dec. 2019].

AUTHOR

A. Fodor is a BSc Computer Engineering student at TU Budapest.