Department 07
Master Computer Science

Deep learning - Dog Breed Classification

Realization of an native Android app using deep learning algorithms

Alice Bollenmiller, Andreas Wilhelm
WS 17/18 IG
January 28, 2018

Contents

1. Introduction

In this chapter, a short introduction leads to the subject of Deep learning. Furthermore, the scope of this work and its points of referencee are described and localized.

1.1. Deep learning

In 2015, AlphaGo - a computer program developed by Google's DeepMind Group that was trained to play the strategic board game Go - was the first program to defeat the multiple European champion Fan Hui under tournament conditions. Back then, terms like Artifical Intelligence (AI), Machine Learning and Deep Learning were talked of, because those were the reason why a computer program was able to defeat a human being. One of the most frequent used techniques of AI is Machine Learning. It uses algorithms to parse data, process it and learn from it. The result is a prediction or determination as a conclusion of what was learnt from the dataset. Deep Learning - which is part of the Machine Learning techniques - sells its application particularly in the field of language and image processing.

In 1958, Frank Rosenblatt introduced the concept of the perceptron which is the fundamental idea of all Deep learning approaches. It consists of multiple artifical neurons which are coupled with weights and biases. In the case of a single-layer perceptron, the input nodes are fully connected to one or more output nodes. During the learning process the weights are adapted according to the learning progress. When such a structure is extended with layers, it becomes a multi-layer perceptron (MLP). This basic structure can be found in special neural network architectures e.g. Convolutional Neural Networks (CNN). CNNs which are frequently used for object detection and image/ audio recognition etc. consists of multiple neurons. When a neuron receives an input, it performs a dot production and adapts the weights. Because of their special structure, CNNs are able to detect local properties of an image. Basically, the network represents a differentiable score function which is applied on the raw image pixels and computes the class scores as an output.

1.2. Terms of Referencee

The problem of Deep learning architectures is their high performance requirements regarding computational power. Common frameworks for open source development in the field of deep learning which are discussed in section 2.1 introduced several models for the integration in mobile applications.

In order to overcome the above mentioned problem, optimizing methods will be combined with appropriate models which require less computing power and are suitable for mobile application integration. As a result of this work, a dog breed analyzer will be implemented. This mobile app will take a live camera stream as an input and determine the breed of the focused dog. The three highest probabilities of breeds will be shown by the app.

2. Methodological fundamentals

This chapter describes the most frequently used frameworks in deep learning for developing applications. Furthermore, common models for deep learning are introduced. The chapter closes listing key requirements for an appropriate dataset which increases the quality of the training results.

2.1. Common Frameworks for Deep Learning Applications

The demands on neural networks increase with the complexity of problems to solve. Concurrently, there's an expanding offer of deep learning frameworks with a varity of features and tools. The

most common used ones are represented in the following section.

2.1.1. Tensorflow

In 2015, the Google Brain Team introduced the most popular deep learning API Tensorflow which is an open-source library for numerical computation. Its current version 1.4.1 was released on December 8th, 2017. Tensorflow is primarily used for machine learning and deep neural network research. Based on the programming language Python, Tensorflow is capable of running on multiple CPUs and GPUs. Furthermore, C++ and R are supported by Tensorflow. Another feature is the possibility to generate models and export them as .pb file which holds the graph definition (GraphDef). The export is done by protocol buffers (protobuf) which includes tools for serializing and processing structured data. When loading a .pb file by protobuf, an graph object is created which holds a network of nodes. Each of these nodes represent an operation and the output is used as input for another operation. This concept enables an user to create self-built tensors. To simplify the usability, Tensorflow developed a high-level wrapper of the native API which is called Tensorflow Slim. Futhermore, in order to run Tensorflow on performance critical devices like e.g. mobile devices, two lightweight solutions of Tensorflow are available: Tensorflow Mobile and Tensorflow Lite. The latter one is an evolution of Tensorflow Mobile and still in developer mode. But both are predestinated for integration in mobile applications.

2.1.2. Keras

In order to simplify the utilization of Tensorflow the Python based interface Keras can be configured to work on top of Tensorflow. It allows building neural networks in a simple way and is part of Tensorflow.

2.1.3. Caffe

Another deep learning library is Caffe which was developed by Berkeley AI Research (BAIR). Based on C++ or Python, it focuses on modeling CNNs. An main advantage of Caffe is the offer of pretrained models available in the Caffe Model Zoo.

2.1.4. Torch and PyTorch

Besides Tensorflow and Caffe, Torch is another common deep learning framework. It was developed by Facebook, Twitter and Google. Based on C/C++, Torch supports CUDA for GPU processing. Like above mentioned frameworks, Torch facilitates the building of neural networks. The Python based version of Torch is available through PyTorch.

2.2. Common Models in Deep Learning Applications

State of the art models for image recognition in deep learning applications are based on the CNNs architecture. The most important ones are explained in this section.

The first model was LeNet-5 described in LeCun et al. (1998). As shown in figure 1, it contains seven layers in summary. These are two convolutional layers with a 5x5 filter each followed by a sub-sampling pooling layer. Two fully connected layers complete the architecture of the multilayer perceptron with its softmax function applied.

In 2012, a CNN based network called AlexNet pictured in Krizhevsky et al. (2012) wons the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) for the first time (figure 2). From this time, CNN became ubiquitous.

Figure 1: Model architecture of LeNet-5 (LeCun et al., 1998)

Figure 2: ImageNet Classification top-5 error (%) (He, 2016)

Through its immensely improved accuracy the network created new possibilities. This CNN consists of eight layers shown in figure 3. The first convolutional layer has a large 11x11 filter. Furthermore, there is one layer with a 5x5 filter and another three with a 3x3 filter, so in total 5 convolutional layers. Three pooling layers and three fully connected layers complete the architecture. For the first time, a so called Rectified Linear Unit (ReLU) was used which is capable of being spread across two GPUs.

Figure 3: Model architecture of AlexNet (He, 2016)

As shown in figure 2, the next improvements were two architectures called VGG as described in Simonyan and Zisserman (2015) and GoogLeNet which is presented in Szegedy et al. (2014). Introduced in 2014, they were considered as very deep neural networks. GoogleNet is today better known as 'Inception'.

VGG comprises 16 or 19 layers in total (figure 4. This network is characterized by its simplicity because it uses convolutional layers with a 3x3 filter which are stacked on top of each other. Size reduction is handled by pooling layers. At the end, there are three fully connected layers with the softmax function followed.

Figure 4: Model architecture of VGG19 (He, 2016)

Whereas, GoogLeNet has 22 layers (figure 5) and was the final winner in 2014. At the beginning, it has two convolutional layers followed by stacked Inception modules. Such modules consists of several parallel convolutional layers with different filter size and in addition one pooling layer as shown in figure 6. On the right side of this figure a convolutional layer with a 1x1 filter is pictured which is used to reduce feature depth. At the end,the output of these modules are concatenated and taken as new input for the next layer. The network ends with one fully connected layer.

Figure 5: Model architecture of Inception (Szegedy et al., 2014)

Figure 6: Inception modules (Szegedy et al., 2014)

Since 2015 it gets very deep with a new architecture named ResNet of **?**. On ILSVRC this network swept all competitiors in classification and detection. The network itself is designed as a 152 layer model. The ResNet consists of many residual network blocks (figure 7). Each of them has two convolutional layers with a 3x3 filter. Furthermore, there is a shortcut connection which will be used if the input and output dimension of this block are the same.
Especially for performance critical devices MobileNet was developed by Google as described in **?**. The network structure consists of depthwise seperable convolutional parts but also one standard convolutional layer at the beginning. The depthwise part is separated in a depthwise convolutional layer and a pointwise one which applies a 1x1 convolution. Each of this layers are followed by batchnormalisation and ReLu as shown in figure 8. In the depthwise convolutional section two convolutional layers are splitted in the one for filtering and in the one for combining. As an effect, the computation power and model size are massively reduced which results in better performance on low performance devices.

2.3. Key requirements for an appropriate dataset

Supervised learning tasks such as image recognition are based on operations where an output is taken as an input for the next node. Every raw pixel input is taken to compute an intermediate representation - a vector containing all learned information about the dataset. As a consequence, the training results are only as good as the dataset itself. For better accuracy its important to train a model on a variety of images for each object which should later be classified by the model.

Figure 7: ResNet block (**?**)

Figure 8: Standard convolution and depthwise seperable convolution (**?**)

It's recommended to take images of an object which were taken at different times, with different devices and at different places. Otherwise, the model will concentrate on other things like for example the background instead of details about the object itself. Therefore, a huge dataset is required especially for non pre-trained models. Training a model from scratch will require a huge dataset, a lot of computing power and time. Whereas pre-trained models only require a small dataset of about hundreds of images. For that reason, a pre-trained model will be used in this work.

3. Concept

First, this chapter describes the selection of the appropriate framework. Futhermore, the structure of the model which was used for classification is explained based on its architecture. The chapter closes with the class diagram of the mobile application.

3.1. Frameworks

As a deep learning framework, Tensorflow was used to retrain the model. This decision was mainly based on recommendations. Even companies like e.g. NVIDIA Corporation, Intel Corporation etc. use Tensorflow. It is one of the common frameworks for deep learning applications and also provides solutions for integration in mobile apps. Furthermore, Tensorflow provides a variety of tutorials for working with neural networks. Beside those advantages, there is a large community about Tensorflow talking about issues and solutions.

To run the tensor within a mobile application, the first approach was to use Tensorflow Lite which is still in development state. But many attempts resulted in corrupt models which caused the app to terminate. Because of this experiences Tensorflow Mobile was used to optimize the model for app integration.

3.2. Qualified models for mobile app integration

In Tensorflow Lite only InceptionV3 and MobileNet models are supported. With Tensorflow Mobile it's the same. There are different versions of MobileNet. They differ in the input image size and in the number of parameters which is also proportional to the size and needed computation power of the network.
While the Inception model was well known and established, the MobileNet is relatively new. InceptionV3 models gets an higher accuracy than MobileNets but MobileNets are more optimized on small size, low latency and low-power consumption which are important characteristics for mobile usage. (TensorFlow, nd)

In figure 9, the MobileNet and Inception is compared with each other and also with other popular models already mentioned before in section 2.2. It shows the Top-1 accuracy and the Multiply-Accumulates (MACs) which measures the number of fused multiplication and addition operations. The latter numbers reflect the latency and power consumption of the network and so in result the efficiency. For well comparison reasons between MobileNet and Inception, a MobileNet model with similiar accuracy to the Inception one was picked, in order to be more precisely than with the MobileNet_1.0_224. To check out the possibilities of MobileNet, another model, the MobileNet_-0.50_244, was selected, too.

3.3. Application based Architecture

This chapter describes the functions to a basic understanding of how the application works based on the application's architecture. It focuses on the important classes and methods of the mobile

Figure 9: Comparison of popular models (TensorFlow, nd)

application.
Tensorflow provides a mobile application for demo purposes. Because of the complexity the application was adapted to our needs. The approach was to understand how the application was implemented regarding the functionality of the camera, image input for the network and background classification besides live camera stream.

The ClassifierActivity is set as the launcher activity of the application. It extends the class CameraActivity and loads the CameraFragment which controls the camera view. The CustomTextureView enables the possibility to capture frames from a camera stream and process it. Because of this function, the CustomTextureView is part of the class CameraFragment. If an image is captured, the method onImageAvailable of the class ClassifierActivity is invoked. The classification itself is done by the TensorflowImageClassifier which implements the interface Classifier from the Tensorflow API. The results are displayed by the class RecognitionScoreView which implements the interface ResultsView.

4. Realization

In general, this chapter describes the methodical procedure of solving the problem mentioned in section 1.2. After describing the used dataset all required software and hardware components are explained in detail. Furthermore, the chapter leads through the installation steps of Tensorflow and the setup of Android Studio. Followed by the installation process the retraining of a pre-trained model is depicted. Afterwards, the re-trained model is tested and validated. The chapter ends with the description of the realization of the Android app.

4.1. Dataset

The most famous image dataset is ImageNet which is known from the ILSVRC2012. It contains overall about 14 million Images (ImageNet, 2010). This dataset is also used for nearly all pre-trained models. The dataset includes a set of 120 different dog breeds with about 150 images each breed. This dog dataset can be downloaded from the following Stanford website `http://vision.stanford.edu/aditya86/ImageNetDogs/`.
Another dataset with dogs is from Udacity and can be downloaded from `https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip`. It contains 133 dog breeds with a mean of 63 images each breed. The images in this dataset are completely different from the ImageNet dataset.

Because of the reason that the ImageNet dataset is already used to train the pretrained models, the last dataset is used for this project, which is fully unknown for any pretrained model.

4.2. Hardware environment

Different hardware setups are used for this project to compare the performance of the mobile application. At first the lowest one is a notebook with an Intel(R) Core(TM) i7-4720HQ CPU @ 2.60GHz and 16GB memory. The other one is a powerful desktop computer with an Intel(R) Core(TM) i7-6700K CPU @ 4.4 GHz and 16 GB memory. In this computer there is also a NVIDIA GeForce GTX 980 with CUDA capabilities version 5.2. It has 2048 CUDA cores running @ 1.291 GHz and 4 GB memory.
For good performance comparison of the mobile application also two different mobile phones are used. The first one is a Samsung Galaxy S4 (GT-I9515) with a Qualcomm Snapdragon S600 Quad-Core @1.9 GHz (32-bit) and 2 GB memory. The operating system is Android 5.0.1 (API 21) is running. The other one is a Motorola Moto X (2nd Gen, 2014) with a Qualcomm Snapdragon 801 Quad-Core @2.5 GHz (32-bit, ARMv7-ISA) and 2 GB memory. Here, the operating system is Android 6.0 (API 23).

4.3. Installation of software

This chapter describes all necessary steps for installing the software environment including Tensorflow.

4.3.1. Prerequisites

The software environment was set up on the Linux distribution Ubuntu 16.04 LTS. To install the software environment for Tensorflow, Python is required. Therefore, the current version of Python 3.6 was installed by default. Tensorflow also supports Bazel which was installed by following command.

```
1 sudo apt-get install openjdk-8-jdk
2
3 echo "deb [arch=amd64] http://storage.googleapis.com/bazel-apt stable jdk1.8" | sudo
   tee /etc/apt/          sources.list.d/bazel.list
4 curl https://bazel.build/bazel-release.pub.gpg | sudo apt-key add -
5
6 sudo apt-get update && sudo apt-get install bazel
7
8 sudo apt-get upgrade bazel
```

Listing 1: Bazel Installation

Futhermore, the package and environment management tool Anaconda was installed by the following steps: First, the Anaconda installer was downloaded from `https://www.anaconda.com/download/#linux`. During the installation process the prompts were answered by the default suggestions except the following prompt: ”Do you wish the installer to prepend the Anaconda3 install location to PATH in your /home/aw/.bashrc ? [yes—no]”. ”yes” was typed in and conda was tested using the ”conda list” command.

In the development of native Android Apps, Java is used as the programming language. Within the installation of Anaconda, the JDK in the version 8 was installed.
For using a GPU, further NVIDIA software must be installed which is in detail Compute Unified Device Architecture (CUDA) and the CUDA Deep Neural Network library (cuDNN). The requirements will be decribed in the following otherwise check out `https://www.tensorflow.org/install/install_linux#nvidia_requirements_to_run_tensorflow_with_gpu_support`.

Precaution, to use TensorFlow wit GPU support the odler CUDA Toolkit in version 8.0 and cuDNN in version 6.0 has to be installed!
Firstly validate that the GPU card support at least CUDA Compute Capability 3.0 which can be seen on `https://developer.nvidia.com/cuda-gpus`.
Secondly download CUDA Toolkit 8.0 GA2 on `https://developer.nvidia.com/cuda-downloads`. Pick the correct target settings and download the deb-file. An installation guide can also be downloaded, however all relevant steps are described here.
The kernel headers and development packages for the currently running kernel has to be installed using following command in listing 2.

```
sudo apt-get install linux-headers-$(uname -r)
```

Listing 2: Installation of linux-headers

Like in listing 3 shown, install CUDA.

```
sudo dpkg -i cuda-repo-ubuntu1604-8-0-local-ga2_8.0.61-1_amd64.deb
sudo apt-get update
sudo apt-get install cuda-8-0
```

Listing 3: Installation of CUDA

If a newer version of CUDA is already installed, remove it with prompts of listing 4

```
sudo apt-get remove cuda
sudo apt-get autoremove
sudo apt-key remove /var/cuda-repo-.../....pub  #fill in correct path/file
```

Listing 4: Remove CUDA

To verify the installation, firstly setup the verification environment with following commands (refer to listing 5).

```
export PATH=/usr/local/cuda-8.0/bin${PATH:+:${PATH}}

#To change the environment variables for 32-bit operating systems:
export LD_LIBRARY_PATH=/usr/local/cuda-8.0/lib\${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH
 }}
#To change the environment variables for 64-bit operating systems:
export LD_LIBRARY_PATH=/usr/local/cuda-8.0/lib64\${LD_LIBRARY_PATH:+:${
 LD_LIBRARY_PATH}}

cuda-install-samples-8.0.sh <dir>   #fill in directory for downloaded samples
```

Listing 5: Setup CUDA verification environment

Then compile the downloaded samples by changing to *NVIDIA_CUDA-8.0_Samples* and typing *make*. Thereafter execute *deviceQuery* which is placed somewhere under *NVIDIA_CUDA-8.0_Samples/bin/*. The output should look similar to that shown in figure 10. Important outcomes are that a device was found and that the final test passed.
In order to check if the system and the CUDA device are able to communicate correctly, execute *bandwidthTest*. The Output should look like in figure 11.
For downloading cuDNN on `https://developer.nvidia.com/cudnn`, a registration for the NVIDIA Developer Program is needed. Download the three deb-files of cuDNN 6.0 for CUDA 8.0 for your corresponding platform environment. Afterwards install it with the prompts of listing 6.

```
sudo dpkg -i libcudnn6_6.0.21-1+cuda8.0_amd64.deb
sudo dpkg -i libcudnn6-dev_6.0.21-1+cuda8.0_amd64.deb
sudo dpkg -i libcudnn6-doc_6.0.21-1+cuda8.0_amd64.deb
```

Listing 6: Installation of cuDNN

To verify that cuDNN is working correctly, compile and execute mnistCUDNN with following commands (listing 7).

```
aw@tux:~/Downloads/NVIDIA_CUDA-8.0_Samples/bin/x86_64/linux/release$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 980"
  CUDA Driver Version / Runtime Version          9.0 / 8.0
  CUDA Capability Major/Minor version number:    5.2
  Total amount of global memory:                 4029 MBytes (4224909312 bytes)
  (16) Multiprocessors, (128) CUDA Cores/MP:     2048 CUDA Cores
  GPU Max Clock rate:                            1291 MHz (1.29 GHz)
  Memory Clock rate:                             3505 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.0, CUDA Runtime Version = 8.0, NumDevs = 1, Device0 = GeForce GTX 980
Result = PASS
```

Figure 10: CUDA Verification - valid results from deviceQuery

```
aw@tux:~/Downloads/NVIDIA_CUDA-8.0_Samples/bin/x86_64/linux/release$ ./bandwidthTest
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: GeForce GTX 980
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     12908.5

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     13218.3

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     159783.0

Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
```

Figure 11: CUDA Verification - valid results from bandwidthTest

```
cp -r /usr/src/cudnn_samples_v6/ $HOME
cd $HOME/cudnn_samples_v6/mnistCUDNN
make clean && make
./mnistCUDNN
```

Listing 7: Verification of cuDNN

At the end of the outcome *Test passed!* should occur.
Lastly install the libcupti-dev library and export it's path with listing 8.

```
sudo apt-get install cuda-command-line-tools
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/extras/CUPTI/lib64
```

Listing 8: Verification of cuDNN

4.3.2. Tensorflow based on Python

First, there are some necessary dependencies to install before the actual installation can begin (refer to listing 9).

```
sudo apt-get install python3-numpy python3-dev python3-pip python3-wheel
```

Listing 9: Installing the dependecies for Tensorflow based on Python 3.n

In order to install Tensorflow with Anaconda, an Anaconda environment must be created using following command (refer to listing 10).

```
conda create -n tensorflow python=3.6
```

Listing 10: Creating an Anaconda environment

This command depends on the version of Python installed on the computer. Next, the created environment is activated through following line (listing 11).

```
source activate tensorflow
```

Listing 11: Activating the Anaconda environment

Futhermore, Tensorflow was installed through the Anaconda environment for CPU only by the following command (refer to listing 12).

```
pip install --ignore-installed --upgrade https://storage.googleapis.com/tensorflow/
 linux/cpu/   tensorflow-1.4.1-cp36-cp36m-linux_x86_64.whl
```

Listing 12: Installing Tensorflow through Anaconda

For GPU the tfBinaryURL was changed to `https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow_gpu-1.4.1-cp36-cp36m-linux_x86_64.whl`.

In order to validate the installation, a following short Python script was executed in the Anaconda environment (refer to listing 13).

```
# Python
import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print(sess.run(hello))
```

Listing 13: Installing Tensorflow through Anaconda

If the output is 'Hello, TensorFlow!', the installation was conducted sucessfully. Each time when working with Tensorflow, the environment has to be activated using the command in listing 11. As a consequence, the environment has to be deactivated when the work is done using the command 'deactivate'.

4.3.3. Tensorflow based on Bazel

Another way of working with Tensorflow is using Bazel. In order to install, refer to listing 1. After the Bazel installation is completed, the Tensorflow repository can be cloned by following command (refer to listing 14).

```
1 git clone https://github.com/tensorflow/tensorflow
```

Listing 14: Cloning the tensorflow repository

Now, the installation can be configured. The configuration and its prompts are shown in section A.1. If GPU usage is desired, some configurations might differ from the shown configuration steps. The differences are explained in section 4.3.1.

Furthermore, the Android SDK and NDK must be added to the workspace of Bazel. This is done by removing the comments in the WORKSPACE file which is located in the source directory of the Tensorflow repository and adapting the PATH variable according to the location of the SDK and NDK. NDK 16 which was released on November 2017 is incompatible with Bazel.

Then, the Tensorflow pip package can be built by Bazel using the command shown in listing 15 which also enables Streaming SIMD Extensions 2 introduced by Intel Corporation to process data with doubled accuracy. For GPU support the option '–config=cuda' was added to the build command.

```
1 bazel build -c opt --copt=-msse4.2 //tensorflow/tools/pip_package:build_pip_package
2 bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg
```

Listing 15: Building the Tensorflow pip package

Finally, the pip package is installed by executing the following command listing 16.

```
1 sudo pip install --upgrade /tmp/tensorflow_pkg/tensorflow-*.whl
```

Listing 16: Installing the pip package

Instead of using the placeholder '-*', the version can also be specified. Which Tensorflow version is used, can be determined by executing 'pip list |grep tensorflow' depending on the Python version. The version of Tensorflow used in this work is 1.4.1. The validation of the installation is done by executing the command shown in listing 13.

4.3.4. Installing Android Studio and its Delevopment Kit

As an app development environment the free IDE Android Studio in its version 3.0.1 was installed by downloading from `https://developer.android.com/studio/index.html`. After the archive file was extracted, the script studio.sh was executed. Required dependencies are installed by Android Studio itself. Futhermore, Android Studio automatically installs the necessary SDK and NDK. So, the SDK in the version 26.1.1 and the NDK in its version 16.1 were used. If Android Studio doesn't install the necessary Development Kits, the appropriate SDK and NDK version can be installed by selecting the SDK and NDK in the section File - Settings - Appearance and Behaviour - System Settings - Android SDK. Thereby, in order to avoid any collapses when running the app, it's important to check the used Tensorflow version in the dependency which has to correspond to the Tensorflow version which was used to retrain the model.

4.4. Building the models

In this section the build and train respectively retrain process are described. At the end a short comparison of execution time on the different hardware setup specified in section 4.2 is done.

4.4.1. Execution methods

After setting up the hole software environment whether with python or bazel or with GPU support or without, the models can be trained. For several reasons as already explained in section 2.3 and section 4.1 pretrained models are used for this project. So the next step now is to download such a pretrained model and retraining it. Certainly with the script used for retraining the desired pretrained model will be already downloaded automatically.

Generally on retraining all existing weights of the pretrained model will be leaved untouched except for the final layer which is retrained from scratch. Thus, the pretrained model with its primal tasks or labels is adapted to new labels respectively restricted to certain labels or even adapted to new tasks. This technique is also called transfer learning. (TensorFlow, 2017)
For easier use some variables has been defined which are shown in listing 17. Some of them must be changed depending on the used model, others are optionally. All these variables are used in the entire project for every execution of a python script or an executable built by bazel!

```
#for InceptionV3 models use:
INPUT_SIZE=299              #size of input layer
ARCHITECTURE=inception_v3   #name of architecture
INPUT_LAYER=Mul             #name of input layer
#OUTPUT_LAYER=softmax       #name of output layer (only pretrained model)
OUTPUT_LAYER=final_result   #name of output layer (after retraining)

#for MobileNet models use:
INPUT_SIZE=224              #size of input layer
VERSION=0.50                #version of MobileNet
ARCHITECTURE=mobilenet_${VERSION}_${INPUT_SIZE} #name of architecture
INPUT_LAYER=input           #name of input layer
#OUTPUT_LAYER=MobilenetV1/Predictions/Reshape_1 #name of output layer (only
 pretrained model)
OUTPUT_LAYER=final_result   #name of output layer (after retraining)

#optional
DOG_PATH=${HOME}/dog_photos   #path with images for retraining
RETRAIN_PATH=${HOME}/MobileInceptionRetrained   #path for all output data
TRAINING_STEPS=500          #how many training steps
LEARNING_RATE=0.01          #learning rate
```

Listing 17: Defining terminal variables

To retrain the model the python script *code/scripts/retrain.py* in the github-repo of this project is used and executed as shown in listing 18. This script is taken from the 'tensorflow-for-poets'-tutorial and is extended with some time measurings. All important parameters are in the listing mentioned before and explained later in this section. However for all possible options just execute *python -m scripts.retrain -h*.

```
python -m scripts.retrain \
  --bottleneck_dir=${RETRAIN_PATH}/bottlenecks_${ARCHITECTURE} \
  --how_many_training_steps=${TRAINING_STEPS} \
  --model_dir=${RETRAIN_PATH}/models_${ARCHITECTURE}/ \
  --summaries_dir=${RETRAIN_PATH}/training_summaries/${ARCHITECTURE}_${TRAINING_STEPS
 }/${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE} \
  --output_graph=${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/
 retrained_dog_graph_${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.pb \
  --output_labels=${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/
 retrained_dog_labels_${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.txt \
  --architecture=${ARCHITECTURE} \
  --image_dir=${DOG_PATH} \
  --learning_rate=${LEARNING_RATE}
```

Listing 18: Call of *retrain.py*

For the alternative execution method with the binary built by bazel, see listing 19.

```
bazel build tensorflow/examples/image_retraining:retrain && \
bazel-bin/tensorflow/examples/image_retraining/retrain \
  --bottleneck_dir=${RETRAIN_PATH}/bottlenecks_${ARCHITECTURE} \
  --how_many_training_steps=${TRAINING_STEPS} \
  --model_dir=${RETRAIN_PATH}/models_${ARCHITECTURE}/ \
  --summaries_dir=${RETRAIN_PATH}/training_summaries/${ARCHITECTURE}_${TRAINING_STEPS
 }/${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE} \
  --output_graph=${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/
 retrained_dog_graph_${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.pb \
  --output_labels=${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/
 retrained_dog_labels_${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.txt \
  --architecture=${ARCHITECTURE} \
  --image_dir=${DOG_PATH} \
  --learning_rate=${LEARNING_RATE}
```

Listing 19: Build and call of *retrain*

Initially the explanations of the parameters after that a detailed description about the process and some technical terms are following.
The first one *bottleneck_dir* is the path to cache bottleneck values in files. The *how_many_training_steps* defines the quantity of training steps. *model_dir* is the directory, where to download and extract the pretrained model. *summaries_dir* is the path, where additionally visualization data will be stored. In *output_graph* and *output_labels* the path and name of the new graph respectively new labels are defined. *architecture* represents the desired architecture. *image_dir* is the location of the image to retrain on. *learning_rate* specifies the learning rate during training.

At first this script downloads the pretrained model from tensorflow if it doesn't already exist. Then i loads the model info of the pretrained graph and sets up the graph internally.
Afterwards the function *create_image_lists* is called, which seperates all images from the image directory for training, validation and testing sets. The separation is dependent on the label name, because the script calculates a hash of it and according to that it gets its corresponding set label. The benefit is, that all existing images keeps in the same set even if later more files are added.

After this preparatives, bottleneck values for all images are calculated and cached on disk. Bottleneck is mostly used for the layer just before the final result layer. Because of reusing every image multiple times during training it spares a lot of time using the cached values as the first part until the bottleneck is constant.
This takes some time depending on the hardware setup and on the choosen model, which is indicated in figure 12. Of course the setup with GPU is the fastest with about 50 seconds for MobileNets and 300 seconds for InceptionV3 At this point a GPU is a great benefit. On average the powerful destop-CPU takes 4 times longer and the notebook even takes 6 times longer as with the GPU setup. The notebook takes on average 1.4 times longer than the desktop-CPU although the desktop-CPU theoretically is 1.6 times faster.
Listing 20 shows the workload of the GPU during calculating which can be showed by executing *nvidia-smi -l 1*. It is working at 38% capacity and uses about 3160 MiB memory. Whereas the CPU is only utilized at 110% of the maximum 800% because of four cores with hyper threading which can be seen in the top part of figure 13. The bottom part of this figure shows about 470% workload if calculating everything with desktop-CPU only.

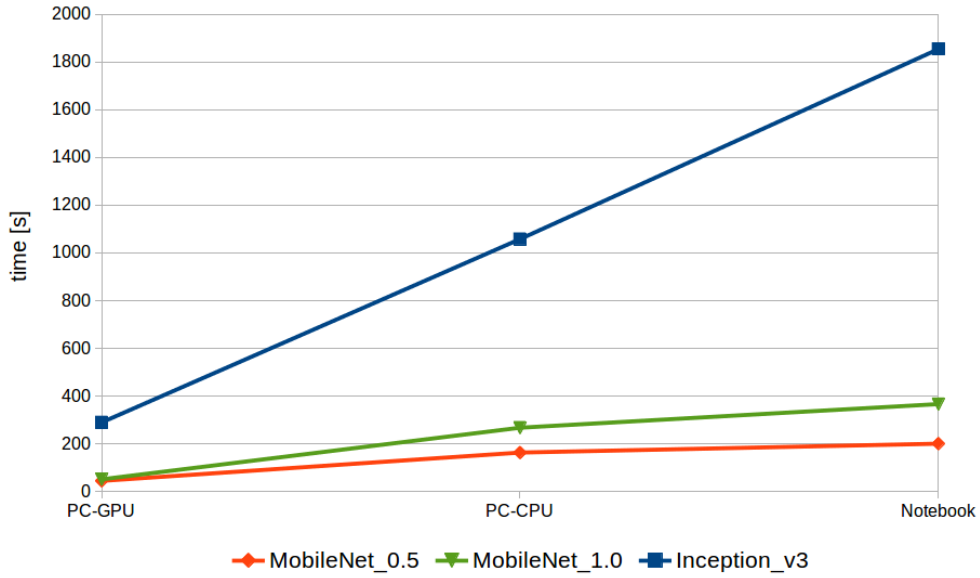


Figure 12: Total time creating bottlenecks

```
nvidia-smi -l 1
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 980     Off  | 00000000:01:00.0  On |                  N/A |
|  1%   30C    P2    68W / 196W |   3873MiB /  4029MiB |     38%      Default |
+-------------------------------+----------------------+----------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      1065      G   /usr/lib/xorg/Xorg                           584MiB |
|    0      1675      G   compiz                                        83MiB |
|    0     12867      G   ...-token=A79FC92A8A549F181FB5DBE83D71EFC2     31MiB |
|    0     26527      C   python                                      3159MiB |
```

Listing 20: Build and call of *retrain*

GPU calculating

```
top - 13:52:11 up 2 days,  2:19,  1 user,  load average: 0,52, 0,50, 0,70
Tasks: 242 gesamt,   1 laufend, 241 schlafend,   0 gestoppt,   0 Zombie
%CPU(s): 14,8 be,  0,9 sy,  0,0 ni, 84,2 un,  0,0 wa,  0,0 hi,  0,1 si,  0,0 st
KiB Spch : 16381648 gesamt,   488276 frei,  5314092 belegt, 10579280 Puff/Cache
KiB Swap:   974844 gesamt,   974844 frei,        0 belegt. 10501708 verfü Spch

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     ZEIT+ BEFEHL
17880 aw        20   0 31,781g 1,254g 396376 S 110,6  8,0   0:20.41 python
 1065 root      20   0 1137300 366304  88180 S  16,6  2,2  60:10.97 Xorg
```

CPU calculating all

```
top - 13:54:33 up 2 days,  2:21,  1 user,  load average: 1,05, 0,62, 0,71
Tasks: 245 gesamt,   2 laufend, 243 schlafend,   0 gestoppt,   0 Zombie
%CPU(s): 59,0 be,  0,9 sy,  0,0 ni, 40,0 un,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Spch : 16381648 gesamt,  1172768 frei,  4503052 belegt, 10705828 Puff/Cache
KiB Swap:   974844 gesamt,   974844 frei,        0 belegt. 11375376 verfü Spch

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     ZEIT+ BEFEHL
22270 aw        20   0 1760060 245380  49108 S 473,7  1,5   0:44.93 python
 1065 root      20   0 1137300 366304  88180 R   7,0  2,2  60:23.61 Xorg
```

Figure 13: CPU usage creating bottlenecks - Top: GPU - Bottom: CPU only

Once finished with creating the bottleneck values, training of the final layer begins. Each step random images of the training set will be fed into the final layer to get prediction. Those predictions are now used to update the final layer weigts through the back-propagation process. The script outputs the *train accuracy*, *cross entropy* and the *validation accuracy* each 10th step as can be

seen on listing 21, which is a shortened output. At the end a final test evaluation with the testing image set is run which results in the *final test accuracy*. Furthermore some time measurings are printed out. (TensorFlow, 2017)

```
INFO:tensorflow:2018-01-23 18:49:46.730940: Step 0: Train accuracy = 34.0%
INFO:tensorflow:2018-01-23 18:49:46.731069: Step 0: Cross entropy = 4.808121
INFO:tensorflow:2018-01-23 18:49:46.839474: Step 0: Validation accuracy = 4.0%
INFO:tensorflow:2018-01-23 18:49:47.309740: Step 10: Train accuracy = 27.0%
INFO:tensorflow:2018-01-23 18:49:47.309843: Step 10: Cross entropy = 4.646721
INFO:tensorflow:2018-01-23 18:49:47.356538: Step 10: Validation accuracy = 15.0%
INFO:tensorflow:2018-01-23 18:49:47.828190: Step 20: Train accuracy = 50.0%
INFO:tensorflow:2018-01-23 18:49:47.828293: Step 20: Cross entropy = 4.465392
INFO:tensorflow:2018-01-23 18:49:47.875599: Step 20: Validation accuracy = 32.0%
### shortened ###
INFO:tensorflow:2018-01-23 18:53:15.319134: Step 3990: Train accuracy = 91.0%
INFO:tensorflow:2018-01-23 18:53:15.319235: Step 3990: Cross entropy = 0.282067
INFO:tensorflow:2018-01-23 18:53:15.369910: Step 3990: Validation accuracy = 89.0%
INFO:tensorflow:2018-01-23 18:53:15.796850: Step 3999: Train accuracy = 96.0%
INFO:tensorflow:2018-01-23 18:53:15.796950: Step 3999: Cross entropy = 0.204091
INFO:tensorflow:2018-01-23 18:53:15.844287: Step 3999: Validation accuracy = 89.0%
INFO:tensorflow:Final test accuracy = 91.0% (N=818)
INFO:tensorflow:Froze 2 variables.
Converted 2 variables to const ops.

bottleneck time: 290.75745s
Evaluation training time: 48.22647s
Evaluation total time: 209.29127s
```

Listing 21: Output of *retrain.py*

For explanation, *train accuracy* shows the percentage of correctly labeled images in the current training batch. The *validation accuracy* is also the percentage of correctly-labeled images but now on random images from the validation set. *Cross entropy* is a loss function that indicates how well the learning process is progressing. (TensorFlow, 2017)

If during training the error ”Label xxx has no images in the category validation” occur, renaming this label names solves this issue because another hash will be calculated and the separation of the sets changes.

If using both, python and bazel execution methods the same version of TensorFlow has to be choosen, otherwise compatibility issues can occur! Especially the bazel one which is by default the newest version because of the master branch of github.

During the retrain process the outputted accuracies, cross entropy and some more data are written to the summaries directory and can be visualized with TensorBoard which is automatically installed with TensorFlow. To start it execute command in listing 22.

```
tensorboard --logdir ${RETRAIN_PATH}/training_summaries/${ARCHITECTURE}_${
 TRAINING_STEPS}/ &
```

Listing 22: Build and call of *retrain*

To validate the retrained model, see section 4.5.

Now, the graph of the model can also be visualized in TensorBoard. The retrained graph of the MobileNet model is depicted in figure 14 including the training nodes. In the appendix under section **??** is a more detailed one with all convolutional layer and under section **??** the view on the depthwise convolution containing its depthwise and pointwise convolutional layers as described in section 2.2.
The whole retrained graph of the MobileNet model is depicted in the appendix under section **??**. Furthermore, under section **??** a detailed view on a mixed_layer and under section **??** it's specific operations within a convolutional layer.

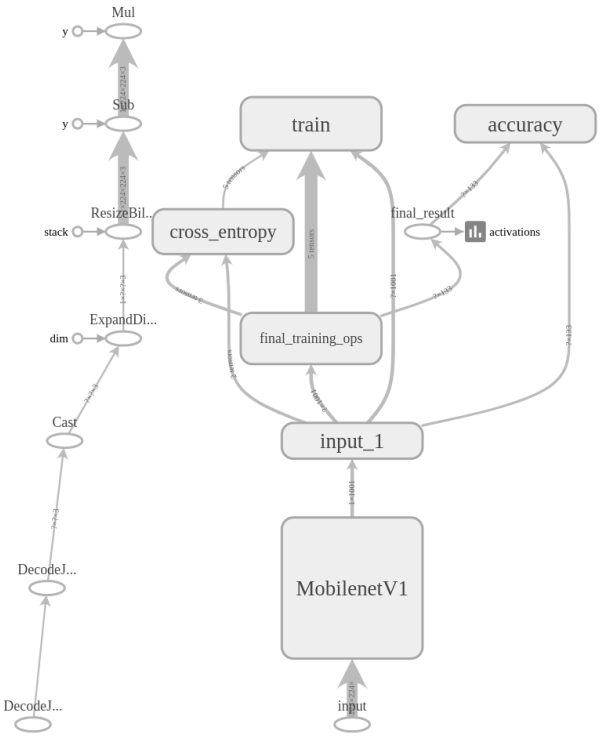


Figure 14: MobileNet retrained graph

Playing around with the parameters *how_many_training_steps* and *learning_rate* can improve accuracy. To measure the performance of the network, the validation accuracy is a well indicator because these images aren't used for training. Even better to check the performance of the classification task is the evaluation test at the end of the retraining process, because those images are entirely unknown to the model.
"If the train accuracy is high but the validation accuracy remains low, that means the network is overfitting and memorizing particular features in the training images that aren't helpful more generally." (TensorFlow, 2017)

In figure 15 the differences of retraining the InceptionV3 model with 5000 training steps and different learning rates is shown. For MobileNets it looks basically similar. The final results of trying out different training steps and learning rates are described in section 5.
As result of the retraining process the model is saved in the predefined file. So theoretically the models are now ready to use for inference.

However, to use the model in the mobile app, the DecodeJpeg Operation has to be stripped from the retrained model, because the Inference Interface of TensorFlow Mobile currently doesn't support it. This could be done by the *strip_unused.py*-script of the official tensorflow github repository.

Nevertheless it is recommend to optimize the pretrained graph for mobile inference to achieve the best tradeoff between accuracy, latency, size and performance. There are several ways and different options to do the optimization.

The first one, which will be called in following as '*opt1*', is only to run the *optimize_for_inference*-script of the official tensorflow github repository. The corresponing call is shown in listing 23. This script removes all nodes that aren't needed anymore like all training operations, strips unused parts,

Figure 15: InceptionV3 with 5000 training steps and different learning rates

removes debug informations, folding batch normalization ops into the pre-calculated weights and fusing common operations into unified versions.

```
python -m tensorflow.python.tools.optimize_for_inference \
  --input=${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/
 retrained_dog_graph_${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.pb \
  --output=${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/
 opt1_retrained_dog_graph_${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.pb \
  --input_names=${INPUT_LAYER} \
  --output_names="final_result"
```

Listing 23: Call of *optimize_for_inference.py*

The next attempt is to use the already optimized graph of *opt1* and additionally quantizes the graph by rounding the weights. Using the script of tensorflow-for-poets-2-tutorial and calling it as shown in listing 24). This will make the model compressible, which will reduce the model size significantly, thus every mobile app compresses the package before distribution.

```
python -m scripts.quantize_graph \
  --input=${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/
 opt1_retrained_dog_graph_${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.pb \
  --output=${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/
 opt2_retrained_dog_graph_${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.pb \
  --output_node_names=final_result \
  --mode=weights_rounded
```

Listing 24: Call of *quantize_graph.py*

To compare the improvement, use the gzip command in listing 25 on the model of *opt1* and also on this new one.

```
gzip -c ${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/
 retrained_dog_graph_${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.pb > ${
 RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/retrained_dog_graph_${
 ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.pb.gz

gzip -l ${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/
 retrained_dog_graph_${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.pb.gz
```

Listing 25: Call of gzip

The detailed result starting with the *opt1* output and then ending with the *opt2* output is shown in listing 26. So the *opt2* model is 65,4% lower in size and that's independent of the model used. As example with the InceptionV3 model the size is dropped from about 88 MB to only 24 MB which is a great improvement.

```
compressed  uncompressed   ratio uncompressed_name
81839945 88222145 7,20% /home/aw/MobileInceptionRetrained/graphs/inception_v3_4000/
 opt1_retrained_dog_graph_inception_v3_4000_0.03.pb

compressed  uncompressed   ratio uncompressed_name
24164803 88224397 72,60%   /home/aw/MobileInceptionRetrained/graphs/inception_v3_4000
 /opt2_retrained_dog_graph_inception_v3_4000_0.03.pb
```

Listing 26: Output of gzip

These first two ways of optimization are described in the tensorflow-for-poets-2-tutorial.
The next two attempts are using *transform_graph* of the official tensorflow github repository. Unfortunately there exists none python version of that tool, so it has to be built with bazel first.
The tool *transform_graph* has an parameter, called *tranforms*, which defines all operations to be executed. Calling the commands in listing 27 strips all unused nodes and folding batch normalization ops, so very similar to *opt1*. This will be *opt3* in the following content.

```
bazel build tensorflow/tools/graph_transforms:transform_graph && \
bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
--in_graph=${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/
 retrained_dog_graph_${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.pb \
--out_graph=${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/
 opt3_retrained_dog_graph_${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.pb \
--inputs=${INPUT_LAYER} \
--outputs='final_result:0' \
--transforms='
   strip_unused_nodes(type=float, shape="1,${INPUT_SIZE},${INPUT_SIZE},3")
   remove_nodes(op=Identity, op=CheckNumerics)
   fold_old_batch_norms'
```

Listing 27: Build and call of *transform_graph*

With *opt4* the command in listing 28 has been extended by *fold_old_batch_norms*, *round_weights* and *sort_by_execution_order*. Thus this one also rounds the weights, it is also compressible, which is similar to the *opt2* one.

```
bazel build tensorflow/tools/graph_transforms:transform_graph && \
bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
--in_graph=${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/
 retrained_dog_graph_${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.pb \
--out_graph=${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/
 opt4_retrained_dog_graph_${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.pb \
--inputs=${INPUT_LAYER} \
--outputs=final_result:0 \
--transforms='
   strip_unused_nodes(type=float, shape="1,${INPUT_SIZE},${INPUT_SIZE},3")
   remove_nodes(op=Identity, op=CheckNumerics)
   fold_batch_norms
   fold_old_batch_norms
   round_weights(num_steps=256)
   sort_by_execution_order'
```

Listing 28: Build and call of *transform_graph*

There are several helper tools, two of them are definitely worth to introduce here right now.

The first one, is the *graph_pb2tb.py*-script, again from the tensorflow-for-poets-tutorial. It generates a summary file, similar than during retraining process, except that only the data of the graph is stored for visualization with TensorBoard. So after optimizing the graph and calling the command of listing 29 the graph of the MobileNet model should look like depicted in figure 16 which is completely stripped of unused nodes as the training stuff and thus optimized for inference. The InceptionV3 model is in the appendix under section **??**.

```
python -m scripts.graph_pb2tb ${RETRAIN_PATH}/training_summaries/${ARCHITECTURE}_${
 TRAINING_STEPS}/${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}/retrained \
  ${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/retrained_dog_graph_${
 ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.pb
```

Listing 29: Call of *graph_pb2tb.py*

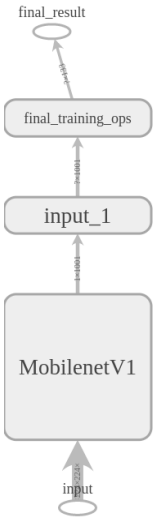


Figure 16: Optimized graph of a MobileNet model

The other tool is *summarize_graph* of the official tensorflow github repository, which has to be built by bazel. After building and calling command of listing 30 a short summary of the model is printed out which includes useful information like the shape, the name of input and output layers or quantity of constants. An example with the optimized retrained InceptionV3 model is shown in listing 31.

```
bazel build tensorflow/tools/graph_transforms:summarize_graph && \
bazel-bin/tensorflow/tools/graph_transforms/summarize_graph \
   --in_graph=${RETRAIN_PATH}/graphs/${ARCHITECTURE}_${TRAINING_STEPS}/
 retrained_dog_graph_${ARCHITECTURE}_${TRAINING_STEPS}_${LEARNING_RATE}.pb
```

Listing 30: Build and call of *summarize_graph*

```
Found 1 possible inputs: (name=Mul, type=float(1), shape=[1,299,299,3])
No variables spotted.
Found 1 possible outputs: (name=final_result, op=Softmax)
Found 22040882 (22.04M) const parameters, 0 (0) variable parameters, and 0
 control_edges
Op types used: 202 Const, 94 BiasAdd, 94 Conv2D, 94 Relu, 11 Concat, 9 AvgPool, 5
 MaxPool, 1 Add, 1 MatMul, 1  Placeholder, 1 PlaceholderWithDefault, 1 Reshape, 1
 Softmax
```

Listing 31: Output of *summarize_graph* on a retrained and optimized InceptionV3 model

4.5. Output Tests and Validation

In order to test and validate the output of the model regardless of whether the pretrained, retrained or optimized one is used the python script named *label_image*.py of the 'tensorflow-for-poets'-tutorial was applied. Three variables has to be defined differently which is shown at the beginning of listing 32. Depending which model is used for labeling an image, the first or second section must be applied.

```
#for InceptionV3 models use:
INPUT_SIZE=299       #size of input layer
INPUT_LAYER=Mul      #name of input layer
OUTPUT_LAYER=softmax #name of output layer (only for pretrained model otherwise '
 final_result' or as defined in retrain-script)

#for MobileNet models use:
INPUT_SIZE=224       #size of input layer
INPUT_LAYER=input    #name of input layer
OUTPUT_LAYER=MobilenetV1/Predictions/Reshape_1  #name of output layer (only for
 pretrained model otherwise 'final_result' or as defined in retrain-script)

python -m scripts.label_image \
--graph=${HOME}/retrained_graph.pb \
--labels=${HOME}/retrained_labels.txt \
--output_layer=final_result \
--input_layer=${INPUT_LAYER} \
--image=${HOME}/dl/label_image_pics/Affenpinscher_00001.jpg \
--input_width=${INPUT_SIZE} \
--input_height=${INPUT_SIZE}
```

Listing 32: Call of *label_image.py*

The corresponding command based on Bazel is shown in listing 32 where again the three variables need to be specified. The call itself is fundamentally the same. However, the executable binary has to be built before which is shown in listing 33.

```
bazel build tensorflow/examples/label_image:label_image && \
bazel-bin/tensorflow/examples/label_image/label_image \
   --graph=${HOME}/retrained_graph.pb \
   --labels=${HOME}/retrained_labels.txt \
   --output_layer=final_result \
   --input_layer=${INPUT_LAYER} \
   --image=${HOME}/dl/label_image_pics/Affenpinscher_00001.jpg \
   --input_width=${INPUT_SIZE} \
   --input_height=${INPUT_SIZE}
```

Listing 33: Build and call of *label_image*

If retraining of the model was conducted successfully, the output of *label_image* consists of five labels containing the highest values and corresponding accuracy which are returned by the model (refer to listing 34).

```
Evaluation time (1-image): 0.350s

001 affenpinscher 0.9940035
038 brussels griffon 0.002003342
042 cairn terrier 0.0008437461
100 lowchen 0.00018292216
099 lhasa apso 0.00012668292
```

Listing 34: Output of *label_image.py*

4.6. Implementation of an native Android App

This subchapter describes the processing of the camera input stream and classifying the particular images. Because of the vast extent of the application, the focus lies on the explanation of how the app works and on its important implementation parts.

When clicking on the icon of the app, a camera view showing results in the bottom part is loaded. In the background, the ClassifierActivity is set as the launcher activity in the AndroidManifest.xml which is a file containing all configuration settings for the app. When an activity is loaded for the first time, the onCreate method is invoked. Because the ClassifierActivity extends the CameraActivity the onCreate method of the CameraActivity is executed and loads the layout activity_camera.xml consisting of a container and a result field. Next, the setFragment method of the class CameraActivity is invoked if permission for the camera is given. This causes the replacement of the container with the adapted fragment according to the actual size and orientation of the screen. When the size is set, the method onPreviewSizeChosen is invoked where a TensorFlowImageClassifier is instantiated with following input parameters: MODEL_FILE, LABEL_FILE, INPUT_SIZE, IMAGE_MEAN, IMAGE_STD, INPUT_NAME, OUTPUT_NAME. Details about the classification itself follow later in the section.

When the fragment was replaced by the CameraActivity, an instance of the class CameraFragment was created. Next, the method onViewCreated of the class CameraFragment is called and a texture view is created. The latter class is part of the Android API and used for frame capturing from an image stream as OPENGL ES texture. Thereby, the most recent image is stored within a texture object. This object is observed by a listener which invokes the method onSurfaceTextureAvailable when the texture object is available. Within this method the camera is opened by the method openCamera(width, height) given the width and height of the camera preview (Android, 2017). First, the method openCamera sets the camera parameters within setUpCameraOutputs e.g. sensorOrientation, previewSize, cameraId etc. In order to classifiy a given OPENGL ES texture, the coordinate column vectors of the texture must be transformed into the proper sampling location in the streamed texture. So, the matrix has to be prepared with the correct configuration which happens in the method configureTransform. The next important step is done by the camera manager which opens the camera by invoking the method openCamer(cameraId, stateCallback, backgroundHandler). The id represents the specific camera device whereas the stateCallback is necessary to manage the life circle of the camera device and to handle different states of the camera device. The backgroundHandler ensures that the classification is done in background mode. When the camera is opened, the camera preview is started by the method createCameraPreviewSession. Within this method the preview reader is initialized and set to read images from the ImageListener which observes whether an image is available. Therefore, an Imagelistener which was instantiated by the CameraActivity is transfered to the CameraFragment.

When an image is available the method onImageAvailable of the class ClassifierActivity is invoked. The image is read by the preview reader and stored in an object. First, the image is preprocessed meaning transfered into planes, stored as bytes, cropped and stored as a Bitmap which is an Android graphic. Google provides an Tensorlflow Mobile API to run tensors on a performance critical device such as mobile devices. To use this API, the dependency has to be added to the build.gradle file as shown inlisting 35.

```
allprojects {
   repositories {
      jcenter()
   }
}

dependencies {
   compile 'org.tensorflow:tensorflow-android:+'
}
```

Listing 35: Tensorflow API in build.gradle

When including the API, Tensorflow's class Classifier can be used to classify images. Therefore, the TensorFlowImageClassifier which was initialized before invokes the method recognizeImage(croppedBitmap) on the preprocessed image. Within this method, the image data is transfered

from int to float (refer to listing 39). Then, the previous initialized object of the TensorflowInferenceInterface calls the method feed to pass the float values to the input layer of the model. Afterwards, the inferenceInterface invokes the method run with the specified output layer in order to process the classification. Subsequently, the output data is fetched by calling fetch on the output layer which is depicted in listing 36.

```

// Copy the input data into TensorFlow.
Trace.beginSection("feed");
inferenceInterface.feed(inputName, floatValues, 1, inputSize, inputSize, 3);
Trace.endSection();

// Run the inference call.
Trace.beginSection("run");
inferenceInterface.run(outputNames, logStats);
Trace.endSection();

// Copy the output Tensor back into the output array.
Trace.beginSection("fetch");
inferenceInterface.fetch(outputName, outputs);
Trace.endSection();
```

Listing 36: Classifying images by the inferenceInterface

Afterwards, the results are reordered according to the highest probability and returned to the result field for displaying the output.

4.7. Deployment and Validation

In order to get the model running in the application, a few things must be adapted. First, the model in the format of a .pb file and its retrained labels as a text file must be imported. Therefore, a assets folder was created where the model and labels were placed. Next, the path of the model and the labels file must be specified like pictured in listing 37.

```
// Input shapes
private static final int INPUT_SIZE = 224;
private static final int IMAGE_MEAN = 128;
private static final float IMAGE_STD = 128.0f;

// Input Layer
private static final String INPUT_NAME = "input";

// Output Layer
private static final String OUTPUT_NAME = "final_result";

// Model Name from Assets
private static final String MODEL_FILE = "file:///android_asset/
opt4_retrained_dog_graph_mobilenet_0.50_224_700_0.007.pb";


// Label Name from Assets
private static final String LABEL_FILE = "file:///android_asset/
retrained_dog_labels_mobilenet_0.50_224_700_0.007.txt";
```

Listing 37: Including the model in the mobile application

For MobileNet models the default settings are shown in the mentioned listing. Otherwise, if an InceptionV3 is used, the settings must be adapted like shown in listing 38.

```
private static final int INPUT_SIZE = 299;
private static final int IMAGE_MEAN = 128;
private static final float IMAGE_STD = 128.0f;
private static final String INPUT_NAME = "Mul:0";
private static final String OUTPUT_NAME = "final_result";
```

Listing 38: Setup for InceptionV3

The input size defines the size of the input layer of InceptionV3 which is 'Mul'. Whereas the input layer for MobileNet is specified as 'input'. The IMAGE_MEAN and IMAGE_STD values are needed for converting the image data from integer into float values. The conversion is shown in listing 39.

```
for (int i = 0; i < intValues.length; ++i) {
   final int val = intValues[i];
   floatValues[i * 3 + 0] = (((val >> 16) & 0xFF) - imageMean) / imageStd;
   floatValues[i * 3 + 1] = (((val >> 8) & 0xFF) - imageMean) / imageStd;
   floatValues[i * 3 + 2] = ((val & 0xFF) - imageMean) / imageStd;
 }
```

Listing 39: Conversion of image data integer to float

Futhermore, the output layer has to be specified. For MobileNet and InceptionV3, this was set to 'final_result'.

After setting up the application with appropriate values, the application can be built and run. If the application is running successfully and doesn't terminate, the settings are correct. If the result view contains results, everthing's working. If the results are below 0.1, the threshold in the class TensorflowImageClassifier has to be adapted. It's default setting is 0.01f which means only relevant results higher than 0.01 are shown in the result field. As a consequence, bad results given by the network aren't shown in general. In this case, we optimized the model again instead of lower the threshold. Just for the validation of a correct working app, the threshold was set to 0.001f.

5. Evaluation

As described in section 4.4 the accuracy can be improved by playing around with the training steps and the learning rate. It's a tradeoff between good results and overfitting, so that the model is too specialized on the training set.
The best values for the MobileNet0.5 model are achieved with 700 training steps and a learning rate of 0.007.
For the MobileNet1.0 model it was hard to find really well values. So the best here are 500 training steps and a learning rate of 0.01.
The final Inception model was retrained with 4000 training steps and a learning rate of 0.03.
For all models the *accuracy*- and *cross entropy*-diagram from TensorBoard is depicted in the appendix under section A.3 (figure 20 till figure 23). Exact these models are now used for further work.

In oder to compare the different optimization attempts, a bash script named *eval_label-image.sh* was written. It's placed in the github repository of this project in the directory *code/scripts*. The script takes a choosen set of 20 images (works/Evaluierung/eval_label_image_pics) and calls internally the *label_image.py* script to label them. The images, which partly are very difficult to classify, are picked from both dog datasets mentioned in section 4.1. In an Excel-sheet the results are averaged which is depicted in detail in the appendix under section A.4 (table 2 till table 4) and graphically in figure 17.

Because of the great benefit with the compression (refer to section 4.4) and the fact that the accuracy and the classification time of *opt2* and *opt4* isn't dramatically worser in contrast to the others, these two attempts are a good choice. Looking exactly at the tables, *opt4* is a little bit more accurate, so this is the best one and will be used for further comparisons.

For evaluating the different model architectures the last chart (figure 17) was represented as a scatter plot, which is shown in figure 18. So in this plot the upper left corner would be the best tradeoff

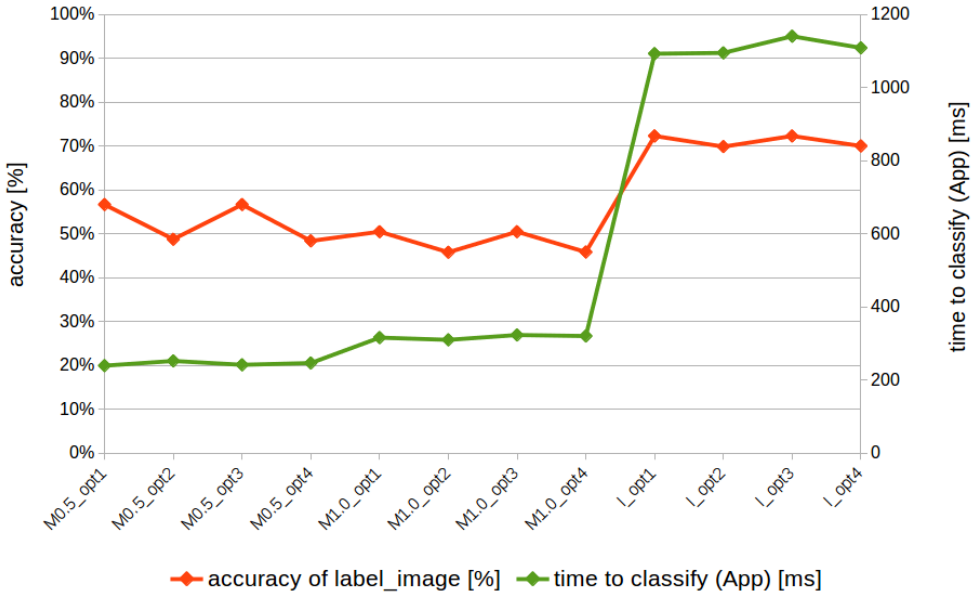


Figure 17: Accuracy and time to classify of different optimization attempts

between accuracy and performance of the mobile app. Thus one of the two MobileNets would be selected.

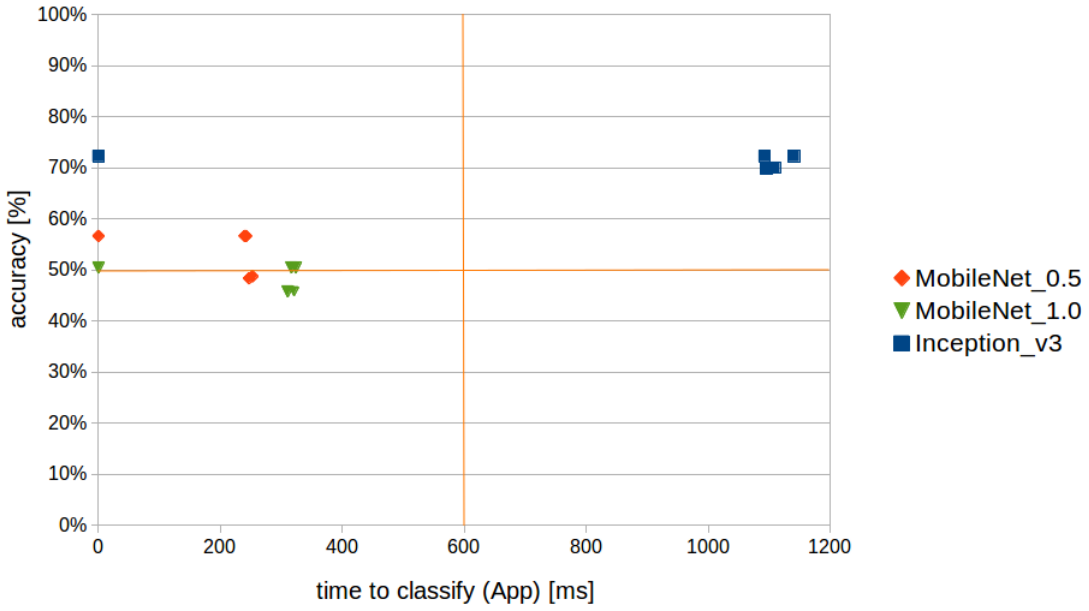


Figure 18: Accuracy in relation to classification time of different models

However for a more accurate evaluation of the models, some more facts have to be considered, which is shown in table 1 in numbers and graphically in figure 19.
The table contains the overall accuracy, the quantity of how many images are classified with a wrong label, how long the label_image on the computer took, how long it takes to load the model on the app, how long the classification on the app took and finally the compressed model size. Except the quantity of misclassified images and the model size, all values are shown in the graph.

Model	accuracy [%]	misclassified	labelImage [ms]	LoadModelApp [ms]	classifyApp [ms]	size [MB]
MobileNet0.5	48%	8	45	70	247	1.8
MobileNet1.0	46%	8	76	120	321	4.8
InceptionV3	70%	3	285	491	1108	24.1

Table 1: Values for final evaluation of model architecture

As expected the InceptionV3 model is the most accurate one, indeed it need a lot of performance, too.
A little surprise was the results between both MobileNets because the MobileNet_0.5 with less parameters is more accurate, has a smaller model size and further is in all tasks about 1.7 times faster than the other one.
The differences between MobileNet_0.5 and InceptionV3 are essential. On the one side InceptionV3 is about 1.4 times more accurate and classifies all images correctly except three images. On the other side the model size is much more higher. Furthermore Inception lags all the time because it's about six times slower, which is a disaster for responsiveness and usability. Especially for a mobile app it's more significant to run well on many devices even if the accuracy suffers a little bit. Additionally InceptionV3 also needs more time for retraining (refer to section 4.4).
All in all the MobileNet_0.5 is with this hardware setup the best tradeoff between accuracy and latency, model size and performance.

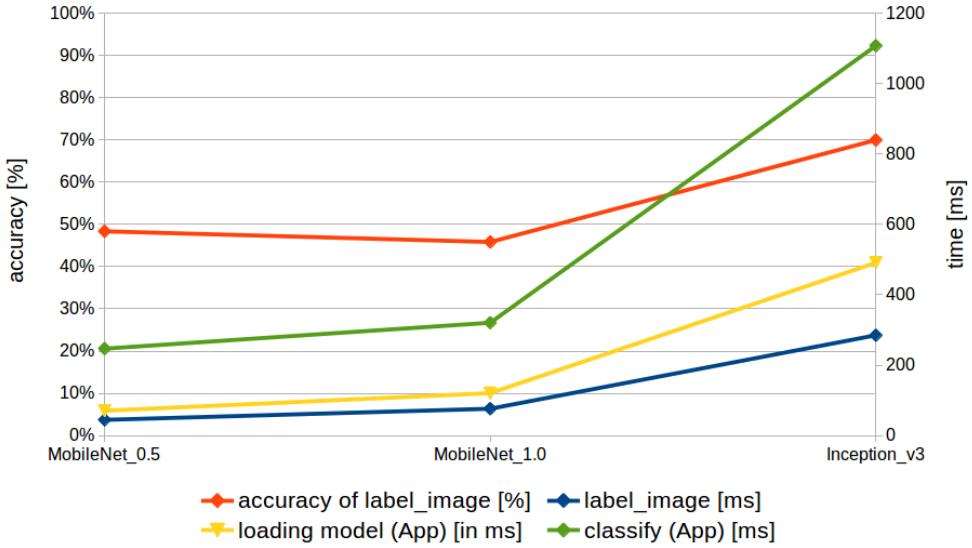


Figure 19: Graph for final evaluation of model architecture

6. Conclusion

At the beginning, a lot of research was done to answer the general question of 'how can a network be integrated in a mobile application and run on a performance critical device'. During the research, Tensorflow Lite was chosen as an API which enables running a network on an Android mobile phone. Because Tensorflow supports the models MobileNet and Inception specifically for mobile application usage, both were chosen. Based on this decision, both models were investigated and a comparison of both of them was incorporated into this work. After determining which framework and models are suitable for mobile applicaton support, the installation began. Because

of a good documentation the installation of Tensorflow and its necessary dependencies was done quickly. Following the tutorials the models were retrained on dog images. For a first, the optimization of the models to its full degree was omitted and the focus was to include the optimized models in the mobile application for running. Therefore, Tensorflow Lite was used to convert the .pb file to a .tflite file. The conversion was conducted successfully, but the produced .tflite file caused the app to terminate. After proving the versions of all used APIs and reconverting the models several times, the app still collapsed without throwing an error. Even after optimizing, rounding, quantizing the models with different commands, the app failed to run. In addition, the tutorials and documentation are kept short on this subject. Because of this behaviour, Tensorflow Mobile was used instead of Tensorflow Lite which is still in development mode. After integration of the .pb file and its corresponding label file, the Tensorflow version had to be adapted. If adding '+' in the dependency section for Tensorflow instead of the specified version, Android Studio installs the most recent one. Futhermore, after integrating an InceptionV3 model, no results were displayed while the app was running. Therefore, the threshold was adapted to a lower value in order to display low results from the network. Later, the threshold was set to 0.1f and the models were optimized. Next, the models were optimized to their full degree based on varying the learning rate in relation to the training steps. Afterwards, the models were comprised and loaded into the mobile application. The result are two applications, one including MobileNet and another containing InceptionV3. In the last step, the models were compared based on performance, time expenditure for producing an optimized model and quality in accuracy. Especially for the time related evaluation, marker were placed in the scripts to measure the time needed for creating the bottlenecks, training on images and evaluating an image. The expectation was that the InceptionV3 was the most accurate model, but the one with the most required performance classifying an image in the mobile application. Futhermore, the MobileNet 1.0 was expected to be the most fastest and accurate model, whereas the MobileNet 0.50 might be accurate, but slower than its successor. It turned out contrary to expectations that the MobileNet 0.50 is the most accurate one with lowest performance required (refer to section 5). As a last surprise, the app was running more smoothly on the Samsung S4 device than on the newer Motorola Moto X which contains a better processor.

References

He, K. (2016). Deep residual networks.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks.

LeCun, Y., Bottou, L., yoshua Bengio, and Haffner, P. (1998). Gradient-based learning applied to document recognition.

Android (2017). Surfacetexture. `https://developer.android.com/reference/android/graphics/SurfaceTexture.html` - accessed on 2018-01-26.

ImageNet (2010). Imagenet - summary and statistics. `http://www.image-net.org/about-stats` - accessed on 2018-01-26.

TensorFlow (2017). How to retrain inception's final layer for new categories. `https://www.tensorflow.org/tutorials/image_retraining` - accessed on 2018-01-22.

TensorFlow (n.d.). Mobilenet_v1. `https://github.com/tensorflow/models/blob/master/research/slim/nets/mobilenet_v1.md` - accessed on 2018-01-26.

Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions.

List of Figures

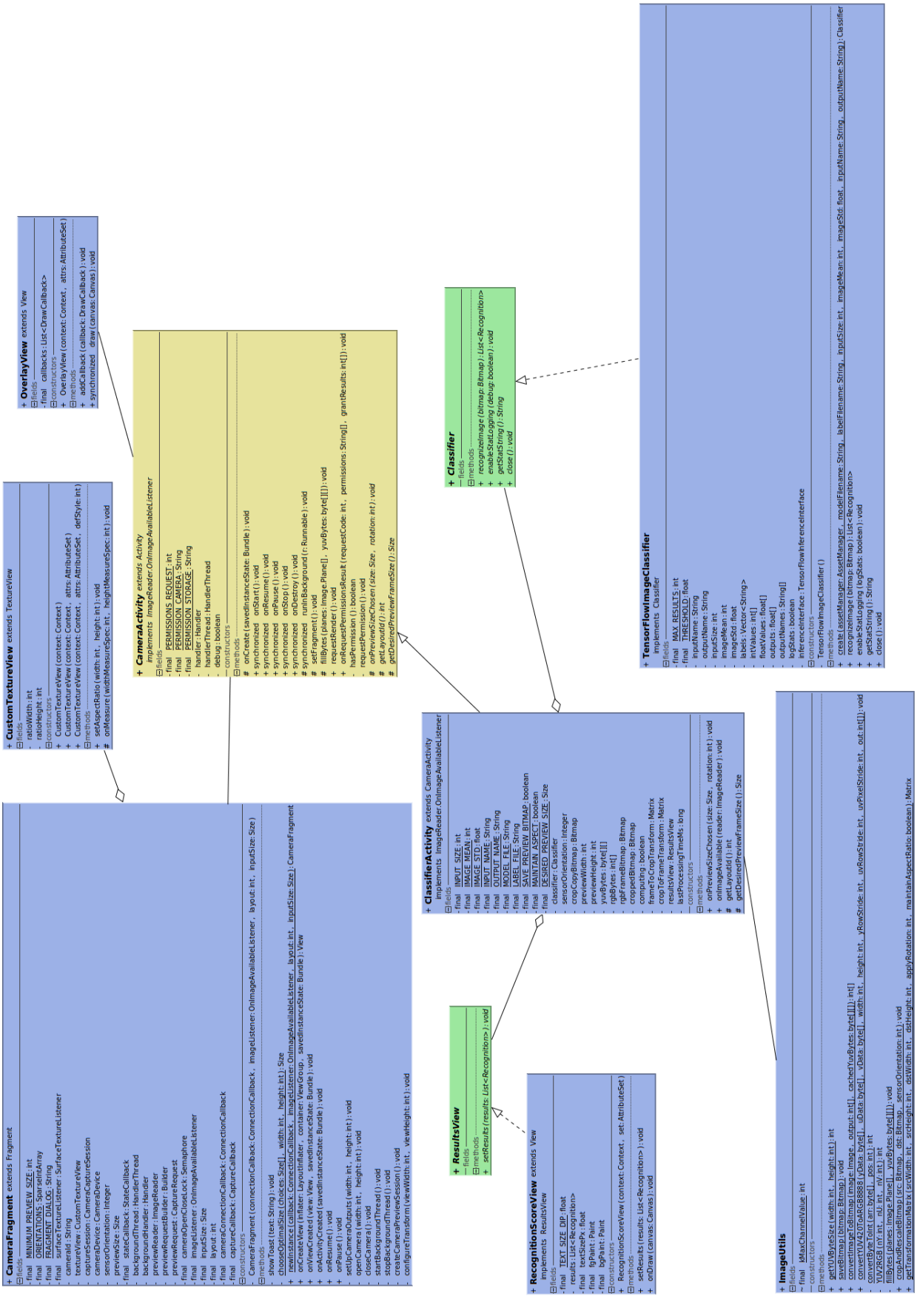
A. Appendix

A.1. Configure Bazel for Tensorflow

```
$ cd tensorflow  # cd to the top-level directory created
$ ./configure
Please specify the location of python. [Default is /usr/bin/python]: /usr/bin/python3
 .6
Found possible Python library paths:
  /usr/local/lib/python3.6/dist-packages
  /usr/lib/python3.6/dist-packages
Please input the desired Python library path to use.  Default is [/usr/lib/python3.6/
 dist-packages]
Using python library path: /usr/local/lib/python3.6/dist-packages
Do you wish to build TensorFlow with MKL support? [y/N]
No MKL support will be enabled for TensorFlow
Please specify optimization flags to use during compilation when bazel option "--
 config=opt" is specified [Default is -march=native]:
Do you wish to use jemalloc as the malloc implementation? [Y/n] n
No jemalloc as malloc support will be enabled for TensorFlow.
Do you wish to build TensorFlow with Google Cloud Platform support? [y/N] n
No Google Cloud Platform support will be enabled for TensorFlow
Do you wish to build TensorFlow with Hadoop File System support? [y/N] n
No Hadoop File System support will be enabled for TensorFlow
Do you wish to build TensorFlow with the XLA just-in-time compiler (experimental)? [y
 /N] n
No XLA support will be enabled for TensorFlow
Do you wish to build TensorFlow with Amazon S3 File System support? [Y/n]: n
No Amazon S3 File System support will be enabled for TensorFlow.
Do you wish to build TensorFlow with GDR support? [y/N]: n
No GDR support will be enabled for TensorFlow.
Do you wish to build TensorFlow with VERBS support? [y/N] n
No VERBS support will be enabled for TensorFlow
Do you wish to build TensorFlow with OpenCL support? [y/N] n
No OpenCL support will be enabled for TensorFlow
Do you wish to build TensorFlow with CUDA support? [y/N] Y
CUDA support will be enabled for TensorFlow
Please specify the Cuda SDK version you want to use, e.g. 7.0. [Leave empty to
 default to CUDA 8.0]: 8.0
Please specify the location where CUDA 8.0 toolkit is installed. Refer to README.md
 for more details. [Default is /usr/local/cuda]:
Please specify the cuDNN version you want to use. [Leave empty to default to cuDNN
 6.0]: 6
Please specify the location where cuDNN 6 library is installed. Refer to README.md
 for more details. [Default is /usr/local/cuda]:
Please specify a list of comma-separated Cuda compute capabilities you want to build
 with.
You can find the compute capability of your device at: https://developer.nvidia.com/
 cuda-gpus.
Please note that each additional compute capability significantly increases your
 build time and binary size.
[Default is: 5.2]: 5.2
Do you want to use clang as CUDA compiler? [y/N] n
nvcc will be used as CUDA compiler
Please specify which gcc should be used by nvcc as the host compiler. [Default is /
 usr/bin/gcc]:
Do you wish to build TensorFlow with MPI support? [y/N] n
MPI support will not be enabled for TensorFlow
Configuration finished
```

Listing 40: Configure bazel for Tensorflow

A.2. Mobile Application Architecture in the form of a Class Diagram



A.3. TensorBoard - optimal achieved accuracy of the different models

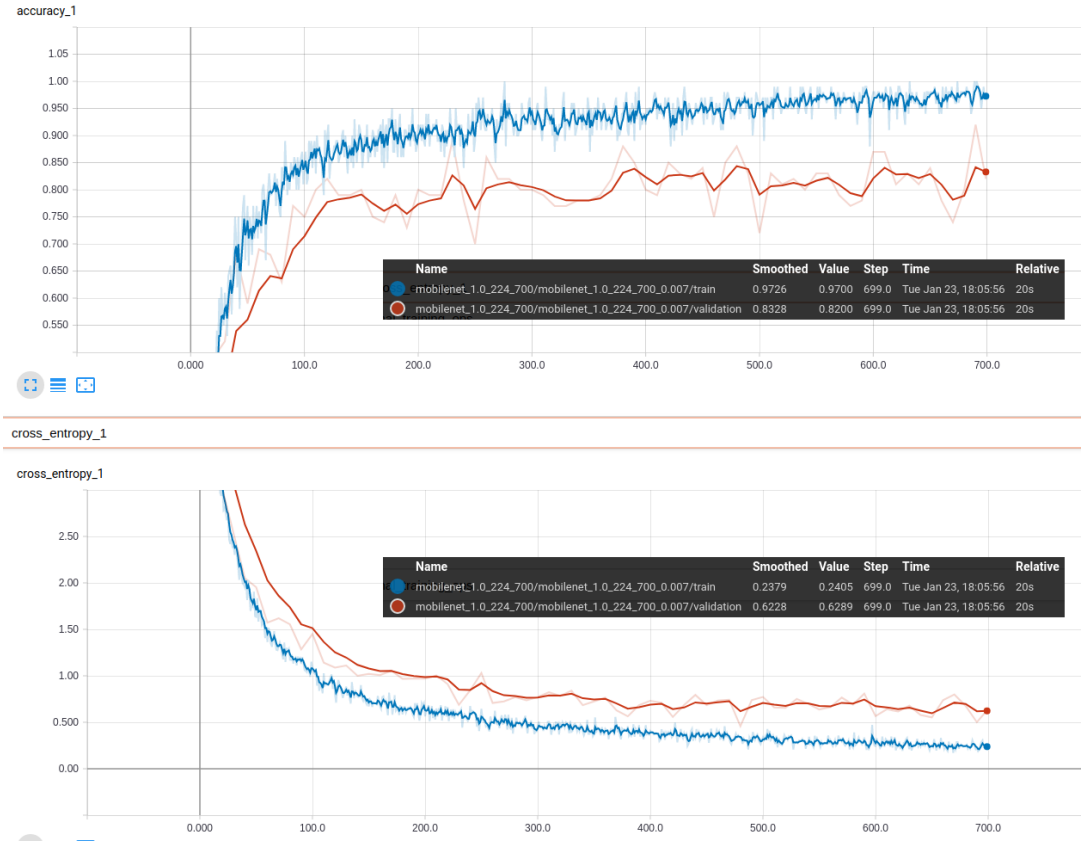


Figure 20: TensorBoard - optimal achieved accuracy of MobileNet_0.5

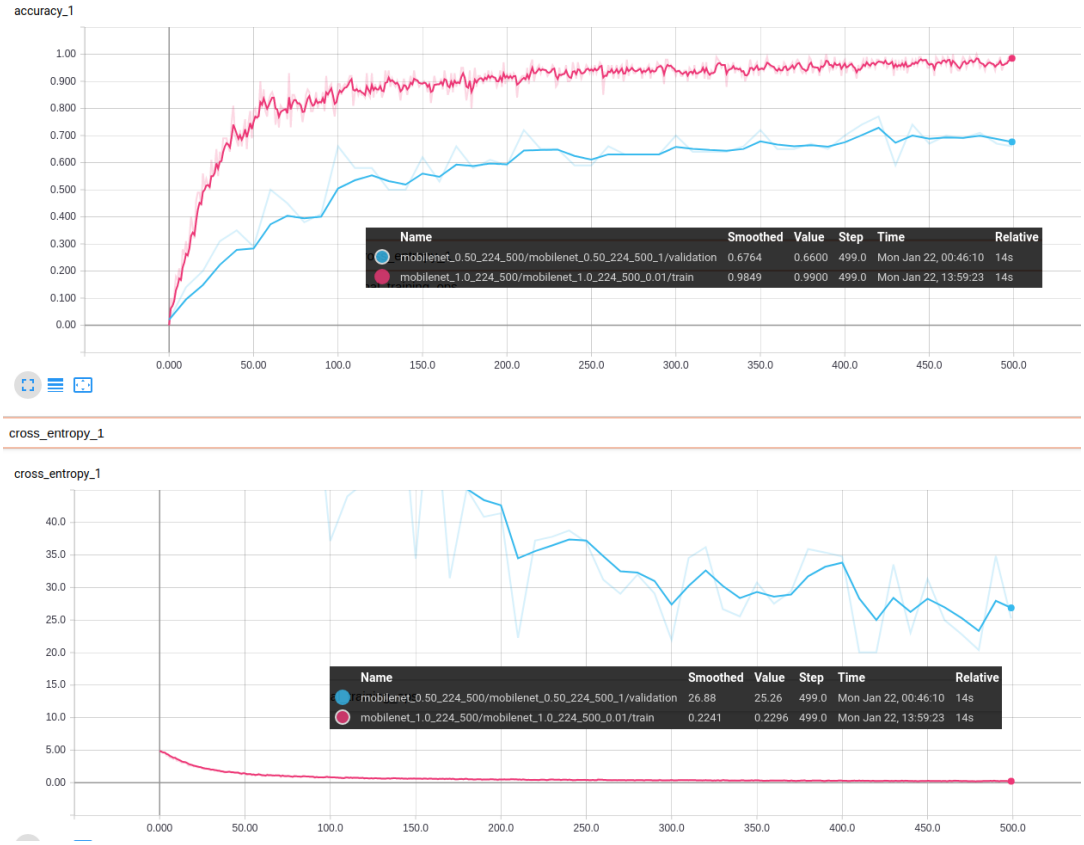


Figure 21: TensorBoard - optimal achieved accuracy of MobileNet_1.0

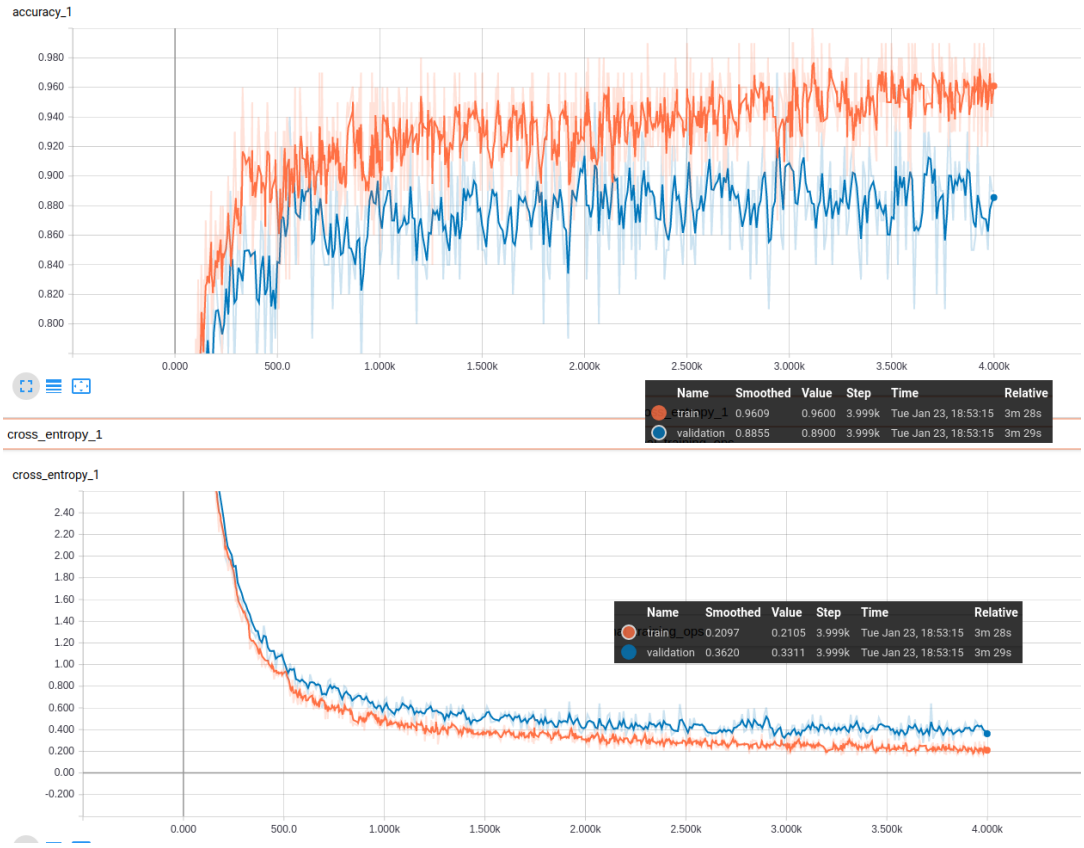


Figure 22: TensorBoard - optimal achieved accuracy of InceptionV3

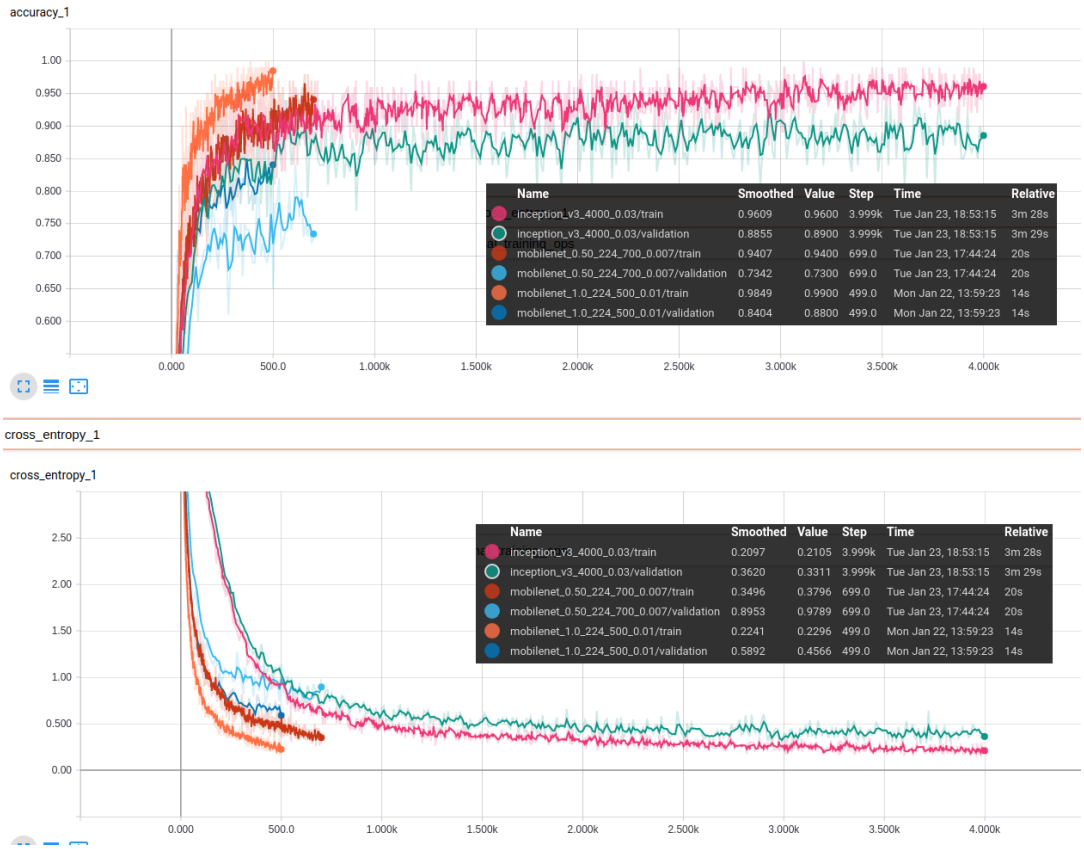


Figure 23: TensorBoard - optimal achieved accuracy of all models

opt attempt	accuracy	misclassified	time to classify (App) [ms]
retrained	0,566641033	5	0
opt1	0,566641033	5	239,6
opt2	0,487681977	8	252,5
opt3	0,566641033	5	241,9
opt4	0,483819713	8	246,8

Table 2: MobileNet_0.5

A.4. Evaluation of optimization attempts

opt attempt	accuracy	misclassified	time to classify (App) [ms]
retrained	0,504834563	7	0
opt1	0,504834563	7	316,2
opt2	0,458024049	8	310,2
opt3	0,504834563	7	323,4
opt4	0,458474453	8	320,6

Table 3: MobileNet_1.0

opt attempt	accuracy	misclassified	time to classify (App) [ms]
retrained	0,722748141	2	0
opt1	0,722748250	2	1092,2
opt2	0,698508064	3	1094,4
opt3	0,722748137	2	1139,9
opt4	0,700055786	3	1108,2

Table 4: InceptionV3