computergraphics tübingen

Image Deep Learning Effects (IDLE)

Jan-Oliver Nick, Jannik Hofmann, Florian Martin

March 15, 2019

Praktikum Computergrafik WiSe2019
Wilhelm Schickard Institute of Computer Science

Image Deep Learning Effects (IDLE)

Jan-Oliver Nick, Jannik Hofmann, Florian Martin

Figure 1: IDLE. Segmentation and effects.

Abstract

There are a lot of applications that focus on end-user friendly image processing and editing as well as filter usage, but none of them seem to utilize more of the image information than maybe face recognition (e.g. Snapchat). Thus, we created an application that applies filters based on additional information gained by machine learning, namely depth maps and person segmentation.

1 Introduction

As Snapchat and Instagram have shown, there is a huge audience for making photography editing - especially selfies and pictures of other people - available for the broad audience. The major part of this editing comes in the form of filters which can be applied to the whole image or parts of it.

The main difference between these filters and professional photo editing is the simplicity with which the former can be applied: Because they lack parameters (or these have been chosen beforehand by the developer), a user only has to choose the filter and does not have to bring any experience in photography editing.

For a user, this approach of photo editing simplification can be further ensued by developing these applications into smartphone apps, which is especially useful since most people already take their photos with their smartphones.

However, these apps focus on photo enhancement or face modification (e.g. Snapchat's infamous dog filter). With our project, we want to take a different approach: Create unreal but fun filters with the help of depth information and separation of persons with respect to the background. These enable us to change the background, apply filters only to either persons or the background, or change image properties like alpha/color channels based on the distance to the camera for example.

Our solution focuses on the blending of existing technologies rather than exploring new possibilities in one of the areas we touch. Thus, our focus was to combine robust, well-documented, tools and software in an app.

The tools that we use can be broken down into three parts: For application development, we focus on the Android platform, which has most users. This works best with Android Studio, which combines IDE, SDK and an Android emulator into one software.

For the neural networks that allowed us to estimate depth maps and segment the image, TensorFlow was our library of choice since it is maybe most common nowadays and also provides a more advanced Android port compared to PyTorch, which would have been our second choice. For the deployment of the trained neural network, we use TensorFlow Mobile.

Filters were created and applied using the C++ image manipulation library OpenCV since it comes with a lot of basic as well as advanced methods, is rather well documented and more efficient than hand-coded pixel iteration.

Since the training of neural networks - especially in the image processing area - can become computationally expensive rather quickly, we were granted access to a computing node of the WSI cluster. It is equipped with 512GB of RAM, four GeForce 1080 Ti graphics cards and two Intel Xeon E5-2650 v4 with 48 logical cores in total and a clock rate of 2.2GHz.

2 Related Work

In the areas of both semantic segmentation and depth map estimation a lot of work has been done already, although the latter is mostly focused on autonomous driving and using temporal dependencies between images (for example from videos) for depth estimation, which also affects the available data sets for depth learning with different goals.

A good overview over semantic segmentation is given by Liu et al. [LDY18]. Although, due to the simplicity of the architecture, we first used an U-Net [RFB15] approach for person segmentation as to get a first pipeline up and running. This model is a simple form of an encoder-decoder architecture that features skip connections between layers of equal size in the encoder and the decoder part. After the pipeline was functional, we discovered that there was an updated version of an architecture described in [LDY18]: The DeepLab V3+ model [CZP+18], which can also be counted towards the encoder-decoder architectures and which we settled on. It is thus further described in the next section.

For depth estimation, Block Matching Algorithms like SGBM [Hir08] are widely popular. They rely on the disparity between two images (stereo) to infer depth information. For the scope of our project, we are more interested in methods that infer depth information from monocular images, even though this is an ill-posed problem since the result with given information can be ambiguous, as the authors of [HWH18] argue.

Google camera [LLC19b] obtains precise depth information in its portrait modus by telling the user to move the smartphone up or down until an accurate enough depth map is created.

Out of usability aspects we are interested in approaches that only use one photo (monocular) for the creation of depth maps. Eventually, we settled for the work of Laina et al[LRB+16] since they have well-documented code available while achieving decent performance.

3 Methodology

3.1 Solution Approach

The first thing we did was write filters in C++, which is native to OpenCV as to test out possibilities and get a grasp on which effects can be applied well and which ones are rather unrealistic.

In parallel, different solutions for person segmentation were evaluated. At first, a U-Net architecture seemed promising, which we trained on the Supervise.ly [LLC19a] person segmentation data set, but the results were not nearly good enough as to be of any help in the app. In the end, we settled on the approach of [CZP+18], which is essentially a U-Net architecture with clever improvements. It is described in detail within subsection 3.2.

For depth map estimation, we originally tried to use Block Matching and Semi-Global Block Matching (SGBM) [Hir08], but the results were really unsatisfactory, which is also due to the fact that it is not user-friendly to take two pictures that keep the same rotation and position except for movement on one axis when small rotations or movements in another direction can already have huge negative effects on the resulting depth map. Since this would put additional requirements on the user, we decided to take a neural network approach for depth map estimation.

We use TensorFlow Mobile, which is on the verge of being replaced by TensorFlow Lite. But as of the start of development, some features required by the models are not supported by TensorFlow Lite yet.

3.2 Segmentation

For semantic segmentation, which is a more general term for what we want to achieve, there are already architectures that come with very good performance.

The model we used, which is already included in the TensorFlow repository along with weights pretrained on PASCAL VOC 2012 and Cityscapes datasets, is Google's DeepLab V3+ model [CZP+18]. It achieves an astounding performance of 89% on PASCAL VOC 2012 which also features pictures of humans and thus exactly the kinds of images that we expect our application to be used on. This also means that the model is trained not only on detecting humans but also on other classes like airplanes and plants, which is unnecessary but not substantially damaging to our project.

For this model, the authors combine an encoder-decoder architecture with atrous spatial pyramid pooling (ASPP) and depthwise separable convolution on an Xception model backbone.

In an encoder-decoder architecture for images, the input's spacial dimensions are downscaled (for example with strided convolution) while the number of feature maps goes up in the encoder and vice versa in the decoder. This allows for the extraction of key features in a reduced spatial dimension, which is mostly done for reasons of computational complexity.

ASPP employs a number of filters with different dilation rates as well as a pooling operation on the same input (feature map) and concatenates their results. This allows for feature mappings on different resolutions (meaning differently sized fields of view) and has proven to be rather effective, given this paper's reported mean intersection over union (IoU) rates on segmentation tasks.

Depthwise separable convolution applies one filter to each channel (feature map) independently of the others (depthwise convolution) and follows this up with pointwise (1x1) convolution over all the feature maps of the depthwise convolution. This architecture requires way less parameters: As opposed to

$$(in\_maps \times out\_maps \times conv\_size),$$

this convolution only requires

$$(in\_maps \times conv\_size + in\_maps \times out\_maps)$$

parameters and is thus less expensive when it comes to computational costs. With an exemplary 8 to 16 channel convolution with a 3x3 window, this is the difference between 200 and 1152 parameters. Depthwise separable convolution is replaces every classical convolution operation in the DeepLab V3+ model. This convolution architecture assumes that it is possible to decouple spatial and channel information, but the results support this hypothesis. In practice, a batch normalization layer is added in between every depthwise and pointwise convolution.

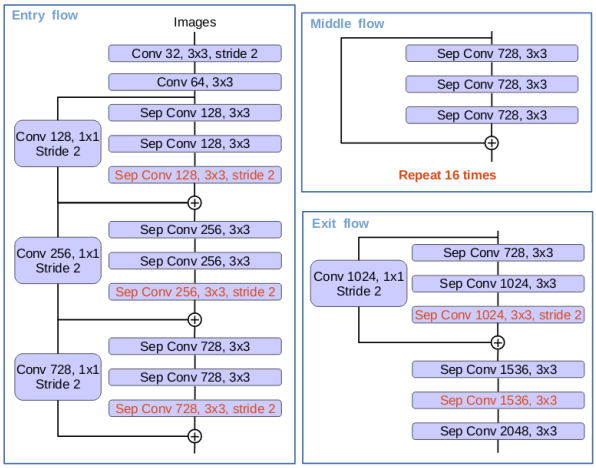

Figure 2: DeepLab V3+ Xception architecture. Missing from the pictures are batch normalization and ReLU layers after every 3x3 convolution. The Image is taken from [CZP+18].

The final architecture consists of the feature extractor part of the encoder that is described in Figure 2 and the ASPP part that can be seen in Figure 3. The decoder utilizes a skip connection, bilinear upsampling and more depthwise separable convolution before giving one feature map for each class indicating pixel-wise probabilities of this pixel belonging to said class. Assigning the class with the highest value to each pixel creates the mask for segmentation.

The model uses an input resolution of 512×512 pixels, so we downscale the image as to match this size with the longer side and pad the other dimension to this quadratic shape.

An example of the segmentation can be found in Figure 4. There, our first U-Net model is compared to the final model that is described in this section.

So far, the model is trained on multiple classes and thus our application also recognizes and segments multiple classes.

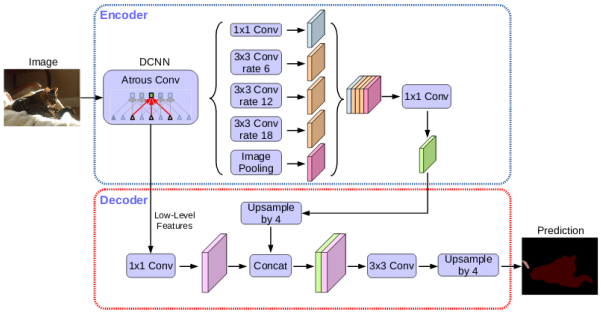

Figure 3: Complete segmentation architecture. The Xception backbone from Figure 2 is labeled DCNN here. The 1x1 convolution from the DCNN in the decoder is followed by batch normalization, ReLU activation and 0.1 dropout while the upsampling is only followed by batch normalization and ReLU. The Image is taken from [CZP+18].

3.3 Depth Map Estimation

Since the depth map created by Google camera is really accurate, one of the first ideas was to call this application with an intent, but the required lens blur feature is only available when Google camera is used directly. Since this app is only available for a limited number of models, the code for this feature is not available publicly and rebuilding it could be seen as a project on its own, we had to use another approach.

As mentioned in Subsection 3.1, Block Matching did not perform well. A comparison with the result from the neural network proposed by [LRB+16], which we eventually used, can be seen in Figure 5 and shows substantial differences, even though the predicted depth map does not seem perfect either. For our application and the effects we base on depth estimation, this depth map is good enough, though.

The architecture used by Laina et al. is shown in Figure 6. It is a rather basic ResNet-50 architecture that employs upsampling as decoder of sorts. The authors argue that for depth estimation, most important cues are to be found in a global context, so a large receptive field is crucial. In comparison with VGG-16 and AlexNet, ResNet-50 wins in that respect, having a receptive field of 483×483 pixels.

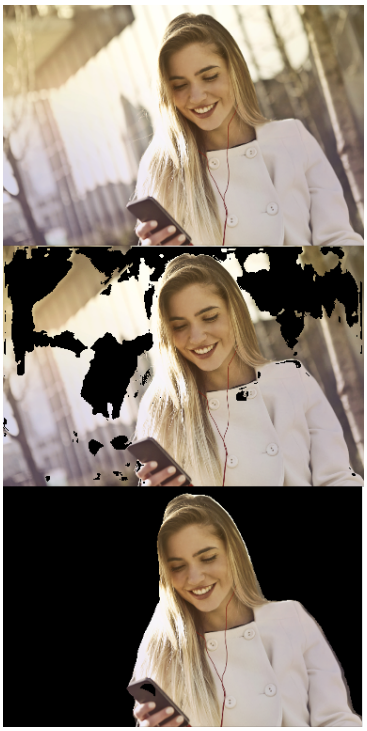

Figure 4: Segmentation results. The upper image shows the input image taken from the Supervise.ly data set while the one below is the mask from the initial U-Net architecture applied to this input. The bottom image is the result of the DeepLab V3+ model.

The encoder part of the network compresses the input into a spatial size of 10×8 and 2048 feature maps. The decoder utilizes four up-convolution layer to achieve a final output size of 160×128 with 64 filter maps. According to the authors, a fifth block did not increase performance.

What is probably most remarkable about the author's approach is the loss function they propose - the reverse Huber loss:

$$\mathcal{B}(x) = \begin{cases} |x| & |x| \leq c \\ \frac{x^2+c^2}{2c} & |x| > c \end{cases}$$

The authors use $c = \frac{1}{5}\max_i(|\tilde{y}_i - y_i|)$ where i indexes all pixels in the current batch. This loss function is a mixture between $\mathcal{L}_1$ and $\mathcal{L}_2$. The authors further argue that, since they experienced heavy-tailed distributions of depth values, this loss function is more appropriate.

Figure 5: Depth map comparison. The upper picture shows the input while the central one shows the depth estimate achieved with Block Matching. Bottommost, the prediction of the neural network can be seen.

3.4 Android Development

For the android app, we combine the filters created with OpenCV and the information inferred from the neural networks. OpenCV is initially written in C++, however there are compiled versions that can be included in the Android Java Environment.

The different effects that can be applied to any image with the application use a mixture of the original image, a foreground mask and the depth map to apply traditional image processing techniques in order to apply various filters to any image chosen by the user. The foreground mask is generated by interpreting the result of the semantic segmentation. If more than one percent of the image pixels belong to a person, only these pixels are treated as foreground.

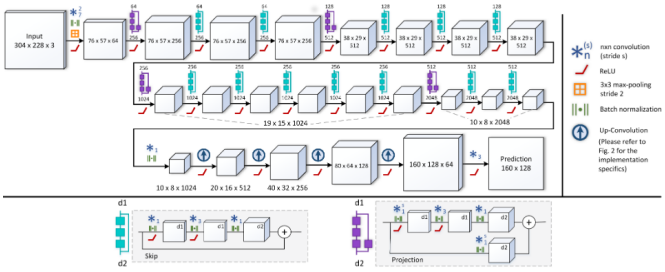

Figure 6: Depth prediction architecture. This architecture builds upon ResNet-50, but replaces the fully connected layers at the end with up-sampling blocks. This image is taken from [LRB+16].

However, if less than one percent of the original image shows a person, then all pixels, that are not part of the background, are treated as foreground elements.

One of these effects is the inpainting filter. After the foreground has been recognized, this effect attempts to remove the foreground subject from the image by filling in the relevant area with information from the surrounding parts of the original image. It uses a fast technique that was developed by Alexandru Telea in 2004 and is "based on the fast marching method for level set applications" [Tel04]. As long as the subject takes up only a small area of the image, the inpainting works well enough to produce a result, that is not immediately recognized as having been edited. If the subject takes up a significant area, for example in a portrait photograph, the inpainting image shows easily visible signs of image processing 7.

This effect is not only used as a filter for the end user but is also utilized in other effects. For example, the motion blur effect uses the inpainted image as a background before blurring it. This way, the foreground subject does not get blurred into the area around it while the background is smeared horizontally using gaussian blur, so the background appears to be blurred independently of the foreground content, which facilitates a more realistic motion blur effect. The inpainting effect is used in a similar manner for the hologram effect.

Figure 7: Inpainting effect. The foreground subject takes up a significant part of the image. Therefore, the inpainting effect has to cover up a large area of the image.

Another filter that can be applied to any image is the fog effect. It uses the normalized depth estimation to dynamically overlay a layer of very bright gray. The further away a pixel is from the camera, the more opaque this layer becomes. If a pixel is close to the camera, it mostly keeps its original value.

The hologram effect uses the inpainted background and enhances it to be slightly darker with more contrast. Then, the foreground object is masked using the semantic segmentation and the OpenCV colormap winter is applied to it, which creates an image that glows in a blue tone. And finally, the colored foreground is blurred vertically, set more transparent and added to the image by adding the pixel values. This creates a glowing smeared effect, to further facilitate the impression of a hologram being projected.

The exact values that define how the images are mixed together were empirically defined during development of the application.

To understand the way that trained TensorFlow networks are ported to android, it is important to first understand how computations with TensorFlow work: At first, a computation graph is built and filled with placeholders for the inputs, which is only to be fed after a complete definition from input to output. The graph determines the shapes that flow from layer to layer and the actual computations that are performed at these nodes.

So after training, the model with its weights has to be saved into checkpoint files containing the graph definition and the weights belonging to this graph. From these checkpoint files, a frozen graph definition can be created, which is essentially a binary file combining the structure and the selected weights of the graph.

These frozen graphs, which have a size of around 150MB for the segmentation network and around 250MB for the depth estimation network, can be used on its own for inference. After installing TensorFlow Mobile, these files can be loaded into TensorFlowInferenceInterfaces, which can in turn be used to get serialized java arrays for further processing.

The user interface, from a technical perspective, consists of two so-called activities, one gallery and one screen for effect selection. The former is where the user can see all his chosen images and add new ones. The gallery is a so called recyclerView, which describes a list of objects that is bound to the view element, so there is a seperation between the model, which holds the data and the view. This pattern of separation of data and representation is called model-view-viewmodel (MVVM).

Swapping between activities or requesting a picture (either by taking one or by choosing one from existing photos on the smartphone) is done by using so-called intents. The app can use these as a tool to request high-level system resources.

The Android framework contains an UI thread which acts as the main process for the whole application. It handles all the events that are triggered by user input and updates the user interface accordingly. For this reason, it needs to be responsive instead of handling expensive calculations. While activities that take several seconds are computed on this thread, the application is frozen. If it stays in this state for too long, it may be shut down by the system due to being unresponsive.

To combat this issue, the Android architecture requires complicated tasks to be executed in separate threads. This way, the main process can start such a thread after receiving user input and then continue waiting for new actions, while the activity is running in the background.

When using this strategy, the communication between main thread and separately running activities needs to be handled with care: These threads are not able to access all variables and have to acquire the permission to write (so-called "locks") into shared variables. This is necessary as to guarantee thread safety, which in turn is required for the application to run in a stable and predictable manner.

The Android framework encourages this concept by providing a straightforward way to run parts of the code from different threads in the main process, where access to the main thread's variables is granted.

Through the usage of threading, the starting process could be sped up by eight seconds. Another way of boosting the loading time of the app was to save not only the original image but additionally a downscaled version that we use for display in the previews.

Due to the size of the neural network files (around 400MB), the final `.apk` of our application has a size of roughly 550 MB.

3.5 User Experience

After the application is started, the user has the choice of either importing an existing image or taking a new one with a photography application of his choice that is started with an Android intent.

Once a picture is imported through one of these methods, it is shown and saved in the application's internal gallery, which is also shown on future launches. The user is shown a notification ("toast"), informing him that the analysis of the image has started, which allows him to do other things in the application or on his phone. When this analysis process has finished, a pop-up enables the user to jump to the effect view where he can see the original image along with the implemented effects applied to the image.

In this effect view, the user can save, delete or share the image along with the chosen effect applied to it.

All this functionality is also explained through an information button in the application's internal gallery or available through long presses on elements of interest. For example, tapping on the segmentation of an image shows the legend of the colors.

Screenshots of the application can be seen in Figure 8.

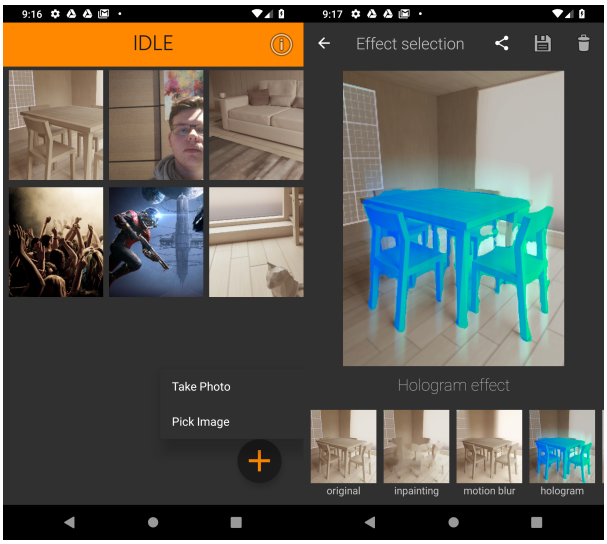

Figure 8: Screenshots of our application. The left image shows the gallery while the right one shows the effect selection screen.

4 Evaluation

In this section, we address two major issues of the application: Memory usage and the time required for the different processing steps.

The largest memory consumption that can be produced is around 3.5GB when a user views a picture while the segmentation is calculated for a different picture. If a smartphone does not have the required amount of free RAM, the application crashes.

For timing, we measure the time it takes to execute our filters on different terminal devices: One emulator on a laptop utilizing an Intel Core i7-6700HQ with 2.6GHz, one OnePlus 3 with a Snapdragon 820 processor and a OnePlus 6 with a Snapdragon 845 CPU.

We compare these devices on the preparation and inference tasks for both neural networks and the application of all effects. The image we take for this example has a resolution of $2,448 \times 3,264$ pixels and 25.76 of the image are a person - it is a selfie. The results can be seen in Table 1.

When taking a look at the timing, it can be seen that the preprocessing (down- and upscaling as well as padding) takes very little time compared to the other tasks.

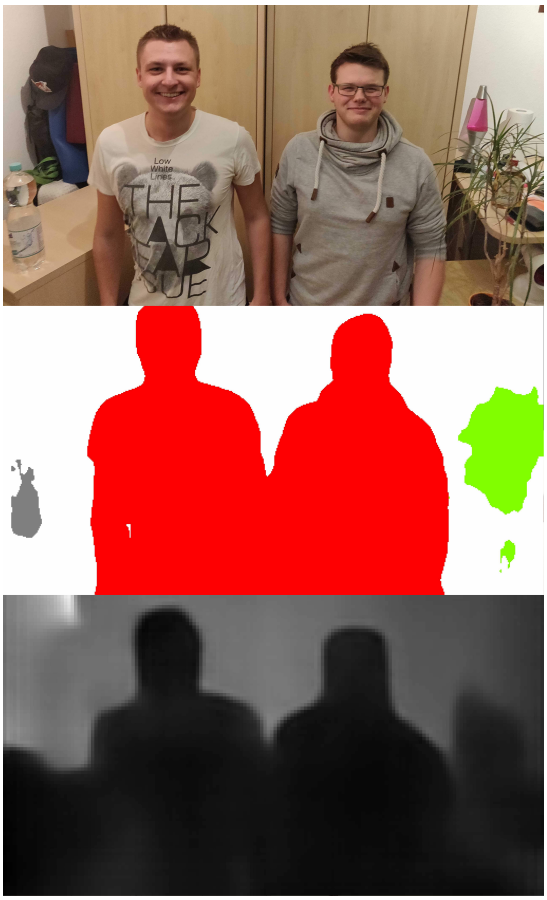

Figure 9: Neural network results. Photo taken by the user and segmentation results & depth map estimation after image analysis. The segmentation recognizes bottle, plant and human.

Figure 10: Overview over all our effects. From top to bottom: Inpaint, motion blur, fog and hologram effects.

Secondly, the time necessary for inference from the networks is very similar on the rather new smartphone compared to the laptop, but for effect application, the two smartphones have rather similar timings in relation to the emulator. A possible reason for this good performance of the OnePlus 6 in comparison to the computer may be a design specialty of the Snapdragon 845, which allegedly utilizes "innovative architectures for artificial intelligence" and natively supports TensorFlow[1].

Also, the overall time required for the application in deployment is probably too high for the average end-user. Even on a top smartphone released in 2018, trying out all filters takes around two minutes in computation time alone.

When looking at the results seen in the exemplary pictures of Figure 10, there are some minor problems that occur in the final filters: In some cases, the person segmentation is inaccurate or of lower resolution (due to downscaling for the neural networks), which is especially problematic for filters that only change the background, e.g. motion blur.

The depth map also produces rather bad results in some cases, which can be seen in Figure 5 for examples. However, these are so far only relevant for the fog effect.

But while the computation time is a rather big issue, these inaccuracies of the network inferences are of little importance.

[1] https://www.qualcomm.com/news/releases/2017/12/06/qualcomm-snapdragon-845-mobile-platform-introduces-new-innovative

	OP 3	OP 6	Emulator
Segnet preparation	0.3s	**0.2s**	0.3s
Segnet inference	32.9s	13s	**12.5s**
Depthnet preparation	0.4s	**0.2s**	**0.2s**
Depthnet inference	5.9s	2.0s	**1.9s**
Inpaint effect	2.8s	**1.0s**	1.1s
Motionblur effect	61.3s	53.1s	**12.5s**
Hologram effect	95.5s	55.7s	**18.0s**
Fog effect	2.1s	**0.8s**	1.4s

Table 1: Timing. This table compares the timing for different parts of our application on different mobile devices. Lower times indicate better performance.

5 Discussion

While this project can generally be seen as an exemplary work for the fusion of different areas of computer vision, it also shows the vastness of the different topics grazed. For each of these areas, an efficient android port could be a project on its own. So our scope for this project was too wide. As can be seen in Section 4, in our case performance was the part which could clearly be optimized further.

To address the issue of performance, some ideas are given in Section 6.

Nonetheless, the result of this project is a smartphone application than can rather reliably segment people, infer depth maps and use this additional information for filters that are rare to be found in other applications.

6 Future Work

As the evaluation of timing has shown, our application is very large in size, requires more RAM than the average application in order to work and is quite slow so far, potentially even too slow for the average user.

One way of speeding up the inference procedure is by quantizing the TensorFlow models. This means that the 32bit floating point precision is converted to 8bit integer (after training - for inference only) as to reduce the memory consumption and possibly also the inference time since computation with integers should be faster and the size of the model that has to be loaded is going down to roughly one quarter of the original size.

However, it has to be said that since TensorFlow does not have quantization implemented for all layers, it is also possible that the inference time does not go down since the integers have to be converted to floats for some layers at least.

Once TensorFlow Lite has all the functionality of TensorFlow Mobile, one could consider to use this backend instead in order to potentially reduce the usage of memory and storage.

Although the model used for person segmentation achieves good results on its own, it can be trained further with the Supervise.ly data set which is introduced in 3.1. Also, the model is trained to recognize multiple classes so far, but according to the no free lunch theorem, this either results in unnecessary model complexity or reduces the performance on our task.

Alternatively, a future version of our application could make more use of the recognition of multiple distinct classes. Various objects in the image could be treated differently by what kind of object they are. For example, vehicles, animals, furniture and humans could have individual image processing applied to them in the same picture. Currently, the application either treats only humans as foreground while ignoring any other objects in the image, or it selects anything that is not part of the background, if less than one percent of the image shows humans.

Also, the information inferred through the segmentation and depth map provides potential for various other advanced effects that can be included in future versions of the application.

So far, our network architectures require different input sizes for inference, which in turn requires us to rescale and pad the image an additional time. Although these operations do not require much time, the additional image requires additional memory, which should be massively reduced for application deployment with a broader audience.

Currently, there is no standardized way of utilizing multiple cameras or other dedicated systems for capturing three-dimensional images on an Android phone. On March 13, 2019 the first beta of Android Q, the upcoming major version of Android was introduced. It is going to natively support an XMP metadata format that stores depth information within a JPEG image [Bur19].

With Android Q, phones that have multiple cameras and other dedicated systems for capturing three-dimensional pictures could capture depth information natively and then save created depth maps into the file.

This way, the user could import pictures that already contain depth data and the neural network for estimating this information would become redundant. This would not only speed up the analysis process but also results in much more precise effects as the depth information would not have to be estimated based on the two-dimensional image content but would be directly captured on creation of the image.

The inpainting effect currently only utilizes a relatively straightforward technique based on the fast marching method [Tel04]. This technique works well for the removal of small objects from an image and is usually used to remove blemishes from scanned photographs. However, once the subject takes up a larger area of the image, it is easily detectable that an area of the image has been removed. In a future version of the application, another neural network for advanced image inpainting could be utilized, so that the foreground area is filled in by intelligently utilizing the information around that area [YLY+18]. This would also retain the textures and minimize the discernibility of the fact that image processing has taken place.

Post processing like conditional random fields or edge smoothing could be applied to the results, especially the segmentation. This should make for more accurate results, which is especially important for filters like motion blur, which leaves the persons unchanged but alters the background. Although, also the hologram effect could benefit from this post processing.

References

[Bur19] Dave Burke. *Introducing Android Q Beta*, 2019 (last accessed March 13, 2019). `https://android-developers.googleblog.com/2019/03/introducing-android-q-beta.html`.

[CZP+18] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *ECCV*, 2018.

[Hir08] Heiko Hirschmuller. Stereo processing by semiglobal matching and mutual information. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(2):328–341, February 2008.

[HWH18] Lei He, Guanghui Wang, and Zhanyi Hu. Learning depth from single images with deep neural network embedding focal length. *CoRR*, abs/1803.10039, 2018.

[LDY18] Xiaolong Liu, Zhidong Deng, and Yuhan Yang. Recent progress in semantic image segmentation. *CoRR*, abs/1809.10198, 2018.

[LLC19a] Deep Systems LLC. *Supervisely*, 2019 (last accessed March 14, 2019). `https://supervise.ly/`.

[LLC19b] Google LLC. *Google Camera*, 2018 (last accessed March 14, 2019). `https://play.google.com/store/apps/details?id=com.google.android.GoogleCamera`.

[LRB+16] Iro Laina, Christian Rupprecht, Vasileios Belagiannis, Federico Tombari, and Nassir Navab. Deeper depth prediction with fully convolutional residual networks. In *3D Vision (3DV), 2016 Fourth International Conference on*, pages 239–248. IEEE, 2016.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[Tel04] Alexandru Telea. An image inpainting technique based on the fast marching method. *Journal of graphics tools*, 9(1):23–34, 2004.

[YLY+18] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas S Huang. Generative image inpainting with contextual attention. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5505–5514, 2018.