Computer Vision with Machine Learning on Smartphones for Beauty Applications

submitted by

Valentin Miu

for the degree of Doctor of Engineering Digital Media

of

Bournemouth University

Centre for Digital Entertainment

November 2021

Abstract

Over the past decade, computer vision has shifted strongly towards deep learning techniques with neural networks, given their relative ease of application to custom tasks, as well as their greatly improved results compared to traditional computer vision techniques. Since the execution of deep learning models is often resource-heavy, this leads to issues when using them on smartphones, as these are generally constrained by their computing power and battery capacity-limited energy consumption. While it is sometimes possible to conduct such resource-heavy tasks on a powerful remote server receiving the smartphone user's input, this is not possible for real-time augmented reality applications, due to latency constraints. Since smartphones are by far the most common consumer-oriented augmented reality platforms, this makes on-device neural network execution a highly active area of research, as evidenced by Google's TensorFlow Lite platform and Apple's CoreML API and Neural Engine-accelerated iOS devices. The overarching goal of the projects carried out in this thesis is to adapt existing desktop computer-oriented computer vision techniques to smartphones, by lowering the computational requirements, or by developing alternative methods. In concordance with the requirements of the placement company, this research contributed to the creation of various beauty-related smartphone and web apps using Unity, as well as TensorFlow Lite and TensorFlowJS for the machine learning components. Beauty is a highly valued market, which has seen increasing adoption of augmented reality technologies to drive user-customized product sales. The projects presented include a novel 6DoF machine learning system for smartphone object tracking, used in a hair care app, an improved wrinkle and facial blemish detection algorithm and implementation in Unity, as well as research on neural architecture search for facial feature segmentation, and makeup style transfer with generative adversarial networks.

Contents

List of Figures

List of Tables

Declaration

This report has been created by myself and has not been submitted in any previous application for any degree. The work in this report has been undertaken by myself except where otherwise stated.

Chapter 1

Introduction

Augmented reality (AR) is the process of introducing virtual, computer-generated elements to a real environment. This generally has the goal of enhancing the environment with additional sensory information, particularly visual, and often interactive. It has seen applications both in industry, for training and military applications, as well as in entertainment, through smartphone AR games.

In order to adapt the virtual elements to the real environment in a believable, seamless way, the computer must be able to process the environment based on sensor inputs. This is known as *scene understanding*, which is a subset of computer vision (when the sensory input is purely visual). Computer vision itself refers to the extraction of meaningful information from visual inputs, like images, videos or video feeds, by a computing device. This could be a desktop computer, server, or smartphone, for example.

Over the past decade, computer vision techniques have increasingly relied on machine learning with deep convolutional neural networks. Machine learning refers to algorithms which can be iteratively improved through the use of data and experience (Mitchell 1997). This avoids the need to explicitly program the computer for a certain task, which provides an alternate solution where traditional, fixed algorithm-driven approaches would be too complex to develop, as is often the case in computer vision.

In computing, neural networks are machine learning systems based on biological neural networks in animals (Hopfield 1982). *Convolutional* neural networks are specifically based on the biological visual cortex, replicating it as a sequence of convolution operations, with the capability to automatically learn visual features (Valueva et al. 2020). They are currently the most common technique for object detection, human and vehicle pose detection, semantic segmentation, and facial keypoint detection, all of which can aid in scene understanding for augmented reality tasks.

In this thesis, focus is put on computer vision and augmented reality techniques, generally using machine learning, aimed specifically at smartphones running either Android or iOS. These devices have significantly more stringent computational and power consumption limits than normal computers and servers, which leads to significant challenges in designing neural networks and algorithms for the required tasks. The computer vision techniques in this thesis specifically target applications in the beauty sector, as required by the placement company under which the research was carried out. These apps using the techniques needed to be implemented in the Unity3D game engine (Unity Technologies 2022b), due to it being used in the existing codebase of the company, due to the complex graphical requirements of some of the applications, and due to its good support for mobile platforms like Android and iOS (Unity Technologies 2022a).

1.1 Motivation of research

While the cutting edge of mixed reality *hardware* is formed of augmented reality (AR) headsets like the Hololens (Garon et al. 2017), Magic Leap One (Magic Leap 2021) and Nreal Light (Nreal 2021), smartphones currently number about 3.3 billion users, according to Statista Research Department (2018), making them the largest augmented reality platform by far. Especially in the consumer market, AR headsets and glasses have a relatively small presence. As such, smartphone platforms are the domain in which novel research is most likely to have the greatest impact, which is why they are the focus of the AR and computer vision (CV) research in this thesis. Furthermore, both smartphones and smartglasses are relatively low-powered devices, with some smartglasses even using the smartphone as the compute device (Nreal Light (Nreal 2021) and likely the upcoming Apple glasses (Lynch and Peckham 2022)). As a result, research aimed at smartphone-optimized computer vision and machine learning techniques should also be largely applicable to smartglasses as well.

The focus of this thesis on the beauty sector was dictated by the requirements of the placement company under which the research was undertaken. As such, a focus purely on advancing the state of the art was not possible, as the company aimed to use the research to bring its products to market, which limited research time in favor of implementation. Despite this, to the extent possible, a focus on novel research was kept when choosing the methods through which the required augmented reality and computer vision problems were tackled, in line with the academic requirements of the thesis.

Due to its current and potential profitability, the beauty industry as a point of

focus also makes sense from a research perspective. This is because much of the existing research on application-oriented augmented reality and computer vision with machine learning (CVML) is driven by market presence and the ability to turn a profit, in order to fund continuing research efforts. Beauty is a very lucrative industry, valued at over half a trillion USD worldwide (Biron 2019). And due to the traditional reliance on brick-and-mortar stores, there is great opportunity for technological advancement in the sector. A greater demand for mobile and browser experiences was also generated by the pandemic of 2020, when the traditional in-store experiences became impossible. In parallel, both augmented reality and the beauty-oriented apps on smartphones have been in part driven by the rise of applications like Snapchat and Instagram, which offer a variety of AR filters, many geared towards beautification (Haines 2021). This illustrates the consumer interest for beauty-related smartphone AR experiences, which can in turn lead to investor funding for research efforts.

Another motivation was that many of the project outputs of this thesis can be used for general applications beyond beauty. For example, the integration of the TensorFlow Lite machine learning library into the Unity game engine (Section 2.4) can be used for any mobile games or graphically intensive applications that require computer vision or augmented reality. The six degree of freedom object pose detection technique in Chapter 3 is usable for any object, so it can be used for anchoring virtual objects into a real scene, or tracking various objects of interest. The makeup style transfer method in Chapter 4 is an experiment in bringing generative adversarial networks to smartphones, which have already seen very successful use for various filters in the Snapchat app (Harbison 2019; Snap Inc. 2021). The wrinkle detection project in Chapter 5 may have applications in smartphone-based face tracking, for more human-like expressions. Finally, the mobile-oriented neural architecture search for semantic segmentation research in Chapter 6 can be used for other augmented reality tasks like object occlusion and background substitution or removal.

1.1.1 Technological gaps in smartphone AR

Aside from the value of the targeted industry sector, the research in this thesis is also motivated by the lack of AR and CV/ML methods for particular tasks on smartphone platforms. While there has been significant work to bring machine learning to mobile, with smartphone-oriented frameworks such as Caffe2 (Facebook 2021c) and TensorFlow Lite (Lee et al. 2019), as well as lightweight smartphone-friendly neural networks such as SqueezeNet (Iandola et al. 2016) and MobileNet (Howard et al. 2017), ensuring both precision and speed on these platforms remains a challenge for many tasks.

Initially, one difficulty in ensuring low neural network latency on mobile was the

lack of hardware acceleration on mobile platforms. This refers to the use of the graphical processing units (GPUs) or other specialized chips (digital signal processors or DSPs) of smartphones, which generally leads to much faster inference than on the CPU. This ceased to be an issue several months after the start of the work in this thesis, when hardware acceleration was added to TensorFlow Lite (the machine learning framework for smartphones used in this thesis), using OpenGL on Android and Metal on iOS. While some older Android models had insufficient OpenGL support for TensorFlow Lite hardware acceleration, the later addition of the OpenCL delegate increased the number of supported Android devices, and provided a significant speedup over the OpenGL delegate.

Despite this, many neural network operations available on the CPU were not available or more limited when using hardware acceleration. While it was often possible to work within these limitations, some operations required either the addition of functionality to TensorFlow Lite GPU, or relegating these operations to Unity shaders or optimized CPU operation using OpenCV, a C++ library for computer vision tasks (Gary 2008).

The greatest issue encountered was the lack of mobile-friendly counterparts to computer-oriented machine learning models for specific tasks, such as six degree of freedom object pose estimation and fast or real-time generative adversarial networks for images and video. While mobile-oriented feature extractors exist, like the aforementioned MobileNet and SqueezeNet, these are generally targeted mainly at classification and segmentation. While these do offer a starting point for extension to other tasks, the variety of desktop-oriented models in existing research is still far greater than that of mobile-oriented models. In our research, this made it necessary to construct new neural network architectures for six degree of freedom object tracking (Chapter 3) and generative adversarial networks (GANs) for style transfer (Chapter 4). Other times, mobile network variants exist but are insufficiently precise or fast, which led to the efforts in Chapter 6, for neural architecture search for improved semantic segmentation on mobile and web platforms.

As a result, the overarching goal of this thesis is to create smartphone-friendly machine learning models, or optimize existing models for smartphone use, for a variety of augmented reality tasks required by the placement company. All of the applications targeted the beauty sector, as this was the target of the placement company under which the research took place.

1.2 Aims and objectives

The various sub-projects of this thesis follow two general aims: the creation of a framework for efficient machine learning inference on smartphones in Unity applications, and creating various machine learning models or algorithms for specific, company-required applications in the beauty sector, then implementing them using the aforementioned framework.

These overall goals resulted in the following objectives:

- Creation of a cross-platform framework to allow computer vision machine learning (CVML) models to be run on the CPU and GPU in smartphone Unity apps, in real-time for applications that demanded it (Section 2.4).
- Extension of the GPU capabilities of the framework, to allow more CVML smartphone models for various tasks to be run with hardware acceleration (Appendix B).
- Real-time six degree-of-freedom (6DoF) pose tracking of a handheld object on Android and iOS smartphones (Chapter 3). *6DoF* means the 3D position and 3D rotation of the object.
- Smartphone-friendly makeup style transfer for faces, meaning the transfer of an unseen target style in a face image to another unseen face image (Chapter 4).
- Fast detection of wrinkles and blemishes on smartphones, using classical computer vision methods, without machine learning (Chapter 5).
- Generation of faster and more accurate machine learning architectures for real-time semantic segmentation of faces on smartphones, using neural architecture search with on-device latency awareness (Chapter 6).

1.3 Contributions

In general, the previously mentioned goals were meant to result in a tangible implementation, usable in a product. To reach this stage, they required research, data preparation, training, testing, incorporation in the Unity system, and refining. These resulted in practical contributions. While not all projects reached the final stages, some still had theoretical contributions resulting from research and design.

As a result, this work has the following contributions:

1.3.1 Theoretical contributions

- A novel architecture for 6DoF object pose detection, based on the single-shot-pose method from Tekin et al. (2018), capable of running in real-time on mid to

high-end Android and iOS smartphones (Chapter 3).

- An extension of a method for wrinkle detection from Elbashir and Yap (2020) to allow for generic blemish detection (like acne and moles), based on the blob detection method from Jerman et al. (2016), and the description of a potential alternate method using Gabor filters, similar to Batool and Chellappa (2015) (Chapter 5). By not relying on machine learning datasets, this method had lower implementation difficulty than existing machine learning-based methods.
- A mobile-friendly version of the makeup style transfer network from Li et al. (2018b), with several improvements to the output quality (fewer output artifacts and resistance to poorly lit faces with heavy shadows) (Chapter 4).
- Several new, mobile-friendly segmentation supernet architectures based on the FBNet classification network from Wu et al. (2019), and several resulting final network architecture variants (Chapter 6).

1.3.2 Implementations and practical contributions

- A prototype C++ plugin that could run TensorFlow Lite models on the CPU on Android, which aided in the creation of a production-grade plugin capable of running models on Android and iOS, both on the CPU and GPU (Section 2.4). For development purposes, it also supported CPU inference on desktop platforms (MacOS and Windows). This became the groundwork for most of the projects of this thesis, as well as other projects done by the company.
- A real-time 6DoF hair curler tracking system using two machine learning models, with good accuracy aside from the roll angle, integrated into a Unity app by means of the TensorFlow Lite plugin (Chapter 3). A version of this app with a single pose detection model and no tracking was delivered to the client company that had ordered its creation.
- The implementation of both wrinkle and blemish detection as a Unity C++ plugin, integrated with the face segmentation and tracking system of the company (Chapter 5).
- A Unity implementation for the new variant of the makeup style transfer, capable of running on the CPU on Android and iOS (Chapter 4). Furthermore, a variant of the instance normalization layers, required for this and many other generative adversarial networks, was created. This worked on TensorFlow Lite GPU on both Android and iOS, albeit slowly.
- A working Unity implementation for the FBNet segmentation network, using the existing TensorFlow Lite Unity plugin, which however did not have better accuracy than the non-NAS versions (Chapter 6).

1.4 The placement company

Given the target industry of the company, all the projects included in the doctorate were beauty-oriented, and generally targeted mobile phone operation. The resulting augmented reality applications mainly used the selfie (front-facing) camera of the smartphone, and targeted the face of the user.

1.4.1 The codebase

The software used by the placement company was generally centered on Unity. While reasonably heavy on computational resources, Unity allows for simple integration of complex 3D elements, and is the most commonly used game engine on smartphones. In general, Unity is relatively easy to program for, as its scripting uses C#. By contrast, Unreal Engine, another commonly used game engine, uses C++, which is a middle level language, requiring more boilerplate code, careful optimization, and custom memory management. However, using C# leads to a small performance hit relative to C++, which raises both a direct need for performance-critical components to be written as C++ plugins, as well as an indirect need, since many performance-oriented third-party libraries, such as OpenCV, are C++ libraries. As a result, the company's Unity projects are written as a combination of C# code and C++ plugins.

Through multi-platform targets in the C++ plugin builds, use of Unity, and efforts in the core code, simple cross-platform use is generally ensured at the high C# level, for iOS, Android, multi-platform Internet browser, MacOS, and Windows support (the latter two generally for development purposes).

Multiple forms of apps using this codebase were created by the company, either for presenting to other companies as demos or commissioned works, or for integrating into the company website as part of the direct-to-consumer (D2C) web apps. For the purposes of testing different computer vision implementations and machine learning models, a tester Unity project was used, which we here refer to as *Tensorflow-Lite-Tester* or TFLT (Subsection 2.4.4).

1.5 Ethical considerations

Foremost, beyond the academic goals and value to the scientific community, the purpose of the work in the thesis projects, particularly those aimed at facial beauty, was to drive users to purchase either the apps themselves, or beauty products displayed in the apps. This was unavoidable due to the nature of the thesis as a collaboration with a company with commercial interests.

In addition, it was ethically necessary to ensure the fairness of the machine learning and computer vision outputs. When dealing with computer vision or machine learning methods aimed at ascertaining human features and traits, a common ethical pitfall is the lack of a sufficiently diverse dataset. In our case, of particular concern was good output precision for human faces of different skin colors, ethnicities, and ages. This concern was mainly caused by the observed lack of diversity in some training datasets, for example that used for the face makeup transfer experiments in Chapter 4, for which the non-makeup dataset was exclusively formed of light-skinned Asian faces, and the makeup dataset was formed only of light-skinned faces. To mitigate this when possible, efforts were made to supplement insufficiently diverse datasets with additional data, using manually or automatically annotated data, or even CGI faces.

A section on project-specific ethical considerations, particularly regarding dataset diversity and accuracy on underrepresented groups, is provided at the end of each chapter. Additionally, given that the value of the research to the greater scientific community is also an ethical consideration, we also discuss possible applications of the project research beyond the beauty industry.

Chapter 2

Literature review and background

Over the past decade, machine learning with convolutional neural networks has transformed computer vision, leading to significantly improved capabilities in robotics, autonomous driving, and augmented reality. All of these areas require various forms of machine learning for the different perception tasks they address. For example, autonomous driving may require pose estimation and classification for objects on the road (to avoid other cars or pedestrians), pattern recognition for street signs, as well as depth perception and mapping in order to navigate the road itself.

In all cases, many of the required machine learning tasks need to be made real-time, since the perceived environment may rapidly change, or the camera may change its position with respect to the environment. In the case of autonomous driving, this can be due to the motion of the car, other cars, or pedestrians on the road, and in the case of augmented reality, due to the need to maintain virtual objects tracked to the real world as the camera moves.

Given these requirements, and the general tendency of convolutional neural networks to be slow and resource-heavy without significant optimization efforts, there have been many works on real-time machine learning for many computer vision tasks, such as classification (Howard et al. 2019), semantic segmentation (Yu et al. 2021), and human pose estimation (Google 2018; Bazarevsky et al. 2020).

Optimization issues are even more apparent in many augmented reality applications for which the target running hardware is not a desktop computer or a powerful mobile GPU, but instead just a smartphone, or even augmented reality glasses. While optimized frameworks and machine learning models have been created for a number of common tasks for smartphone AR, some less common tasks require novel methods,

or optimization of existing methods designed for more powerful hardware. While especially true for real-time neural networks, the constraints can also affect execution of models that do not need to be real-time, as unoptimized models may be too large to fit in smartphone memory, or may run too slowly for even a single inference to take an acceptable amount of time. Another significant drawback of smartphones is that they only come with integrated GPUs, all of which share memory resources with the CPU, unlike the dedicated desktop GPUs with their own VRAM. Since GPU execution is most often required both on desktop computers and smartphones for real-time applications, this is a further significant limitation for smartphone machine learning.

Some existing works are detailed below for augmented reality tasks that became relevant in the research of this thesis, comprising methods both not smartphone-oriented, as well as smartphone-oriented if available. In some cases, these may not involve machine learning, but instead traditional computer vision tasks, if they are better suited for the required purpose.

2.1 Occlusion implementation and SLAM for augmented reality

One of the most significant challenges faced by augmented reality is the implementation of occlusions, through which a virtual object in a real scene is made to appear behind real objects that are closer to the camera. While anchoring virtual objects to 3D positions in the real world does contribute significantly to the illusion of AR, and can be made to work reasonably well even on mobile devices currently, lack of occlusion handling tends to break immersion.

Rough occlusion can be implemented via SLAM (Simultaneous Localization And Mapping) algorithms. SLAM methods create a coarse 3D mesh reproduction of the environment of the sensing equipment, while simultaneously tracking the position of the sensor in the environment (Jamiruddin et al. 2018). This can be accomplished by detecting and analyzing the motion of landmarks between subsequent frames (Durrant-Whyte and Bailey 2006). The resulting mesh can then be used to create occlusions for the desired virtual objects (Holynski and Kopf 2018). While not traditionally machine learning-driven, some of the more recent approaches do use neural networks for improved results (Mittal et al. 2021; Zhang et al. 2018; Bahraini et al. 2019).

In spite of its computational cost, research for simultaneous localization and mapping (SLAM) on smartphones has existed for over a decade (Klein and Murray 2009). Given the availability of low-precision inertial measurement units (gyroscopes and accelerometers) on most current mobile phones, these can also be leveraged to

further inform the SLAM system through Kalman filters. Following the former Google Tango project (Marder-Eppstein 2016), Google's ARCore (Google 2020a) and Apple's ARKit (Apple 2021) are as of 2021 the main platforms for mobile phone augmented reality with SLAM (Nowacki and Woda 2020).

A large issue with SLAM approaches is that they require at least part of the environment to be scanned first. This is because camera motion is required to create the mesh when using monocular RGB input, the environment being re-created using structure-from-motion (Munguía and Grau 2012). Furthermore, subsequent updates to this environment mesh are generally slow, especially on mobile devices. A machine learning approach for depth estimation could instead allow for occlusion handling from a single frame, with subsequent frame detections potentially improving the overall precision of the environment scan.

2.2 Monocular depth estimation on mobile phones

Monocular depth estimation is only a small subset of research on depth estimation in general (Bhoi 2019; Liu et al. 2020; Laga et al. 2020). Robotics and autonomous driving applications generally use stereo (Li et al. 2018a) or multi-view cameras (Chen et al. 2017b) at runtime, as well as radar (Dickmann et al. 2016; Kim et al. 2015), ultrasonic sensors (Borenstein and Koren 1988), or LIDAR (Royo and Ballesta-Garcia 2019), sometimes combined (Taraba et al. 2018, Tesla 2021), as multiple sensors greatly simplify the task. Estimating depth only from a single monocular 2D frame is by nature an ill-posed problem, since there are an infinite number of 3D scenes that produce the same 2D image projection. But by training on real color images with depth (RGB-D) or stereo images, the most probable depth map can be inferred by machine learning models (Poggi et al. 2018; Kuznietsov et al. 2017; Godard et al. 2019).

Many monocular depth estimation works are geared towards autonomous driving applications (Neven et al. 2017; Godard et al. 2017; Harisankar et al. 2020), using mainly the KITTI dataset (Geiger et al. 2012). This dataset offers LIDAR laser scanner depth data, as well as stereo video. For datasets of indoor scenes, Kinect data is often used (Hodaň et al. 2017; Lai et al. 2014, 2011; Silberman and Fergus 2011; Silberman et al. 2012; Song et al. 2015; Xiao et al. 2013), with some stereo video also being available (Scharstein et al. 2014). Following the work of Godard et al. (2017), use of stereo video for network training by enforcing left-right consistency has been generally favored over using depth data directly, as this provides better results and is easier to capture. Given that sequential frames offer significant depth cues, both for machine learning and traditional approaches, many depth estimation networks are recurrent

(Wang et al. 2019b), or are trained on stacks of frames (Zhou et al. 2017).

There is some research targeting embedded-system monocular depth estimation, but this mainly targets NVIDIA Jetsons (Oh et al. 2020; Wofk et al. 2019), and occasionally Raspberry Pi CPUs (Poggi et al. 2018; Peluso et al. 2019). Some approaches have been ported to mobile, such as Pydnet (Poggi et al. 2018; Aleotti et al. 2021), which achieves about 30 fps on an iPhone X. Also, the approach in FastDepth (Wofk et al. 2019) achieves about 41 fps on an iPhone X in their example video (Wofk et al. 2020), but shows a lack of temporal coherence, as it only does single-frame inference.

As of June 2020, monocular mobile-oriented depth perception is also available through ARCore via the Depth API (Google 2020a). ARKit also offers its own Depth API, which also provides a mesh reconstruction of a real scene, but this relies on *Light Detection and Ranging* (LIDAR) scanners (a type of depth detector using lasers), currently available on only 2 iPad Pro models as of August 2020 (Apple 2021).

Non-realtime monocular depth estimation for still images on mobile is achievable with a multi-frame focal stack (multiple images of a target taken with a different focal length) as in Suwajanakorn et al. (2015). On certain recent mobile phones, such as the Google Pixel, a approximate form of depth estimation for depth of focus simulation can be accomplished for portrait pictures, by using human segmentation with neural networks (which can be made real-time) and selective blurring, as was accomplished in Wadhwa et al. (2018). In this same work, when using cameras with dual-pixel autofocus hardware, a weak disparity map could be obtained in close-up (macro) photos, allowing for post-processing depth of focus effects in close-up shots as well. Since almost all front-facing (*selfie*) cameras are fixed-focus, the latter tactic is generally restricted to the rear-facing camera.

Given these existing works on monocular depth perception, there was a desire to attempt implementation on a mobile device, primarily for the purpose of improved occlusion handling. This did not materialize into a full-on project, beyond the initial literature review.

2.3 Adaptation of machine learning to mobile platforms

While applying machine learning to mobile platforms still carries significant challenges, many tools and techniques for this purpose appeared before or during the work in this thesis, which significantly eased our research efforts.

In all the machine learning-based projects in this thesis, most of our optimization work for creating mobile-friendly neural network architectures was based on MobileNetV2 (Sandler et al. 2018), a feature extractor and classification network specifi-

cally designed for fast smartphone inference. The speed and precision of this network was due to its reliance on depthwise separable convolutions, a powerful but low-latency alternative to normal convolutional blocks of desktop-oriented networks.

2.3.1 MobileNetV2 and depthwise separable convolutions

The MobileNetV2 network built upon the earlier MobileNetV1 (Howard et al. 2017), which used depthwise separable convolutions as a central component. These convolutions and variations on them are a fundamental ingredient in most smartphone-oriented vision models today. Originally introduced in Sifre and Mallat (2014), also for computer vision models, depthwise separable convolutions had already seen usage prior to MobileNetV1 in XCeption (Chollet 2017). Depthwise separable convolutions are in fact formed of two convolutions - a (normal) depthwise convolution, and a pointwise convolution.

As shown in their respective implementations in the PyTorch machine learning platform (Paszke et al. 2019), both normal and depthwise convolutions are special cases of grouped convolutions. In a normal convolution operation, each channel of the input tensor is convolved with each of the filters of the convolution weights. In a grouped convolution with two groups, each half of the input tensor (along the channel axis) is convolved with the grouped convolution kernel, and the two resulting outputs are concatenated. This results in a lower computational cost. To be able to exactly divide the input tensor, the number of groups in a grouped convolution must be a divisor of the number of input channels. If it is equal to the number of input channels itself, it is a depthwise convolution. Analogously, the original standard convolution is a grouped convolution with a single group.

As an example, in PyTorch, the type of grouped convolution is controlled by setting the *groups* argument of the Conv2D constructor to 1 in the case of normal convolutions, and to the number of input tensor channels in the case of depthwise convolutions. This leads to the example implementation of a depthwise separable convolution in Code Listing 2.1.

```
class DepthwiseSeparableConvolution(nn.Module):
    def __init__(self, in_layers, out_layers, kernel_size_dw=3):
        super(depthwise_separable_conv, self).__init__()
        self.depth_wise = nn.Conv2d(in_layers, in_layers, \
            kernel_size=kernel_size_dw, padding=1, groups=in_layers)
        self.point_wise = nn.Conv2d(in_layers, out_layers, \
            kernel_size=1)
```

```
def forward(self, x):
    out = self.depthwise(x)
    out = self.pointwise(out)
    return out
```

Listing 2.1: Example depthwise separable convolution implementation in PyTorch

MobileNetV2 modified the depthwise separable convolution block of MobileNetV1 by using linear bottlenecks. This involves a second pointwise convolution before the depthwise layer, and the removal of the ReLU6 activation for the second pointwise convolution, making it a linear operation. Since the number of input and output channels of the depthwise convolution can now be any multiple of the input channel counts of the block itself, this creates a new parameter known as the *expansion factor*, equal to the ratio of the filter numbers of the output and input of the first pointwise convolution.

These blocks put a greater limit on the size of intermediate tensors, thereby reducing memory consumption, which is important for smartphones. As a result, the largest intermediate tensor in MobileNetV2 is four times smaller than the largest in MobileNetV1.

MobileNetV2 also improved on the original MobileNet by using residual connections, also known as skip connections. These involve adding the input of a sequence of layers to its output, creating a *residual block*. These were introduced in He et al. (2016), where they were demonstrated to counteract the degradation problem, in which the accuracy of networks hits a saturation point as the network becomes deeper, after which the accuracy quickly degrades. This was later shown to also mitigate the *vanishing gradient* problem (Veit et al. 2016), which had already been partially mitigated by the introduction of batch normalization and normalized network initialization. This is because residual networks effectively create shorter paths in deep networks, effectively becoming multiple shallow networks working in tandem. Residual networks also have a stable backpropagation (Zaeemzadeh et al. 2021), leading to a stable training, because residual connections are norm-preserving. This means that in the backward path, the norm of the gradient with respect to the input of the residual block is equal to the norm of the gradient with respect to the output.

While originally residual blocks usually contained standard convolutions, MobileNetV2 applied these residual connections to its bottleneck depthwise separable convolution blocks, forming the residual blocks seen in Figure 1.

MobileNetV2 also removed the fully connected layer from the end of MobileNetV1, which helped decrease the parameter count. Overall, with its cumulative improvements over MobileNetV1, MobileNetV2 was shown to reduce the parameter count from 4.2 to 3.4 million, while improving the top 1 ImageNet classification result from 70.6% to

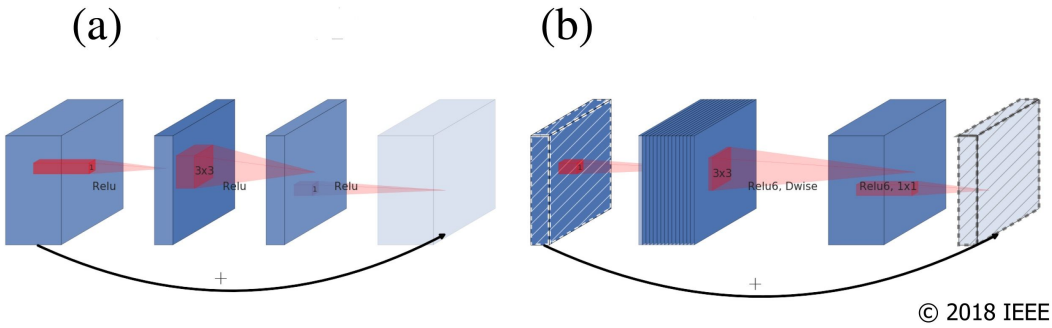

Figure 1: Residual block (a) and inverted residual block (b), both used in MobileNetV2. In both block types, the input is added to the output. Figure from Sandler et al. (2018).

72.0%.

The existence of MobileNetV2 and derivative works allowed us to focus on mobile-friendly network design based on combining MobileNet-based feature extractors or convolutional blocks with task-based elements of desktop-oriented networks, such as YOLO-based end layers for 6DoF object pose tracking (Chapter 3) or UNet-style connections for semantic segmentation (Chapter 6). This allowed us to place greater focus on implementation work.

2.3.2 Tools and challenges for machine learning on mobile platforms

In terms of implementation for mobile devices, we were again able to greatly rely on existing work, such as the TensorFlow, TensorFlow Lite and TensorFlowJS platforms (see Section 2.4). These also allowed for various optimizations such as quantization and hardware acceleration out of the box, in many cases with minimal changes or restrictions to our models. The optimized TensorFlow Lite or TensorFlowJS models can be created automatically from TensorFlow models, allowing for a relatively seamless transition from network training to desktop inference to on-device inference.

Despite this, several implementation issues still arose, requiring additional work. One was the difficulty of converting PyTorch-trained networks to TensorFlow Lite, which ultimately had to be done manually for the 6DoF object pose estimation and face makeup transfer projects (Chapters 3 and 4, respectively).

Another was the lack of TensorFlow Lite GPU support for some of the layers in our networks, despite support for those same layers in TensorFlow Lite CPU. Given that we became largely reliant on hardware (GPU) acceleration due to its significantly lower latency, this was a significant problem, and using different supported layers for

the problem networks was not always feasible. After some efforts to add new operations to TensorFlow Lite GPU (Appendix B), the required functionality was implemented using Unity HLSL shaders and OpenCV CPU operations.

Another issue was the large number of different smartphones that needed to be supported by the resulting Unity apps, not all of which supported GPU operation, and had varying degrees of computational power and network latency. Based on the neural architecture search project in Chapter 6, we intended to eventually generate multiple variants of our networks for different devices, using the on-device latencies of different candidate network building blocks. Given the negative results of the method, we instead relied on manual testing on multiple smartphones, sometimes using Google Firebase Test Labto partially automate this using cloud computing and remote devices (Google 2021b).

Finally, and most importantly, TensorFlow Lite (and later TensorFlowJS) needed to be integrated into Unity before any of the thesis projects could be finalized, to allow the required Unity apps to have machine learning-based capabilities. This is described in the next section, along with the rationale for using the TensorFlow Lite library for our work.

2.4 Integration of TensorFlow Lite with Unity3D

Almost all the projects in this work use machine learning for computer vision and scene understanding. Given the requirement to implement the research on real hardware, as well as the innate engineering aspects of modern machine learning for computer vision, the choice of the appropriate API for these tasks was very important. The most significant requirement of this API was support for smartphone applications, as mobile hardware is significantly different from desktop computers or servers. A further requirement was significant inference speed, given the real-time execution needed for some of the projects.

These requirements lead to the choice of TensorFlow Lite (Google 2021h), which we discuss in this chapter, and outline the changes necessary to make it applicable for the goals set in this work. The reasoning for using it over the other potential mobile-oriented APIs is shown in Subsection 2.4.2.

TensorFlow Lite is offered as part of the TensorFlow API. Introduced in late 2015, TensorFlow (Abadi et al. 2016) is a machine learning platform created by Google, for neural network training and inference on a variety of running environments, generally desktop computers or powerful servers. Alongside PyTorch (Paszke et al. 2019), it is one of the most well-known machine learning platforms.

For the purposes of inference on low-powered devices like smartphones, while it was possible to build and operate TensorFlow on iOS and Android in a similar fashion to desktop use (Google 2019, 2020c), this has been deprecated in favor of TensorFlow Lite. TensorFlow Lite is an inference-only lightweight framework specifically tailored for low-powered devices, such as Android and iOS smartphones, as well as embedded Linux devices and microcontrollers (Google 2021h).

2.4.1 Overview of the TensorFlow Lite API

The main benefit of TensorFlow Lite, relative to just the desktop version of TensorFlow built for smartphone platforms, is improved model inference speed. This partially results from basing the machine learning model format on Flatbuffers serialization (Google 2021c). For mobile inference, this is superior to the TensorFlow protocol buffer format, given its smaller file size and direct data access capabilities, which help speed up inference and lower resource consumption (Google 2021i).

Another significant factor in TensorFlow Lite inference speed is mobile hardware acceleration support. Desktop computers use discrete GPUs (Nvidia or AMD) or various machine learning-oriented accelerators (like Google's Tensor Processing Units), much unlike smartphones, which use integrated GPUs to keep the power consumption low. As a result, normal desktop-oriented TensorFlow does not support hardware acceleration on smartphones, unlike TensorFlow Lite. The latter's hardware acceleration capabilities are discussed in Subsubsection 2.4.3.

While generally providing less of a speedup than hardware acceleration, faster model inference can be achieved through quantization. This involves using lower-precision tensor arithmetic during inference, instead of the usual 32-bit float precision. This is discussed in Subsubsection 2.4.3.

To ease implementation, TensorFlow Lite provides Java, Swift, Objective-C, C++, and Python bindings to its low-level C/C++ code, and employs Metal, OpenGL, and OpenCL for hardware acceleration on iOS and Android smartphones (Google 2021h). For testing purposes, it also allows for model inference on desktop computers, using the Python API, with some support for using the low-level C/C++ API's directly.

2.4.2 Comparison of TensorFlow Lite with similarly-purposed APIs

Given the lightweight nature, focus on smartphone inference, and mobile hardware acceleration support of TensorFlow Lite, any alternative API must at least have these same capabilities. A potential further benefit would be already implemented integra-

tion with Unity3D. All smartphone or Unity-oriented machine learning APIs we could identify are described below, with justification given as to why they were not chosen over TensorFlow Lite.

Caffe2

Before the advent of TensorFlow Lite, Caffe2 (Jia et al. 2014; Facebook 2021c) appears to have been the primary mobile-oriented machine learning model execution library, providing an alternative to simply building desktop-grade libraries for mobile targets (such as the deprecated TensorFlow for Android). Currently, Caffe2 is offered as part of PyTorch (Facebook 2018).

While Caffe2 does offer support for both Android and iOS (Facebook 2021c), development appears to have stagnated. Some effort appears to have been made to add Android GPU support through Vulkan and OpenCL, but the results seem to exist only as unmaintained stubs (Facebook 2020b,a). Given the advent of PyTorch Mobile in late 2019 (Facebook 2021a), which offers similar but improved functionality, Caffe2 appears to have been superseded as a smartphone machine learning platform.

Barracuda

Barracuda (Unity Technologies 2021a) is a machine learning model execution framework specifically for Unity, which supports iOS and Android execution, on both CPU and GPU (Metal on iOS and Vulkan on Android). Its training capabilities are mainly geared towards AI elements in Unity games, as part of the ML-Agents project (Unity Technologies 2021b), but it can be used as a general-purpose neural network inference system.

While this is likely the best alternative to TensorFlow Lite for use with Unity, as its tight integration removes the need for constructing all the required C++-C# bindings, Barracuda appears to have been made public in May 2019, when our attempts at a TensorFlow Lite implementation had already been underway for several months, and GPU support with OpenGL and Metal had already been added to TensorFlow Lite (Lee et al. 2019). Furthermore, Barracuda appears to be a significantly smaller project than TensorFlow Lite, with development occuring at a slower pace, which is likely to lead to it being a less robust and stable framework.

Finally, given our increased use of TensorFlow as opposed to PyTorch, using TensorFlow Lite greatly simplified the conversion from training model to on-device inference model, as Barracuda uses the open-standard ONNX format instead of the TensorFlow Lite .tflite format. While compatible with PyTorch and many other frame-

works, the ONNX format is not fully compatible with TensorFlow, as has been observed in our efforts in converting PyTorch models to TensorFlow Lite (Subsection 3.7.1).

PyTorch Mobile

PyTorch released PyTorch Mobile in October 2019 for neural network inference on Android and iOS devices (Facebook 2021a). While it supports 8-bit quantization, GPU support has not yet been added as of August 2020, but appears to be in the roadmap, for both Android and iOS (Facebook 2021b). This makes it unsuitable for our purposes, even barring the TensorFlow Lite-analogous C++ to C# bindings that would need to be added for use with Unity. But in general, after GPU support is added, PyTorch's ease of use compared to TensorFlow could give it an edge in training networks for mobile platforms, although TensorFlow 2 has caused this ease-of-use gap to shrink.

As a result, TensorFlow Lite was the most suitable framework for this research.

2.4.3 Latency reduction techniques in TensorFlow Lite

Given the focus on real-time performance in some of the thesis projects, there was a need to ensure inference speed as much as possible, while maintaining sufficient accuracy.

In TensorFlow Lite, there are a number of ways to speed up model execution, relative to the default 32-bit precision inference on the smartphone CPU. These include using multiple CPU threads, model quantization, hardware acceleration, or alternate execution backends like XNNPACK (Google 2021d). These are presented and discussed below.

8-bit quantization for fast CPU inference

It has been observed (Jacob et al. 2018) that a model with lower-precision weights and operations is often still capable of inference with just a small accuracy hit. This is known as *model quantization*. TensorFlow offers INT8 quantization through the tensor arithmetic shown in Equation (2.1) (Google 2021j), which is a variation of the method in Jacob et al. (2018). While quantized weights cannot be used directly during training, as this breaks backpropagation, quantized inference can be achieved either through training-time quantization (which uses quantization nodes during later training stages to determine the zero point and scale values for each op), or post-training quantization (which uses post-training inference on a representative dataset to determine those same values). The latter was found to be easier to implement, so it was our preferred method.

While it is possible to use even lower than 8-bit precision in inference without

losing too much accuracy (Choukroun et al. 2019), TensorFlow Lite does not support this.

$$real\_value = (int8\_value - zero\_point) \cdot scale \tag{2.1}$$

While generally superseded by hardware-accelerated inference, INT8 quantization can be a significant latency optimization technique for some older devices, which cannot use the TensorFlow Lite GPU or DSP delegates listed in the next section. This is the case for Android devices without valid OpenCL libraries and OpenGL ES versions below 3.1, and some very old iOS versions. INT8 quantization is also used by other machine learning frameworks for smartphones that do not have full support for hardware acceleration, like PyTorch Mobile, as of April 2021 (Facebook 2021b).

Different types of quantization have been added to TensorFlow Lite during the duration of this project (such as the INT8 quantization with INT16 activations, for certain more accuracy-sensitive applications), but these are outside our scope, as our main focus was hardware accelerated inference on the smartphone GPU. This was observed to be consistently faster than INT8-quantized CPU inference, which is consistent with the latency benchmarks reported in for multiple off-the-shelf models often used for inference on smartphones (Google 2021e,f).

GPU inference

By default, the tensor calculations required for inference are done on the smartphone's CPU. In TensorFlow Lite, there are a number of alternate executors which override this default CPU model execution, generally with the goal of providing faster inference, by performing the calculations in a different way. These are known as *delegates*, and generally involve leveraging the smartphone GPU.

On Android, the OpenCL and OpenGL GPU delegates are available, while the Metal GPU delegate is used for iOS. Each of these provide inference execution on the GPU by using the shading language they are each named after. In addition, the Neural Network API delegate for Android and the CoreML delegate for iOS, specified in the next subsubsection, may also internally leverage GPU execution.

The GPU delegates can speed up inference additionally by restricting precision to 16-bit (half) float instead of the default 32-bit float operation.

Other delegates

To ensure a complete review of all possible acceleration methods with TensorFlow Lite delegates, as to not miss any potential options for our real-time inference projects,

all of the remaining delegates were briefly tested on a OnePlus 6 Android device and an iPhone XS. Later tests were carried out on a larger amount of devices by other placement company employees.

On iOS devices, the CoreML delegate can use the Neural Engine, or fall back to GPU or CPU inference if this is not available or does not support the model. The fallback GPU inference uses Metal internally. In most of our tests, it did not show any benefit over just using the Metal delegate outright.

On Android devices using certain Snapdragon processors, the Hexagon digital signal processor (DSP) delegate is available. This provides a hardware acceleration alternative to using the GPU. In some of our tests on a OnePlus 6 Android device, the Hexagon delegate was observed to outperform the OpenCL and OpenGL delegates in several off-the-shelf models. This is due to it both providing hardware acceleration and by using, and being restricted to using, integer-quantized models. Despite this, given the significant amount of devices that did not support it, as well as the model quantization requirement, this delegate was generally not taken into consideration.

The XNNPACK delegate runs on Android and iOS CPUs. While it required full-float models at the time of testing, it was observed to be slightly faster than quantized CPU inference on multiple Android devices, and could occasionally outperform the OpenGL delegate when multiple execution threads were used. However, it virtually never outperformed the OpenCL delegate when using full float precision.

Some later tests, after quantized model inference support was added to XNNPACK, showed the delegate to outperform the OpenCL delegate as well in some cases, when using sufficient execution threads.

The Neural Network API of the Android NDK has its own delegate, supporting both quantized and unquantized models. At the time of our tests (around mid-2018), attempting to use this delegate led to abnormally large latency increases, with some models with sub-second 32-bit CPU inference taking multiple seconds to run with the NNAPI delegate. As a result, it was generally disconsidered. Given the more reasonable latency values noted in Google (2021g) for the NNAPI delegate on the Pixel 3, retrieved in late 2021, these issues may have been fixed. The NNAPI delegate may therefore be more relevant for future work.

2.4.4 Creation of our TensorFlow Lite plugin for Unity

As mentioned in Subsection 1.4.1, the existing company codebase was based on Unity3D, which is the most commonly used game engine for 3D smartphone applications, including augmented reality ones (Bonfiglio 2018). Despite the support for mobile operation, Unity apps can still be a significant resource and power drain. Since simple graphical

elements can be drawn without a game engine on both iOS and Android, using Unity is not always the best option. For some of the projects in the thesis, with relatively simple graphics, this was arguably the case. However, given the additional development effort required to adapt the codebase to not rely on Unity, the implementations of all the machine learning and computer vision projects in this thesis were based on Unity. As a result, in order to allow for machine learning inference, integrating TensorFlow Lite with Unity was required. This was done in a manner similar to the already existing C++ Unity plugins of the company, by taking advantage of the C/C++ API offered by TensorFlow Lite.

Creating a C++ plugin involves building the required C++ functions into a native library, meaning a separate version of the library must be built for every target platform. This allows these C++ functions to be called from Unity through C# bindings (extern functions), as programming in Unity is done in C#.

The company already employed a variety of C++ Unity plugins in its codebase, largely as part of its proprietary augmented reality platform. This is because despite the added complexity of C and C++ code compared to C#, such plugins allow for speed increases crucial to augmented reality applications, in part due to the use of highly optimized C++ libraries such as OpenCV.

The resulting system is capable of running models on multiple platforms from within Unity, and has been used both for thesis projects, as well as a number of the company's custom apps and web apps, either for clients or for company-hosted projects. This system allows for inference of arbitrary models within Unity on the CPU (Android, iOS, MacOS, Windows) and the GPU (Android, iOS, MacOS), as long as they are supported in TensorFlow Lite. It has been used for real-time facial segmentation, facial keypoint detection, and fingernail segmentation with internally designed and trained models.

To test and benchmark TensorFlow Lite models directly on smartphones, the main devices used were a OnePlus 6 Android device (running Android 8, 9 or 10 over the thesis period) and an iPhone XS iOS device (running iOS 12, 13 and 14 over the thesis period). For more in-depth testing for production-ready applications, some manual testing on multiple other devices was done by other company employees. This was later replaced with an automated system, using Google Firebase Test Lab, which allows for testing on remote devices in the cloud (Google 2021b).

2.4.5 TensorFlow-Lite-Tester

In terms of implementation in the company codebase, the integration of TensorFlow Lite into Unity resulted in an overarching Unity project called TensorFlow-Lite-Tester.

Figure 2: Outputs from TensorFlow-Lite-Tester, showing the makeup virtual try-on capability. Original image by א (Aleph) under CC BY-SA 2.5 (Aleph 2008). Image has been cropped (left), and the crop shown with various overlays (others).

While this was more of a testing project for various computer vision tasks, eventually turning into an amalgamation of different subprojects, the primary focus of this Unity project was the alignment of different pre-designed makeup styles onto a person's face, in real-time. This is referred to as a *virtual try-on*, illustrated in Figure 2, and is sought after for the purposes of offering makeup trials remotely, not in-store, thereby driving sales. Similar capabilities are implemented in other beauty apps, such as Perfect365 (Perfect365 2021) and YouCam Makeup (PerfectCorp 2021). In the TFLT project, the virtual try-on is handled mainly through machine learning models, and allows for the application of makeup, lipstick, and changes in hair color.

From a development perspective, TFLT allows for the testing of machine learning models, mainly for face tracking and semantic segmentation, but also traditional computer vision algorithms, like the wrinkle and blemish detection system described in Chapter 5. It supports iOS, Android, Windows, MacOS, and browsers, both for running the models on the CPU, as well as the GPU. Given that it uses much of the same code as the various makeup apps created by the company, TFLT provided a testing ground and benchmarking capabilities for the various machine learning and computer vision tasks listed in the following chapters.

Unlike normal Unity GPU operations, which are written in the HLSL shading language and automatically compiled to the target device shading languages, GPU operations in TensorFlow Lite need to be written in the target languages directly, requiring OpenGL and OpenCL implementations for Android, and Metal implementations for iOS and MacOS. While cross-platform shaders for Windows, Android, iOS, MacOS, and even Linux could be possible using Vulkan, using MoltenVK in the case of iOS and MacOS, TensorFlow Lite does not yet have Vulkan implementations of the existing GPU ops, and their addition is not currently part of the roadmap, as of July 2021 (Google 2021k). As a result, implementation of new TensorFlow Lite GPU ops is comparatively considerably more work-intensive, relative to implementing Unity

shaders.

The browser support of TensorFlow-Lite-Tester was added later for the company's direct-to-consumer (D2C) web apps, and required some significant changes. While Unity does allow for in-browser experiences through WebGL, the machine learning API needs to be switched from TensorFlow Lite to TensorFlowJS, a JavaScript-based inference library for converted TensorFlow models. This often resulted in a significant performance hit, particularly when using a smartphone browser. A result of this increased constraint was our attempt at using neural architecture search (NAS) to aid in attempts to find lower-latency networks without compromising accuracy too much, initially with only TensorFlowJS in mind, but later with TensorFlow Lite in mind as well (Chapter 6).

Browser support needed to be added because, while beauty smartphone apps are a popular way to present tech to consumers, a browser app is easier and faster to use, requiring no installation. It may also be more suitable for a simpler experience, like a tech demo. Such capabilities have been implemented into the websites of multiple beauty product vendors, such as L'Oreal and Garnier, for hair color preview (L'Oréal 2021; Garnier 2021).

Most of the projects listed in this thesis, with the exception of the non-Unity or non-ML wrinkle and blemish detection project, rely on this system, as do other projects carried out by other company employees. While in general, the capabilities of the C++ API of TensorFlow Lite were sufficient for this system, with engineering work focused on integration into Unity, some potential projects would have benefited from changes to the API itself. One of these projects, involving a scheme for faster face tracking and segmentation, was initially dependant on custom GPU operations being added to TensorFlow Lite.

The addition of these ops proved too difficult to maintain, and ultimately handled with OpenCV and HLSL operations instead.Our TensorFlow Lite op addition attempts are detailed in Appendix B. Particularly notable is the use of the operation fusion mechanism to remove tensors with unsupported data types, and to remove dynamic (variable-sized) tensors, in the case of the CROP_RESIZE operation.

This system was required for the wrinkle and blemish detection project (Chapter 5, in order to be able to extract only the targeted skin regions with a segmentation model, as well as the neural architecture search for segmentation project (Chapter 6), since it aimed to replace the segmentation model with a latency and accuracy-improved version. Furthermore, the system was used for the makeup style transfer experiments in Chapter 4, as it required bounding box detection of the face in the image frame during use. Finally, it further saw use in other company projects, not directly contributed to

by the author of this thesis.

2.5 Six degree of freedom object pose estimation

Given its applications in robotics and self-driving cars (Pauwels and Kragic 2015; Gualtieri et al. 2016; Li et al. 2018a), in which stereo cameras (Fan et al. 2020), depth cameras (Biswas and Veloso 2012; Suay and Chernova 2011) or LIDAR scanners (Royo and Ballesta-Garcia 2019) can be used to provide both RGB and depth data, a significant amount of object pose estimation approaches use RGB and depth data combined as inputs (Choi and Christensen 2012a; Aldoma et al. 2013; Wang et al. 2019a; He et al. 2020). The extra depth data makes methods using it significantly more precise and easier to implement than methods solely using RGB. However, most smartphones are only capable of depthless color input from a single camera, so only monocular RGB methods are relevant to the purposes of this thesis. Due to the additional constraints of the associated project in Chapter 3, as well as a desire to provide generalizable research, focus is placed on markerless pose tracking of custom objects.

The problem of monocular pose detection goes back to the 1980s, with graph and tree-based approaches using the appearance of the object from many viewpoints (Plantinga and Dyer 1986; Ikeuchi 1987; Munkelt 1994; Flynn and Jain 1991). This was later followed by methods using local descriptors, which used perspective-and-point algorithms to extract the 3D pose from the 2D positions of detected features (Collet et al. 2009; Wagner et al. 2010).

Currently, most approaches are based on convolutional neural networks. As in object detection approaches, pose estimation networks can be two-stage (slower but more accurate) or single-shot (faster but less accurate). Examples of the latter are SSD-6D (Pandey et al. 2018) and single-shot-pose/YOLO6D (Tekin et al. 2018). The former is based on the SSD object detection scheme from Liu et al. (2016b), and works by quantizing the pose space, treating it as multiple object detection classes. This only provides a coarse pose, and needs post-inference edge-based refinement to allow for competitive precision.

An example of a two-stage method is BB8 (Rad and Lepetit 2017), which uses a coarse-to-fine rough segmentation mask in the first step, then a neural network to locate the corners of the 3D bounding box, as a second step. The PoseCNN approach in Xiang et al. (2018) is a two-stage method which segments the objects with a higher resolution than BB8, and then regresses the centroid position and rotation (as two quaternions) in separate branches. The position branch uses Hough voting by doing pixel-wise semantic labeling of the direction from that pixel to the centroid, and the

rotation branch regresses the rotation by selecting the regions of interest and pooling the features. However, given the limited precision of the output on monocular images, PoseCNN relied on depth data using iterative closest point optimization to obtain the final, higher-precision pose data, which is not possible with only a monocular image. More recent approaches do not attempt to determine pose directly, but use 2D feature or patch detection, and subsequently project the pose to 3D via perspective-and-point (PnP) or a similar projection method. Single-shot-pose (Tekin et al. 2018), based on the YOLOv2 object detection network (Redmon and Farhadi 2017), outputs the 2D projection positions of the 3D bounding box corners, and applies perspective-and-point. The betapose approach (Zhao et al. 2018) instead localizes SIFT-selected keypoints of the objects as a second step, after cropping the input with a YOLOv3 bounding box detector, and uses the highest-confidence keypoints in the PnP step.

More recently, focus has been put on using segmentation methods to estimate pose, by querying the pixels in the segmentation mask for the position of the bounding box points. PVNet (Peng et al. 2018) uses a confidence-aware per-segmentation-mask-pixel Hough voting scheme to query the position of SURF feature points on objects with given CAD models, following a prior rough object segmentation step. In general, segmentation models such as this (Hu et al. 2019; Barowski et al. 2019) are shown to be more robust to occlusions and truncations than single-shot. However, segmentation is generally slower than single-shot methods, in spite of the lower-resolution output mask required for segmentation-based pose estimation, as opposed to generic segmentation. In addition, for handheld objects, the ground truth segmentation mask cannot be estimated solely by means of posing the CAD model over the object, as the hand will virtually always be occluding the object, and we have observed it to not be easy to mask out well without significant manual labour. Finally, in our particular use case (shown in Section 3.9), initial attempts using the method in Hu et al. (2019) often picked up background elements as mask, which caused the output pose predicted by the pixel-wise outputs to be completely incorrect. This was especially true for experiments using the entire frame as input, but seemed to be mitigated by using tracked crops as input. Investigation may be considered future work.

Some algorithms try to improve output precision with a modified post-processing projection step. Aside from the modified postprocessing in PVNet, the DeepHMap++ method (Fu and Zhou 2019) offers an improved PnP method based on the heat map generated by the feature extractor, rather than the explicit point position outputs.

Given that the pose is extracted from single frames, all the aforementioned methods generally lead to lack of smoothness in output, and do not make use of temporal coherence when inferring on video.

For the specific case of object detection in video, several methods take advantage of temporal coherence through long short-term memory (LSTM) methods, which use intermediate layer outputs from previous network inferences to aid current frame detection. One such method is ROLO (Ning et al. 2017), which uses a single LSTM unit following the heatmap output of a YOLO detection network. To avoid the memory increase normally associated with LSTMs, the approach in Zhu and Liu (2018) introduces a mobile-friendly network with Bottleneck-LSTM layers. This does not however lead to our desired reduction in required input resolution. Furthermore, the ConvLSTM is a custom network operation, requiring custom GPU kernels to run on smartphones with TensorFlow Lite.

Another method of reducing latency is to perform tracking in the compressed domain directly (Ujiie et al. 2018; Liu et al. 2019d), as compression schemes such as H.264 reduce inter-frame information redundancy by encoding motion information in block vectors. However, it requires already compressed offline video (Wang et al. 2019c) or compressed feeds (Ujiie et al. 2018), with the latter generally being restricted to security cameras, not smartphones.

Tracking can serve in aiding bounding-box object detection for unseen objects, if the initial position is provided. While some networks accomplish this by partially training the network at runtime (*online*) (Hare et al. 2011; Babenko et al. 2009; Wang et al. 2015; Kalal et al. 2011), this comes at a performance cost (Held et al. 2016). Others, like the GOTURN tracker (Held et al. 2016), are trained entirely before runtime (*offline*), using random pairs of a large number of objects, leading to fast performance during actual use.

For non-LSTM pure machine learning approaches to object tracking, some methods use separate models for detection and tracking. The BlazeFace tracking method (Bazarevsky et al. 2019) uses an improved single-shot bounding box detector with sparse keypoints, which provides a cropped region of interest for a second network with more keypoints. This differs from traditional region-of-interest models in that the second network subsequently specifies the input for itself on the next frame (a padded crop centered around the detected keypoints). This method has also been applied to 2D and 3D hand pose estimation, as part of the MediaPipe framework (Google 2020b). It is also used in our single-shot-pose-based approach in Chapter 3, to achieve acceptable accuracy.

Explicitly focusing on the tracking of handheld objects has also received some attention, in part due to applications in the visual or visual-odometric tracking of handheld virtual or augmented reality controllers. However, being VR/AR-oriented, they generally address egocentric head-mounted cameras. Pandey et al. (2018), building

on SSD-6D, investigate different variants of hybrid quantized space and extra bounding box parameter approaches, to track a small handheld tracker. The approach in Tekin et al. (2019) does joint 3D hand-object pose estimation, bypassing the PnP step by using a 3D grid with sparse depth cells, and using an recursive sub-network to also perform action recognition. As of September 2019, it appears to currently be the only joint hand-object estimator that uses only monocular RGB data.

The above techniques, with the exception of Bazarevsky et al. (2019) and some of the implementations in Google (2020b), generally address desktop computer operation. At the time of the project, no monocular 6DoF pose networks were identified, so the single-shot-pose approach (Tekin et al. 2018) was adapted in this thesis for smartphone inference, and extended to use simple tracking for better accuracy. This is detailed in Chapter 3.

2.6 Generative adversarial networks

Generative adversarial networks, or GANs, were introduced in Goodfellow et al. (2014). These are characterized by two subnetworks, the generator and discriminator, with loss functions minimized by each other's failure. The discriminator is fed both data from a real dataset, as well as data output by the generator, and is trained to classify the data as either real or fake (generated), being penalized when it classifies a generated image as real, or a real image as fake. The generator is at the same time trained to output data that the discriminator classifies as real, being penalized when the discriminator correctly classifies its output as fake. The global loss function is mathematically defined as in Equation (2.2) (Goodfellow et al. 2014).

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}[logD(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_z(\boldsymbol{z})}[log(1-D(G(\boldsymbol{z})))] \tag{2.2}$$

This loss allows for balanced, concurrent training of the generator and discriminator, through which the former will learn to output data that closely matches the real dataset, but is not part of it. Using this technique, Goodfellow et al. (2014) successfully generated realistic hand-written digits and low-resolution human faces, based on the MNIST and TFD datasets, respectively.

Later, the work in DCGAN (or Deep Convolutional GANs, introduced in Radford et al. (2016)) became a significant step in the development of realistic image generation with GANs. Through extensive experimentation, the authors formulated a number of significant architectural guidelines for both the generator and discriminator, vastly improving the output quality and laying the base for many subsequent image-generating

GANs. These included:

- the use of batch normalization in both the generator and discriminator (later replaced with instance normalization and derivatives in the case of the generator, following Ulyanov et al. (2016))
- removal of fully connected layers (which may have aided in reducing latency, in the smartphone-targeting attempts in this thesis)
- ReLU activation ($ReLU(x) = max(0, x)$) everywhere in the generator except for the output
- LeakyReLU activation (see Equation (2.3)) everywhere in the discriminator
- replacement of the pooling layers with strided convolutions in the discriminator
- fractionally-strided convolutions (also known as transpose convolutions) in the generator

$$LeakyReLU(\alpha, x) = \begin{cases} x & x \geq 0 \\ \alpha x & x < 0, \text{for a constant } \alpha, \text{ generally smaller than 1} \end{cases} \tag{2.3}$$

With the exception of the use of instance normalization instead of batch normalization for the generator, and the testing of replacing transpose convolutions due to their suspected introduction of artifacts in the output, all of these guidelines were also used in the BeautyGAN-based research in Chapter 4.

The authors of DCGAN further discovered that interpolating the latent representations of the output images (meaning the corresponding GAN inputs, which are vectors) resulted in a relatively smooth interpolation between the resulting outputs as well. By averaging out the latent representations corresponding to outputs with certain visual traits, *trait vectors* can be generated, meaning inputs corresponding to a certain facial trait or attribute, like *smiling* or *woman*. When traveling from the latent representation vector of an output without the trait in the direction of this trait vector, the output was modified to obtain the trait, as shown in Figure 3 for the trait vector $smile = smiling\_woman - neutral\_woman$. This smoothness, as well as the latent trait vector concept, are both crucial to generating user-controllable outputs, and would later be more strictly enforced in works such as StyleGAN (Karras et al. 2019) and related works, with significantly superior output quality.

There has been much research on controllable GAN outputs (He et al. 2019; Liu et al. 2019c). A commonly addressed example involves taking an input image and generating an output with only a specific feature modified and no other changes, either based on a target image (Li et al. 2018b), a predefined image style (Zhu et al. 2017),

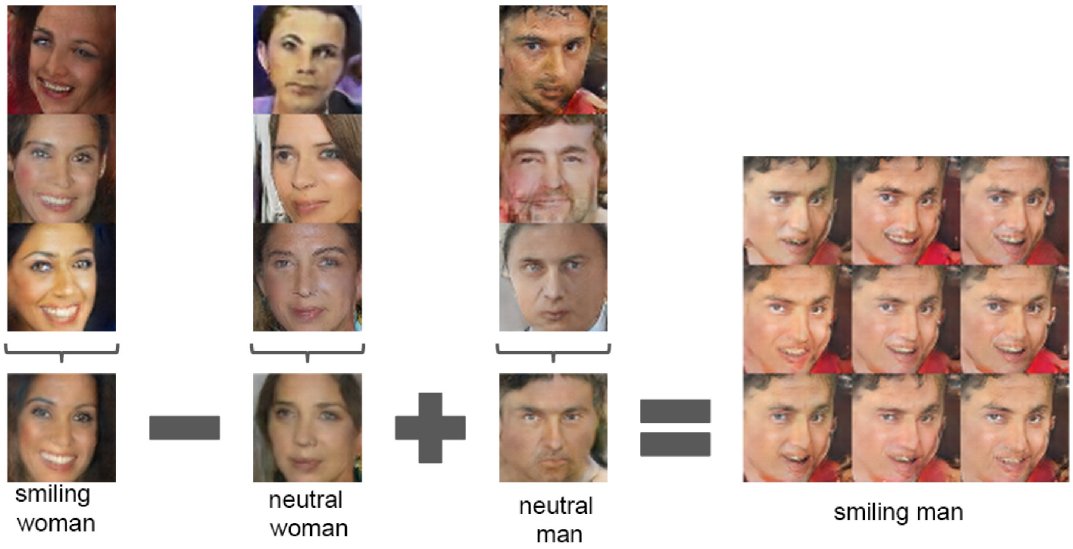

Figure 3: Latent vector arithmetic with DCGAN. Figure from Radford et al. 2016.

or a target descriptor, like one or more slider-controlled scalar values (He et al. 2019; Liu et al. 2019c).

AttGAN (He et al. 2019) is a generative adversarial network that allows for modifying features of a person, while keeping the rest of his image nearly identical. The dataset for this network contains images of faces, as well as a binary vector of attributes (such as *black hair*, *blond hair*, or *presence of beard*). The generator in this case has an encoder-decoder structure, with the decoder taking a the attribute vector as an extra input. The decoder half is run twice, once with the attribute vector of the input, and once with a randomly chosen target attribute vector. The first run is trained to output the unchanged input through reconstruction loss, while the second is trained both to fool the discriminator, as well as get a pre-trained attribute vector classifier to output the same attribute vector.

STGAN (Liu et al. 2019c) improves upon AttGAN by reducing the effect of the network outside the region of the target attribute to change. For example, applying lipstick to a target image causes fewer unwanted modifications in background, face or hair, that normally occur due to the network picking up correlations between different attributes in the training dataset.

2.6.1 GANs for image style transfer

For the purpose of style transfer using an image instead of an attribute vector as the style target, much of current research is built on CycleGAN (Zhu et al. 2017). This

network has successfully been used for tasks such as turning images of horses into images of zebras, or turning photographs into paintings of a certain artist's style (see Figure 4). For a collection X of source (*horse*) images and a collection Y of target style (*zebra*) images, CycleGAN uses two generators G (X to Y) and F (Y back to X). G learns to fool the target style discriminator D_Y by taking X_i as input and generating a fake Y_i, and F learns to fool the source style discriminator D_X by taking G's fake Y_i and reverting it to an X-style X_i'. The *cycle consistency* is the additional constraint that F's X_i' must match as closely as possible the original X_i that G took as input. These constraints are also applied in reverse, with Y as the original input. The overall scheme is illustrated in Figure 5.

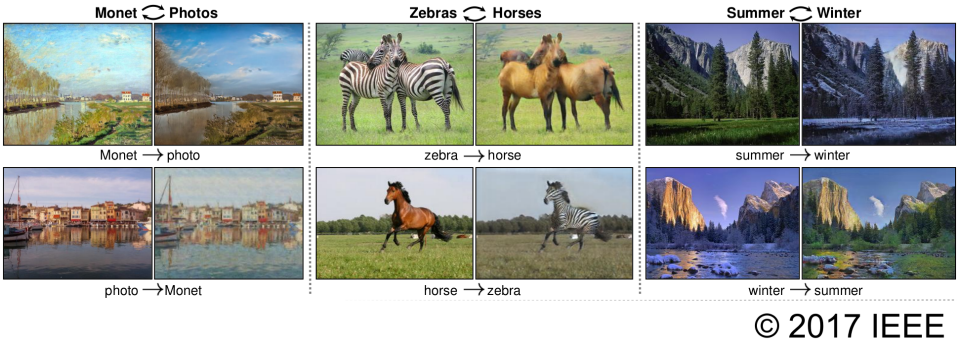

Figure 4: Some results of Zhu et al. (2017)

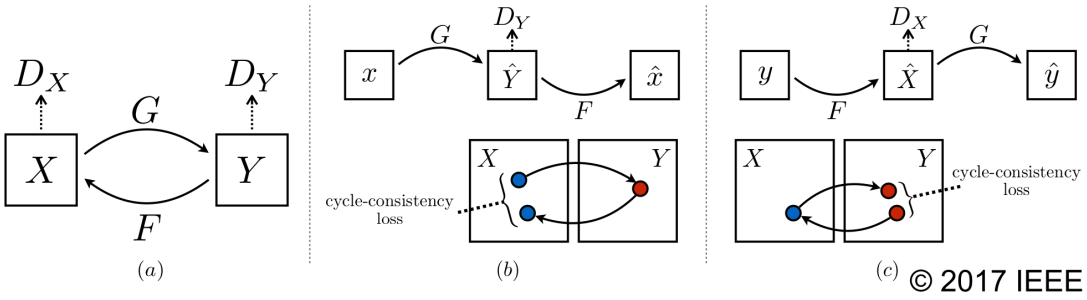

Figure 5: The adversarial scheme (a), forward cycle consistency (b), and reverse cycle consistency (c) in CycleGAN (figure from Zhu et al. (2017))

There appears to be at least one example of CycleGAN-based adversarial networks in smartphone augmented reality, which is the gender-swapping camera filter of the popular messaging app Snapchat (Snap Inc. 2020), introduced in early 2019 (Harbison 2019). While this is not explicitly known to be GAN-based, observed filter outputs are consistent with those expected from a GAN applied to a rotated crop of the user's face, subsequently pasted back onto the input frame. Similarly to CycleGAN, passing

the filter output through the filter a second time has been observed to result in an image similar to the original, although this has not been robustly proven (Jang 2019). The main caveats of this filter appear to be a lack of temporal consistency, particularly observable in the hair output, followed by truncation of long-haired outputs due to the limited face cropping area in the GAN input, as well as sensitivity to face occlusion or face paint.

Snapchat appears to have at least one other GAN-based face filter, also operating in real-time, that transforms the user's face into an anime-style equivalent (Snap Inc. 2021). The limitations and failure cases are similar to the aforementioned gender-swapping filter. Given the closed-source nature of the application and method, the existence of this GAN filter does not significantly reduce the importance of public research on the topic of low-latency, mobile-friendly generative adversarial networks, whether human face-oriented or otherwise. Furthermore, as competing applications, including that of the placement company, often seek to obtain feature parity with other augmented reality apps, continued research into the topic is also relevant to the industry. The rarity of mobile applications using real-time image-based GANs (Snapchat is the only one that could be identified as of September 2021) further proves this point.

Even for desktop and non-realtime applications, the original CycleGAN has some notable issues. Firstly, it has the tendency to modify images outside the target area when doing localized changes. An example of this can be observed in Figure 4, as the horse-to-zebra transform often changes the color of the grass. The simplest way to combat this is through a segmentation network applied to the output of a CycleGAN-based network. However, this requires a well-trained custom segmentation network, so most region-aware GAN approaches instead use segmentation masks at training time, while not requiring them during inference. For GANs with the purpose of face editing, the CelebAMask-HQ dataset (Lee et al. 2020) is often used, as it contains segmentation maps for high-quality aligned face images (512×512 resolution). Examples of GAN projects using this dataset are MaskGAN (Lee et al. 2020) and SPADE-TensorFlow (Kim 2021), which allowed for the editing and generation, respectively, of images following a user-defined or manipulated segmentation map. This allowed for controllable outputs, like the addition of glasses, or changing the length of the hair in the input.

2.6.2 GANs for makeup style transfer

While it is common for makeup apps to allow predefined or user-configurable makeup styles to be tracked to a user's face (PerfectCorp 2021; Perfect365 2021), a makeup style might also be defined by an image of a face wearing it. Since this requires less

effort on the user's part to configure and preview, several works have attempted to implement it.

While later methods used GAN methods similar to CycleGAN for makeup style transfer (Liu et al. 2016a; Li et al. 2018b), Guo and Sim (2009) relied on gradient-domain editing (Pérez et al. 2003), given the face skin region. Unlike previous methods such as Tong et al. (2007), it did not require both a before and after image (denoting the same face with and without the target makeup style, respectively).

The first method to use machine learning for the purpose of makeup style transfer was Liu et al. (2016a), based on the image style transfer work in Gatys et al. (2015) (which predated the cycle-consistency tactic of CycleGAN). To prevent differences between input and output in regions without makeup, Liu et al. (2016a) restricted the effect to the makeup regions (eyes, lips, and the rest of the face), using different configurations for each.

A further example is BeautyGAN (Li et al. 2018b), which during training uses an input face mask defining the target areas to apply makeup to (lips, face, area around eyes), and histogram loss to enforce those areas to match the target color distribution in those same areas. The other regions of the image are therefore left unaffected.

The method in Zhang et al. (2019) disentangled facial appearance from makeup style, by creating separated encoders and encodings for each. Using vector arthmetic with the makeup encodings enabled the combination of multiple styles in the output, as well as making a style lighter or heavier.

The approach in Li et al. (2020) is notable for the use of UV loss to ensure dense correspondence between points on faces, so that the makeup is correctly placed and its texture properly retained. This technique is noted as a potential future step for our research in Subsection 4.6.4. However, the overall method of the work requires makeup and non-makeup pictures of the same person in the training dataset.

Given the results and the availability of code, BeautyGAN was chosen as the base of the research carried out to enable makeup style transfer on smartphones, presented in Chapter 4. In its original form, made available in Jiang (2021), the model was too large for sufficiently quick operation in our tests (see Section 4.3). This occurred even when single-image inference was the target, as opposed to real-time frame-by-frame inference. Furthermore, due to the instance normalization layers present in this and many other generative adversarial networks, the models could only be run on the CPU in TensorFlow Lite, not on the GPU, regardless of the GPU delegate used (OpenGL, OpenCL or Metal). This type of normalization requires full precise calculation at inference time as well as training time, as opposed to the technique used for batch normalization (Ulyanov et al. 2016). This, by contrast, uses the running mean of

the average and standard deviation, calculated during training, to approximate batch normalization at inference time, which significantly increases inference speed. Since calculating the mean and standard deviation of the tensor values are atomic operations, instance normalization is not easily implemented on the GPU with low latency, which probably explains its non-existence in the available TensorFlow Lite GPU ops. Instance normalization is not easily replaced with other lower-latency normalization types, since image stylization-oriented generative adversarial networks have been shown to perform significantly worse without instance normalization (Ulyanov et al. 2016).

Even when BeautyGAN was tested without instance normalization purely as a latency test, it still proved unsufficient for suitable GPU inference, as shown in Section 4.3. Even on MacOS, the original BeautyGAN network was observed to be quite slow, even when attempting to run it on the GPU after removing these instance normalization layers (Table 9). This version ran even more slowly on an iPhone XS running iOS, and on a OnePlus 6 using Android, it did not run at all. As a result, a partially improved version was developed, using low-latency depthwise separable convolution blocks, as encountered in MobileNetV2 (Subsection 4.3.1). Given the insufficient performance of even the original BeautyGAN network on unseen data, deemed to be partially because of the limitations of using histogram loss, as well as the limited size of the dataset used, the research was terminated before a fully mobile-friendly variant could be developed. These findings are described in more detail in Chapter 4.

2.7 Wrinkle and blemish detection

Detection of face wrinkles has been the subject of a fair amount of research, either for dynamic skin folds (which are required to animate virtual human faces realistically) or static wrinkles (for face age and skin quality estimation). While some techniques for dynamic skin folds (Li et al. 2015; Cao et al. 2015; Huang et al. 2011) were looked into shortly for attempted use for the detection of static wrinkles, particularly the facial performance capture technique in Cao et al. (2015), they generally involved high-resolution 3D and 4D face scans, which were not available to the company. This restricted the research to purely image-based detection algorithms, which could involve either machine learning techniques (Alarifi et al. 2017; Alrabiah et al. 2019), or traditional computer vision methods (Batool and Chellappa 2012a, 2014; Cula et al. 2013), using various filters or line tracing or both. Given the dataset requirements of machine learning approaches, which are difficult to satisfy, particularly for semantic segmentation tasks such as wrinkle and blemish detection, the focus of the research in this thesis was on traditional computer vision techniques. An overview of these for both wrinkles and

blemishes is presented below.

One of the older techniques for detecting wrinkle-like structures is the Frangi filter. While originally used for detection of blood vessels for medical applications, the Frangi filter (Frangi 2001) can be used for any continuous edges, including rivers and wrinkles. An improvement on this was the approach from Cula et al. (2013), which was based on a fingerprint image enhancement algorithm (Lin Hong et al. 1998). After generating a low-resolution direction map roughly following the wrinkle directions, it identified the wrinkles using Gabor filters tuned to these directions. However, this was tested using a special polarized lighting system that greatly improved wrinkle contrast, and would need testing on real-use images to compare with other methods, as mentioned in Ng et al. (2015b). Gabor filters were also used in Batool and Chellappa (2015), by combining multiple such filters set to angles equally spaced over the first two quadrants, and choosing the maximum output for each pixel. It followed this with several steps, including line tracing and double detection removal, to improve this output. This method was built upon in the latter part of the research conducted in Chapter 5 of this thesis. The Hybrid Hessian Filter, introduced in Ng et al. (2015a), was shown to improve on both the Frangi filter and Cula et al. (2013). Hessian Line Tracking (Ng et al. 2015b) was later shown to be better than all three in most conditions, obtaining an accuracy of 84%.

Wrinkle detection can also be used to estimate face age automatically, even without the use of heavy machine learning models. However, wrinkles generally have a higher effect on perceived age rather than real. Aznar-Casanova et al. (2010) analysed the perceived age relative to the number of wrinkles, by varying the wrinkle counts on composite images. It noted factors other than age that could affect wrinkle formation, such as exposure to sunlight and amount of exercise of facial muscles, which separates the concepts of age and perceived face age. More recently, the approach in Ng et al. (2016) uses sequential minimal optimization after detecting the wrinkles using Hessian Line Tracking, in order to obtain the age's relationship to the wrinkles.

For the purposes of the research in Chapter 5, including the limitations of what the methods looked into can detect, blemishes are understood as undesirable skin conditions, characterized by relatively small circular or amorphous color discontinuities, such as acne, warts, skin tags, moles, and blackheads. Normally this is extended to other skin conditions, characterized by larger skin structures, such as more serious cases of hives, psoriasis, and eczema, or general reddened skin. However, large-area skin color variations are not picked up by short-range detection schemes targeting sudden color discontinuities, as were used in Chapter 5 for small blemish detection, and would have required a significantly different approach (Lu et al. 2010; Roy et al. 2019; Muhimmah

et al. 2021). As a result, they were not addressed in this thesis.

For detection of the types of blemishes addressed in Chapter 5, there are many medical dermatological datasets available for detecting skin diseases such as cancer, and differentiating it from benign skin conditions. These classes of blemishes are unsuitable for differentiating benign skin conditions relevant to beauty care (acne, blackheads, moles), but they could be used for generic blemish segmentation. A skin dataset could be tagged in a coarse manner by using rough manually annotated segmentation regions with the blemish classes noted, while our generic blemish detector could provide finer segmentation masks based on these regions.

For most beauty apps, there is a demand for retouching capabilities, for the removal of wrinkles and blemishes. Ideally, the retouching experience must be made as simple as possible for an unexperienced user, while providing enough user control for a satisfactory result. In its simplest form, a retouching process may just detect the skin regions and blur the skin, thereby removing high-frequency discontinuities such as relatively small wrinkles and blemishes. The Perfect365 (Perfect365 2021), PicBeauty (Active Beans Inc. 2021) and Beauty Camera (IJoysoft 2021) apps appear to be examples of this. However, the results are too visibly manipulated, in part due to lacking any replacement high frequency elements of healthy skin texture, such as pores, giving an unrealistic blurry look to the face.

Of the available third party beauty apps identified, the most advanced appears to be YouCam Makeup (PerfectCorp 2021). Aside from the hair coloring effects and virtual try-ons of different makeup styles, it also allows for various face beauty metrics, including wrinkles and blemishes. It further allows for photo retouching to reduce wrinkles, spots, and excessive skin texture.

A superior result such as this can be achieved by sampling the texture of a healthy patch of skin, in order to paint over affected regions, while maintaining a smooth blending between the painted regions and the regions left unchanged. The latter can be achieved through Poisson blending, which combines source elements in the gradient domain for seamless compositing (Pérez et al. 2003). This is the algorithm used by Adobe Photoshop's healing tool, which uses manual sampling of "correct" regions by the artist to cover up the "wrong" ones. In a beauty app, however, the sampling of healthy skin needs to be automatic. On such approach, using Poisson blending combined with automatic segmentation and removal of both wrinkles and blemishes, is Batool and Chellappa (2014). This combined the Gabor filter bank method also used in Batool and Chellappa (2015) with Markov random fields, creating an improved filter giving fewer false positive areas than Gabor filter banks alone. By considering the image as a grid of square patches, each patch containing a region to be removed

Figure 6: Patched image (with wrinkles and blemishes covered) before and after Poisson blending step. Image from Batool and Chellappa (2014).

used the closest "clean" patch as a texture source, and Poisson blending was used to smoothen the resulting image and remove the "checkerboard" aspect (Figure 6).

A large amount of skin blemish analysis is rooted in the medical industry, for detection of skin cancers, and classifying skin marks as dangerous or benign (Hoshyar et al. 2014; Alfed et al. 2015; Jain et al. 2015; Alquran et al. 2017). These approaches generally rely on high-resolution imaging, generally containing a single blemish, and employ machine learning and large dermatology datasets. As such, these methods generally have limited applicability to the purposes of face retouching on a potentialy low-resolution image of a user's entire face.

The approach in Alamdari et al. (2016) addresses acne detection specifically for mobile application use, and therefore unlike many of the previous methods, focuses on low-resolution images with multiple acne lesion zones, which is different from most dermatology datasets. The work compared several different methods for acne segmentation (k-means clustering, texture analysis and HSV colorspace segmentation), as well as classification of normal skin, inflammatory acne and acne scars (fuzzy c-means clustering and support vector machines), achieving the best precision with two-level k-means clustering and fuzzy c-means, respectively. This was however observed on a relatively small dataset of 35 images.

In both the case of wrinkles and blemishes, if the detection of affected skin regions is done through classical machine learning methods, there needs to first be a step to identify the skin regions. In Batool and Chellappa (2014), for example, the skin regions to be corrected were provided by the user through a polygonal selection tool, which would be feasible on a smartphone. In existing apps, like YouCam Makeup (PerfectCorp 2021), this is sometimes handled by asking the user to align their face with a face-shaped graphic overlay on top of the camera feed on the phone. A more user-friendly method would use an automated facial keypoint detector, potentially in combination with the manual alignment method. In this thesis, facial segmentation is already provided as part of the virtual try-on system, and is used in the wrinkle and blemish detection

implementation.

While it initially appears to only be relevant to wrinkle detection, we note the approach used in Elbashir and Yap (2020), based on the vessel-detection scheme in Jerman et al. (2016), both of which detect long structures. This is what the majority of our work on blemish detection is based on, as it is shown to be similarly applicable to detecting circular structures such as most blemishes, using the variant for cross-section of blood vessel detection also described in Jerman et al. (2016), but not previously considered with respect to dermatology. Given the similarity between the blemish and wrinkle detection implementations when using this algorithm, much of the code required for the two is the same, which simplifies code implementation and maintenance.

This is built upon in Chapter 5, along with rough estimation of age and skin quality based on the outputs of this method. As mentioned previously, the wrinkle detection implementation is built upon using a Gabor filter approach similar to the initial step in Batool and Chellappa (2015), initially as a postprocessing step, but subsequently also considered as a single detection step.

2.8 Neural architecture search for segmentation

Neural architecture search (NAS) involves the optimization of the structure of the network and the choices of operations it uses, not only the trainable weights of the layers, as in traditional network training. This seeks to remove the difficult art of manual neural network design, which requires highly specialized work, and may be highly dependant on the target operational hardware. As will be relevant to the research in Chapter 6, neural architecture search can sometimes allow for crafting multiple variants of a model to target different inference-time hardware, so custom versions for multiple smartphones, as well as both CPU and GPU versions, can be created simply by changing parameters in a lookup table used by the search process. The desired end result of neural architecture search is in general primarily high accuracy, combined with lowering the latency and computational requirements of model inference.

Aside from early NAS approaches, which constructed the whole network directly from base layers, a two-hierarchy approach is used in most of the more recent works, formed of a cell-level, and a network-level search. A cell is understood as a group of candidate base blocks, formed of one, more, or even zero operations (to allow the cell to be skipped, as in the FBNet approach in Wu et al. (2019)). On the cell-level search, the optimal block for each cell is chosen. On the network level, the cells and the connections between them are optimized, eventually giving the final network as the result of the search. In some cases, the network level is fixed, as is the case for FBNet

(Wu et al. 2019).

2.8.1 Differentiable architecture search and latency awareness

While many successful neural architecture search schemes use reinforcement learning (Zoph and Le 2017; Tan et al. 2019), these are too computationally intensive to be attempted with the resources available to us. As an example, the CIFAR10 search of Zoph and Le (2017)) required "800 networks being trained on 800 GPUs concurrently at any time".

An alternative which has seen increased popularity is differentiable architecture search, which allows the search process to be optimized in a manner similar to traditional network training. One of the first attempts was DARTS (Liu et al. 2019b), which defines a supernetwork of individual cells, in turn formed of a set of *edge* sub-operations (basic ones like convolution, max pooling etc.) between *nodes* (input/output tensors to sub-operations). The search converges towards an optimal *mixture* of edges, derived through training with a softmax operation on the outputs of the candidate edges, and picks the edges with the largest mixing weight. The best edges in each cell are selected, giving the final full network, which is fine-tuned in a second training run.

The search scheme of DARTS has been brought into question by Wang et al. (2021a), that showed the poor quality of the architectures selected by the DARTS scheme, and offered an improved version based on estimating the importance of operations by removing them from the candidate architecture and measuring the performance impact.

While the lower computational cost of differentiable architecture search is significant for our research, it is even more important to attempt optimizing for latency as well as accuracy, given our target of designing mobile-friendly networks through architecture search. While inference speed is less important for desktop-oriented approaches, many neural architecture search methods do also partially optimize for lower latency or computational requirements. When the target inference device is unknown, this can be accomplished by aiming to lower the number and overall complexity of the network operations, more specifically by reducing the multiply-add or FLOP (floating point operation) count, as in EfficientNet (Tan and Le 2019). The MAdd count is understood as the total number of multiply-add operations during one inference run of the model, while the FLOP count is understood as the total number of floating point operations during one inference run. Evidently, these counts can at best only be a rough proxy for the latency of the model. While they have the benefit of being hardware-agnostic, which may allow them to be the best representation of latency on average, they are not the best estimate when the target device is known, and getting the on-device latencies

of the network components is possible.

The most precise solution to this is to run the converted candidate models directly on the target device, a tactic used for MNasNet (Tan et al. 2019). However, it is difficult to implement even when optimizing for a single device. Conversion has a significant amount of overhead, as does pushing and running the model on device, likely requiring multiple device instances, as was the case for MNasNet. And since it would be common to try to optimize the search over a range of possible target devices, to obtain a "best overall" version for that range, on-device measurement would require multiple instances of each device in the range.

It was observed that even when the latency of candidate models is explicitly and precisely measured on the target devices, differences in the on-device execution environment can result in poorer results. For examples, while MNasNet achieved state of the art latency results for execution on the big CPU core of a Pixel 1 phone, it was slower than a non-NAS MobilenetV2 variant of comparable precision when timed on the GPU of the same phone (based on the Mobilenet_V2_1.0_224 and MnasNet_1.0_192 latencies reported in Google (2021g)).

A better proxy for model latency is used in FBNet (Wu et al. 2019), which allows for latency-aware searching by timing every block of every cell on the target device, and using the sum of the blocks in the currently sampled candidate network as the model latency proxy. The two values were shown to have a good correlation by testing on a large number of randomly sampled networks. Latency awareness aside, the search scheme of FBNet is similar to ProxylessNAS (Cai et al. 2018) and NAS-UNet (Weng et al. 2019). FBNet samples a single candidate network per iteration, and uses the Gumbel-Softmax trick to obtain a differentiable, stochastic approximation to the sampling process, thus allowing the entire search scheme to be differentiable. The supernetwork defining the search space is a linear sequence of cells, formed of a small number of candidate blocks, or a *skip*, which is merely an identity operation, to allow cells unwanted by the search scheme to be removed from the final architecture. Similarly to DARTS, each block in the cell is associated with a θ value, which increases or decreases depending on how optimal its corresponding block is to the overall network accuracy and latency, and defines the probability of the block being sampled in a candidate network on a certain iteration. After the search process is completed, the blocks with the highest theta value are sampled from each cell, and the resulting final network architecture is fine-tuned until convergence.

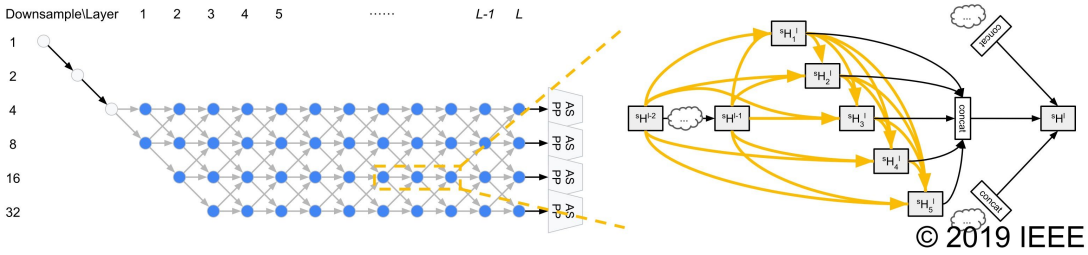

Figure 7: The network-level search space in Auto-DeepLab, shown on the left. Each blue node is a searchable cell, and each possible path through them is a candidate architecture. The right shows the densely connected cell structure, which forms part of the cell-level search space, that tends to a single optimized candidate block for each cell. Figure from Liu et al. 2019a.

2.8.2 Segmentation-specific neural architecture search

In general, neural architecture search approaches focus on classification (Zoph et al. 2018; Real et al. 2019; Liu et al. 2018). However, several approaches focusing explicitly on image segmentation exist, such as Auto-Deeplab (Liu et al. 2019a). This used a generalized search space, which included but was not limited to the DeepLabv3 (Chen et al. 2017a), Conv-Deconv (Noh et al. 2015), and Stacked Hourglass (Newell et al. 2016) architectures as subspaces. As its search method was differentiable, it allowed for a short search time (3 P100 GPU days), comparable to DARTS (Liu et al. 2019b). While keeping the cell-level search of previous NAS approaches, in which different blocks were tested for each node in a larger fixed architecture, it also added a network-level search step. This implies modification of the larger architecture, in this case by optimizing a path between nodes at multiple resolutions, as shown in Figure 7. Each path corresponds to a single network-level architecture. This approach enables Auto-DeepLab to produce several variants with precision similar to DeepLabv3, with varying but significant levels of improved multiply-add counts.

More recently, NAS for segmentation has been the subject of study particularly in the medical domain, generally also by employing differentiable search (Zhu et al. 2019). This often deals with segmentation in three and even four dimensions, for example in tomography scans, which generally require 3D and 4D convolutions. V-NAS (Zhu et al. 2019) tackles 3D segmentation with NAS by making the search choose between 3D, 2D and pseudo-2D convolutions, and improves upon non-NAS architectures such as 3D-UNET (Çiçek et al. 2016).

Basing itself on UNet, the approach in NAS-UNet (Weng et al. 2019) leaves the network-level architecture fixed to a UNet structure, and groups all the operations between two scaling operations (downscaling or upscaling) into a searchable cell, where

the latter scaling operation is considered part of the cell. This gives two types of cells, downscaling (DownSC) and upscaling (UpSC) type. Both cell types are directed acyclic graphs representing an over-parameterized network formed of all the candidate paths, as in DARTS, in which the edges are chosen from multiple candidates during the cell search. The search is differentiable, but differs from DARTS by choosing a single path at each iteration, as in ProxylessNAS (Cai et al. 2018), instead of using a weighted sum of all of them. This is equivalent to a one-hot-weighted sum, and has the benefit of significantly reducing memory use during training. This method was shown to improve on existing segmentation methods on medical segmentation datasets.

Several works have been identified that train specifically for segmentation, and also use on-device latency to guide the architecture search. These are known as *proxyless neural architecture searches*, as they do not search on a proxy task such as classification, and then simply re-use the architecure with an added segmentation tail (like the segmentation MobileNetV3 variant from Howard et al. (2019)). SqueezeNAS (Shaw et al. 2019) is such a method, similar to FBNet in its latency-aware architecture search. A further similarity is its measurement of candidate cell latencies for each searched block, and using the overall sum of these to inform the search. The main difference lies in the segmentation decoder, which is fixed (not searched) to ASPP (*Atrous Spatial Pyramid Pooling*), and LRASPP (*Lite Reduced Atrous Spatial Pyramid Pooling*), in separate versions of SqueezeNAS. A fixed-decoder approach is also leveraged in the adaptation of FBNet to segmentation, in Chapter 6.

While an Nvidia Xavier (a Jetson-family GPU for embedded applications) is the target of the SqueezeNAS search in the paper, the depthwise-convolution-based candidate cells it uses makes application to smartphones feasible, requiring only a smartphone-tuned latency table. Despite this, in our research, due to its similarity to one of the FBNet-based approaches studied, as well as its implementation in PyTorch being a barrier to TensorFlow Lite deployment, an implementation of SqueezeNAS was not attempted.

As a suitable segmentation-specific and mobile-friendly approach was not immediately found, an attempt was instead made in this thesis to adapt an existing on-device latency-aware classification method (FBNet) for lip segmentation. This used several variants with multiple input sizes, and several different supernet architectures of different lengths, both with a fixed (unsearched) and searched decoder. It was the eventual intention, if this approach was successful, to also attempt this for other networks used by the company, like the coarse full-face segmentation and keypoint model, the eye segmentation networks, and a newer version of the lip model that also provided keypoints. However, while the latencies of the resulting lip segmentation networks were

smaller or comparable to that of the non-NAS network, none of them outperformed it in accuracy. These FBNet-based experiments are detailed in 6.

2.8.3 State-of-the-art and emerging methods

The advent of differentiable neural architecture search seems to have led to an increase in segmentation approaches through neural architecture search at the cutting edge, some of which also focus on latency awareness. We note several approaches that appeared or we discovered after and during our work with FBNet, thus being too recent to be tackled in our efforts, or too difficult to implement given our needs.

A more recent approach than FBNet, which does not leave the network-level architecture fixed, is FasterSeg (Chen et al. 2019). This proposes an improved network-level search algorithm, which incorporated multiresolution branches, instead of using a fixed network-level backbone like FBNet. Furthermore, it introduces a new latency regularization method, which corrects the claimed issue of latency-aware NAS methods collapsing to low-latency but low-precision architectures.

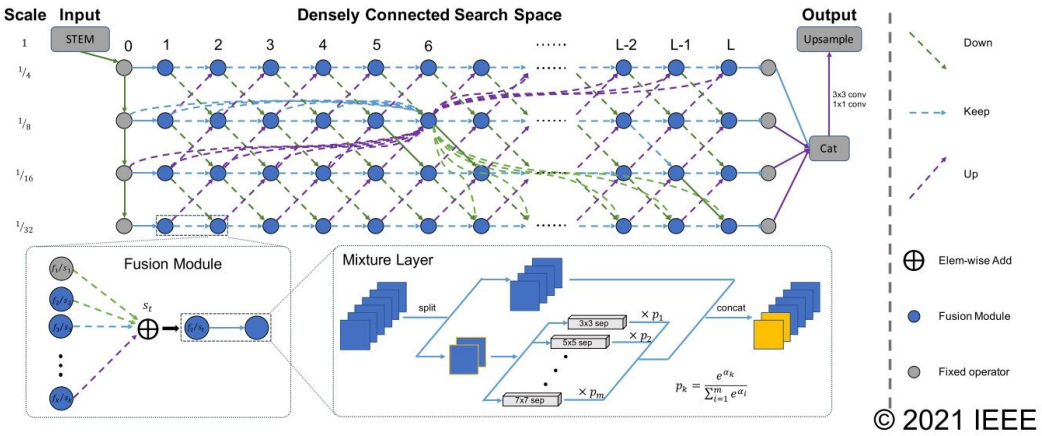

Figure 8: The search space and searching scheme of DCNAS. Image from Zhang et al. (2021).

The block-domain search of FasterSeg introduces a candidate block called a *zoomed convolution*, formed of a $\times 2$ bilinear downsampling, one or two normal convolutions, and an $\times 2$ bilinear upsampling. This is shown by the original authors to be preferable to MobileNetV2-style depthwise separable convolutions, but only on the work's target platform of an Nvidia 1080Ti running TensorRT. This must also be verified for smartphones to be considered for purposes similar to those in this thesis.

FasterSeg was given some consideration for use for the same lip segmentation model as our FBNet segmentation approach, but these did not go beyond an attempt

to adapt it for the lip dataset, given implementation difficulties and time contraints. This is detailed further in Section 6.3.

Another recent relevant work is DCNAS, or Densely Connected Neural Architecture Search (Zhang et al. 2021), which aimed to increase the search space compared to previous works, while continuing to use differentiable search. This is particularly relevant for image prediction tasks, which require representations at multiple scales (although for lower-resolution input networks for real-time execution such as our own, fewer resolution scales are required). Therefore, DCNAS uses dense connections over multiple scales as candidates for the final architecture, as given by the search space illustrated in Figure 8.

Chapter 3

6DoF Object Pose Estimation on Smartphones

Object tracking and pose estimation is a common problem in augmented reality and computer vision. It can be used to offer information to the user about relevant objects (like in a military targeting system), to allow a computer to interact with objects in the environment (as in autonomous driving applications), or to allow virtual and real objects to interact in a believable way. In this chapter, an example of the first case is addressed.

In this project the object to be tracked was a handheld hair curler, which needed full six degree of freedom pose to be inferred in real-time on mobile devices (Android and iOS). Aside from the difficulty of these latency and target hardware requirements, the tracking also required robustness to occlusions by the hand and hair. While occlusions are a common issue in object detection, they are particularly significant when tracking handheld objects, as the hand is almost always partially occluding the object.

In this chapter, we present an approach for real-time 6DoF pose tracking for objects in monocular RGB video. By using a modified set of mobile-specific convolutional neural networks, as well as temporal coherence, we were able to achieve real-time performance on high and mid-range smartphone GPUs (Android and iOS), without low-level optimizations like custom GPU operations. We show how tracking a moving region of interest, a technique previously used for real-time face and hand detection (Bazarevsky et al. 2019; Google 2020b), can drastically improve the performance of 6DoF pose estimations with machine learning. The tracking method can be applied to pose detection for both handheld and non-handheld objects, as well as other generic AR applications, such as semantic segmentation of small objects. While we targeted TensorFlow Lite inference of the machine learning model in Unity, the tracking system is applicable to

any augmented reality platform, on both smartphones and desktop computers.

3.1 Motivation and value of this research

Aside from being required by the placement company for a commissioned app, this was the first project to use our integration of TensorFlow Lite into Unity, and therefore the first proper test of this plugin. It further gave insight into what improvements could and should be done to improve the system, and what needed to be implemented on the Unity side to properly support it on all required devices.

Furthermore, it determined the basic architecture and blocks that would be used for many of the company's machine learning models going forward, including those required for the projects in this thesis. This mainly consisted of MobileNetV2 and MobileNetV3-based models, containing residual blocks with depthwise separable convolutions for faster execution. The sequential use of multiple models, as used in the tracking form of the 6DoF object pose detector, was also used in other company projects, as well as the use of temporal coherence to reduce jitter in outputs in subsequent video feed frames.

With regards to the value to the greater scientific community, the resulting 6DoF object pose tracking system can be used for any object, handheld or not, which broadens the applicability of the method. On smartphones, it could be used for various augmented reality experiences and games, for example to overlay textures or particles onto real objects, to allow virtual objects or agents to interact with real objects, or to replace real objects with virtual ones.

3.2 Commercial context of this research

The goal for the placement company, as required by another client company that commissioned the project, was to create a smartphone app for the education of users in the proper use of a new type of hair curler, whose use differed from existing curler models. For example, for a beach wave curl, the new curler needed to be be clamped onto a lock of hair close to the root, while being held diagonally relative to the direction of the hair, and slowly pulled down the lock of hair. This would give the lock of hair a helical shape. By contrast, to achieve the same style with a traditional hair curler, the lock must be wrapped around the curler in a helical shape, and maintained there for an extended period of time.

The client company observed during focus group testing that these differences caused users to employ the incorrect, traditional ways of use, without attempting to

follow the instructions available in the box or online. It was therefore the hope of the client company that an app associated with the curler might inspire users to follow the instructions. They also sought to gauge how well the users were performing, which required six degree of freedom (3D location and 3D rotation) pose detection of the curler, leading to the machine learning efforts in this chapter. The app included other elements such as presentation of new styles, marketing integration, and various gamification and reward systems, but these were outside of the scope of this thesis.

For the placement company, this was an initial beauty project, which led to a singular focus on beauty applications. As a result, the work in this chapter is somewhat separate from the subsequent projects in the thesis, as these all focus specifically on the face of the user. This is due to their reliance on a face tracking system developed after the curler project. Nevertheless, as mentioned previously, it gave the technical, software, and design basis of later computer vision with machine learning work in this thesis and at the placement company.

3.3 State of the art in 6DoF object pose detection

As specified in Chapter 2, at the time of the beginning of this project (late 2018), there was no existing 6DoF pose detection work using neural networks that specifically targeted smartphones, only desktop computers and higher-power mobile GPUs (like Nvidia Jetsons). Therefore, several state-of-the-art (SOTA) desktop-oriented networks were looked into for adaptation for smartphones. Due to the effort required to create the pose datasets and fully implement the curler pose detection system in Unity, the project needed to start from an existing code implementation if possible. Furthermore, the code implementation needed to have a permissive license, as the placement company aimed to use the resulting network for commercial purposes.

The more relevant SOTA pose detection approaches identified were single-shot-pose (Tekin et al. 2018), PVNet (Peng et al. 2018), and segmentation-driven-pose (Hu et al. 2019), all of which have previously been discussed in Chapter 2. Single-shot-pose uses a variant of the YOLOv2 2D bounding box object detection network, modified to output the positions of the 3D bounding box corners, from which the pose can be extracted using perspective-and-point. Both PVNet and segmentation-driven-pose generate a coarse (low-resolution) segmentation map of the detected object, each pixel of which gives the positions of the 3D bounding box points. All these outputs are reduced to an overall pose using RANSAC-based perspective-and-point. Both of these methods have been shown to be more resistant to occlusions and truncations, due to the segmentation element, which should make them more suited for the hand-occluded

objects like the curler. However, none of the two had sufficiently permissive licenses, so the single-shot-pose was the main method used. Still, the segmentation-driven-pose method was still tested, but gave inferior results to single-shot-pose in our attempts, as shown in Section 3.9.

For the purposes of our implementation, a dataset of images of the tracked object with associated pose data needed to be collected. This was achieved through two methods, listed in Section 3.4 and Section 3.5.

3.4 Data collection with manual and interpolated annotation

This involved an annotation system created in Unity, in which the CAD model of the curler was manually posed to align with frames of the video feed. To reduce the amount of manual annotation, an optical flow algorithm based on Choi and Christensen (2012b) was used to automatically set the pose between nearby manually annotated frames. This method was specifically designed for tracking textureless objects, which suited the curler well, as it was a matte black object, and could not be redesigned by the client company to have more easily trackable features. This led to a dataset of about 19000 frames total. Some example images are shown in Figure 9.

The method cannot be described in detail, as it was not implemented by the author of this thesis.

3.5 Automatic data collection with a VIVE VR system

As manual annotation was time-consuming, a new automatic pose data collection technique was employed, using an alternative VIVE-based setup, illustrated in Figure 10. In this figure, the VIVE tracker t_c rigidly attached to the curler c gives the curler's pose $P_{c,v}$ in the VIVE system space of origin v, while a second VIVE tracker attached to a calibration pattern at a known point gives the calibration pattern's pose $P_{p,v}$ in this space. Since the pose of the calibration pattern in the camera space $P_{p,cam}$ can be retrieved through perspective-and-point, the camera-relative pose $P_{c,cam}$ of the curler can be calculated as in Equation (3.1). While this could in theory also be achieved with a cube with ArUco markers on its surface, tracking would be more easily lost. Furthermore, the cube would need to always be in the line of sight of the camera, but would often be below it in the training data, if attached at the same point as the curler tracker.

$$P_{c,cam} = P_{c,t_c} P_{t_c,v} P_{t_p,v}^{-1} P_{p,t_p}^{-1} P_{p,cam} \tag{3.1}$$

Figure 9: Samples from the real dataset, using the manual and semi-automatic annotation in Unity.

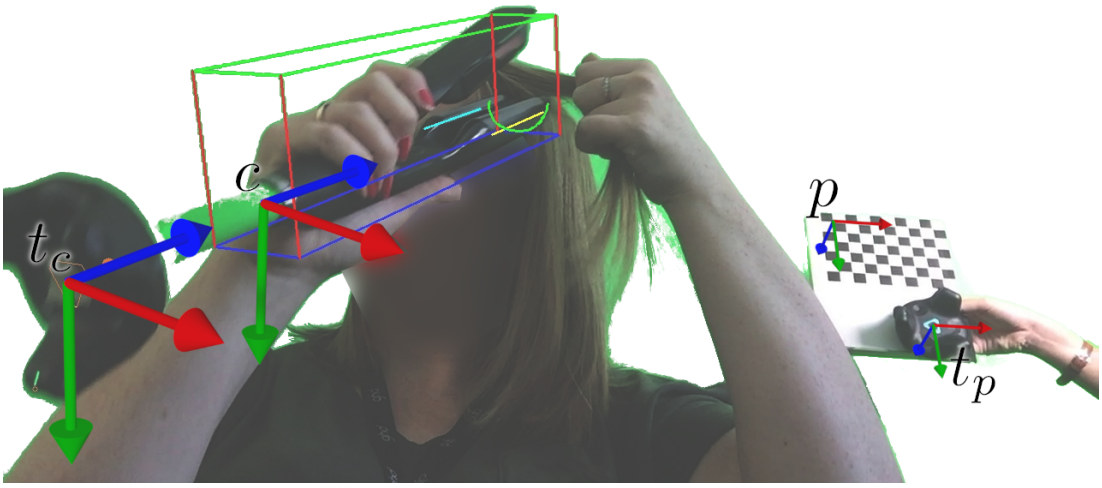

Figure 10: The VIVE setup. Not shown are v (the VIVE space origin) and *cam* (the camera origin). The relative poses between calibration pattern and pattern VIVE tracker, as well as between curler origin and curler-attached tracker, are retrieved through manual alignment.

To avoid overfitting on the object-and-VIVE-attachment system, the VIVE attachment was masked out using the tracking data, by applying the pose data to a rough 3D mesh of the attachment system in Blender, and rendering it as a segmentation mask. This mask was treated as a "hole" in the image, and filled using a patch-based method in a batch Adobe Photoshop operation.

For this task, we had previously tried a feature in Adobe After Effects that enabled the re-filling the "hole" left after an unwanted object (like out curler attachment) was masked out (a technique known as *inpainting*). This is a machine learning-based feature called *content-aware fill*, and allows for temporally smooth, believable inpainting. However, for our task of removing the attachment from a large number of frames, it proved to be too unreliable and hard to automate.

This automated VIVE annotation method has the caveat that the curler is greatly occluded by the attachment when the base faces the camera. Therefore, some manual annotation with the VIVE attachment removed is needed to cover these edge cases.

Most of the training images (about 47000 in total) were taken against a green-screen, to allow for augmentation with random backgrounds from the Pascal VOC dataset (Everingham et al. 2010).

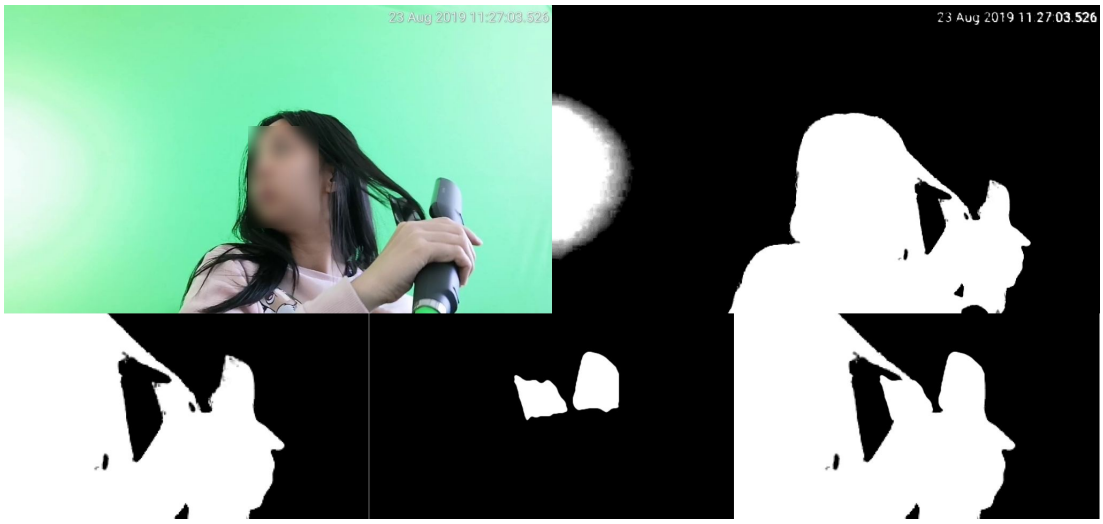

Figure 11: Segmentation mask edges when using a bad greenscreen that requires the Rotobrush and/or aggressive keying with a sharp cutoff. Top left shows the original image, top right is the clipping mask if using only greenscreen keying. Bottom left shows the roughness of this mask around the curler. Bottom center shows the regions that require manual Rotobrushing - the large clamp of the curler, due to the reflective component, and the small clamp, due to the roughness of the keyed mask. Bottom right shows the region in the combined final mask, which also requires a manual polygonal mask to remove the spot on the left and the timecode in the full mask (top right).

3.5.1 Background removal and randomization

For both the manually annotated and VIVE-annotated footage, most of the source videos were greenscreened with at least partial background coverage. In general, this was removed by using the Keylight and Linear Color Key effects in After Effects. Additional irrelevant background elements (like the checkerboard calibration pattern) were able to be removed with varying degrees of success, using manual masking.

During the new VIVE-aided data collection, in order to increase the robustness of trained networks to extreme lighting conditions, bright lights were shone on the subjects for part of the ground truth footage. This had the unfortunate effect of also brightening sections of the green screen, complicating the color keying process, and leading to frame-by-frame Rotobrush effect use and manual masking in many cases. This also led to foreground elements having rough edges. The silvery elements of the curler also tended to reflect the greenscreen, requiring manual Rotobrushing, although this would be required regardless of the quality of the greenscreen. These issues are illustrated in Figure 11.

This background removal was used to create grayscale foreground masks, which were used during network training to composite the foreground on top of scaled images

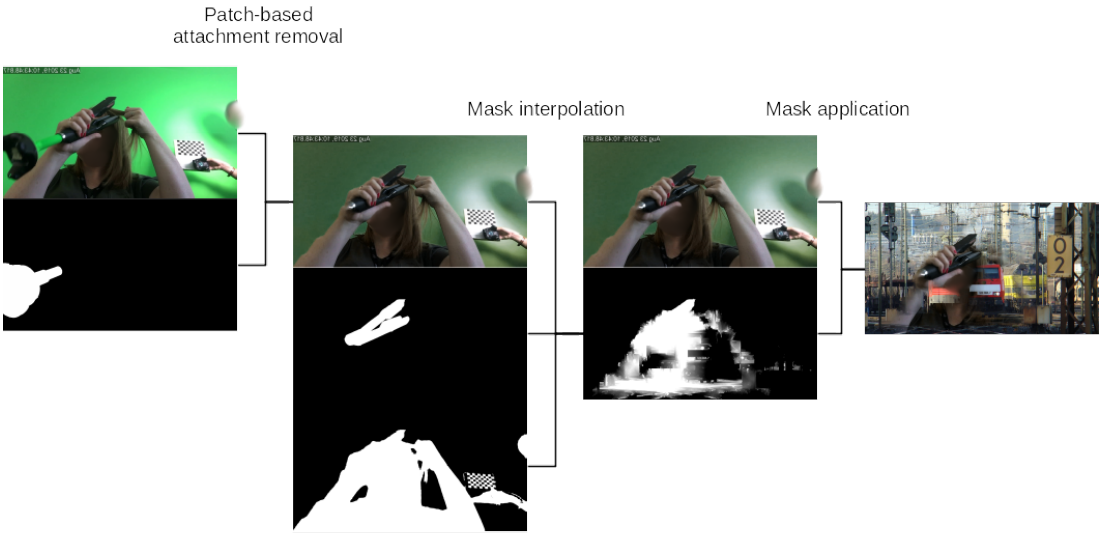

Figure 12: The processing steps used to create the curler dataset. Initially, the patch-based filling mechanism removes the unwanted attachment, and the resulting image is color corrected with green-screen despilling, through which reflected light from the green screen is muted. The resulting image, as well as the two masks, are used to retrieve the interpolated mask. The image and background are then blended using this mask. It can be observed that the foreground has higher opacity near the hand holding the curler. Background in right image from Pascal VOC dataset (Everingham et al. 2010).

from the Pascal VOC 2012 dataset (Everingham et al. 2010). This was done as a preprocessing step during training, in order to make the trained network more robust to new backgrounds at test time.

For further diversity in augmentation, an interpolation scheme using a color difference-weighted distance function was used for the foreground mask at training time. Through this scheme, the mask varied between including just the curler, and the entire subject. This interpolation was designed to favor including the hand, as it gives localization features that will still be present and relevant at test time, while the rest of the human features are less relevant, and are more likely to vary during real-life use.

The complete preprocessing pipeline is illustrated in Figure 12.

In our case, it was initially also a requirement that the network verify whether the curler was open or closed. Two different bounding boxes were used for the open and closed case. It was observed that the open/closed status was more accurately retrieved by choosing the bounding box that gave the minimum reprojection error, rather than treating the open and closed curler as two different classes and using the highest class

confidence. We note that training with the same bounding box for both open and closed would likely have decreased the reprojection error, giving better overall results.

3.5.2 Generalizability of the data collection method

Both here and in general, the collection of ground truth is often hampered by the need to match the *in the wild* (real-life use) domain, since the network may learn features specific only to the training dataset. This generally prevents the collection of ground truth pose using object-attached markers, or retrieving face pose using facial tracking markers, as the network will generally pick them up as features.

If in a marker-based ground truth collection scheme, all markers can be detected, and are not too large, then a method similar to ours could be used to obtain pose data automatically, by getting the approximate marker segmentation masks using the pose data, and filling the resulting "holes" believably. This could allow for similar automatic pose data retrieval, significantly reducing dataset creation time and effort.

3.6 Underlying work for 6DoF pose network architecture

The structure of the model used for our pose detection project draws from a desktop computer-oriented 6DoF pose detector known as single-shot-pose (Tekin et al. 2018), and the mobile-oriented feature extractor of MobileNetV2 (Sandler et al. 2018).

The single-shot-pose (YOLO6D) network is illustrated in Figure 13. YOLO6D is based on the YOLOv2 object detection network (Redmon and Farhadi 2017), using the same Darknet-19 feature extractor and grid-based detection, but modified to output 9 value pairs per object hypothesis, corresponding to the 2D projections of the object centroid and the corners of its 3D bounding box, instead of the 2 value pairs defining the bounding box position and size relative to the upper-left grid corner, as in YOLOv2.

In the general YOLOv2 2D bounding box object detection network, given an RGB image of dimensions (X, Y) as input, the output is a $(W, H, A \cdot (5 + C))$ grid of cells, where the cells are of identical size (W_C, H_C). Here, A is the number of anchors of the output, and C is the number of object classes to be detected (Redmon et al. 2016; Redmon and Farhadi 2017). An *anchor* is a box of predefined width and height relative to the cell, which shares its center point (X_{A_i}, Y_{A_i}) with that of the cell. For each anchor A_i of width W_{A_i} and height H_{A_i}, a confidence value for each class is predicted (the certainty that an object of that class is in this anchor box), an overall confidence value for the anchor box itself (indicating the likelihood that some object is in that box), and four values (w_i, h_i, x_i, y_i) such that

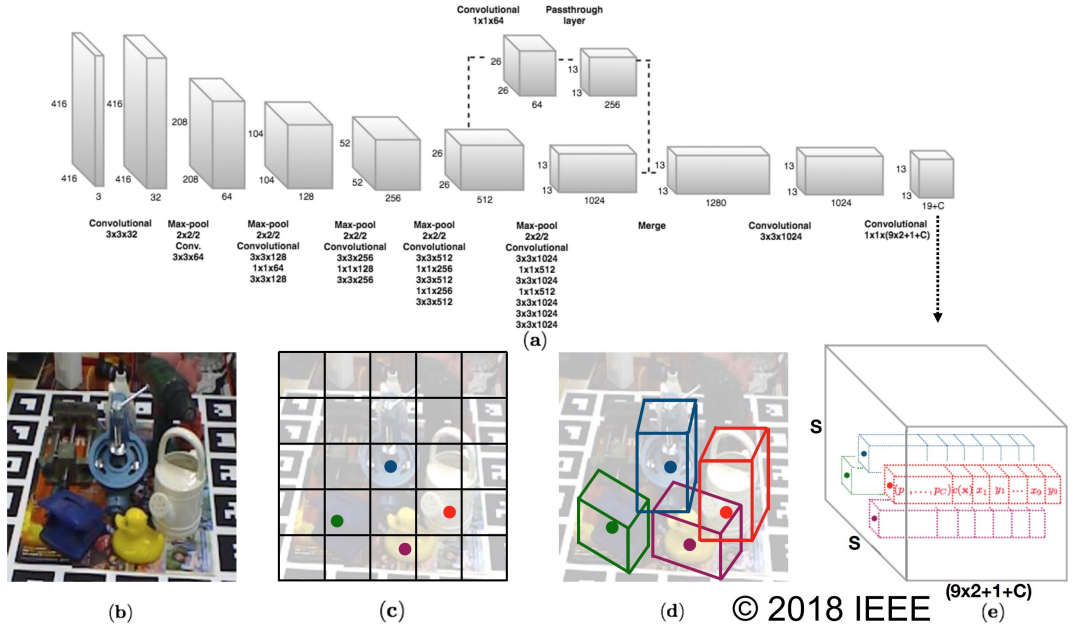

Figure 13: The single-shot-pose network, with the architecture shown in (a). For an input image (b), the output (e) is a set of $S \times S$ cells, containing the overall detection probability, the (X, Y) position of the object centroid in the cell (c), the (X, Y) positions of the projections of the eight 3D bounding box vertices (relative to the centroid but not necessarily inside the cell) (d), and the class probabilities. Figure from Tekin et al. (2018).

$$\frac{1}{1+e^{-w_i}} \cdot W_{A_i}, \frac{1}{1+e^{-h_i}} \cdot H_{A_i} \tag{3.2}$$

are the predicted dimensions of the bounding box of the detected object, and

$$\left(\frac{1}{1+e^{-x_i}} - 0.5\right) \cdot W_C + X_{A_i}, \left(\frac{1}{1+e^{-y_i}} - 0.5\right) \cdot H_C + Y_{A_i} \tag{3.3}$$

are the xy-coordinates of the bounding box.

Unlike the original YOLO (YOLOv1), which has a fully connected layer at the end, YOLOv2 is fully convolutional, and can be adapted to any input size multiple of 32, both during training and inference. The single-shot-pose network modified the YOLOv2 output structure by using a single anchor of size $(1.0, 1.0)$, and only limiting the object centroid to the anchor area by means of the sigmoid function, while outputting the raw cell-relative positions of the eight 3D bounding box corners directly (as in Figure 13c,d).

While YOLOv1 was simply a sequential model (with no branching), YOLOv2

and single-shot-pose have one branching close to the end of the feature extractor, as in Figure 13a. This is not a skip connection, as it has one convolution on the secondary branch before the two are merged through the *reorg* layer, specific to the DarkNet framework. Despite this, it is mentioned in the YOLOv2 paper (Redmon and Farhadi 2017) that this change is a response to the noted improvements brought by the skip connections in residual networks.

3.7 Adaptation of single-shot-pose for mobile platforms

To lower the latency and computational requirements of the single-shot-pose network, two new network architectures were created, both involving the replacement of the Darknet-19 feature extractor with a more lightweight backbone.

The first architecture replaced Darknet-19 with the feature extractor of the tinyYOLOv2 network (Redmon and Farhadi 2017), which has fewer operations (32.01 million per output cell instead of YOLOv2's 174.3 million per output cell). This was done partially due to the tinyYOLOV2 network being part of the same Darknet framework as Darknet19, therefore simplifying backbone replacement by editing the configuration files. TinyYOLOV2 is a relatively simple feature extractor, formed of a linear sequence of convolutions.

The second architecture instead used the MobileNetV2 backbone, which has better latency on smartphones than both Darknet-19 and tinyYOLOv2, as shown in Table 2, as well as the numerous architecture improvements shown in Subsection 2.3.1.

For both networks, variants for both 224×224 and 512×512 input sizes were trained, to verify the effect on accuracy, and use the smaller input size if possible, for even faster smartphone inference.

To test the latency of the resulting networks on-device, and verify that they worked with the Unity and TensorFlow Lite system described in Chapter 2.4, the resulting model needed to be converted from its PyTorch-readable format to the TensorFlow Lite .tflite format. Given that TensorFlow to TensorFlow Lite conversion can be done automatically by TensorFlow, the issue was reduced to PyTorch to TensorFlow conversion.

3.7.1 Attempts at automatic machine learning model conversion

In order to simplify migration between machine learning frameworks, the *.onnx* format was introduced by the Open Neural Network Exchange (The Linux Foundation 2021). This aimed to offer a standardized representation of neural networks, working as an intermediate step during conversion to and from various machine learning frameworks.

As a result, we attempted to use it for automatic conversion from PyTorch to TensorFlow, for subsequent conversion to TensorFlow Lite. For our use case, this ultimately failed due to improper support for depthwise convolutions, and lack of support for channel-last (NHWC) tensor axis ordering.

The use of depthwise convolutions in the curler pose models resulted in issues when attempting automatic conversion from PyTorch to TensorFlow via ONNX. The initial PyTorch-ONNX conversion was successful, with the ONNX network structure appearing as expected when studied in Netron, a neural network visualization tool (Roeder 2021). However, converting from ONNX to TensorFlow caused the depthwise convolutions to be broken into a large amount of operations. These were formed of a channel-wise splitting of the input tensor, followed by a regular convolution of each split using a separate copy of the original depthwise convolution weights, and the concatenation of the outputs. While these are technically equivalent to depthwise convolutions, and the network output did match the PyTorch model, this resulted in a very large increase in the latency of the resulting TensorFlow Lite model on both the CPU and GPU, too large for it to be usable on mobile devices. This appears to be a result of TensorFlow not officially supporting ONNX, even though support for it is built into PyTorch. As a result, automatic conversion with ONNX did not work for our purposes.

We note that due to channel ordering differences between PyTorch and TensorFlow Lite, ONNX would not have worked for conversion even in the absence of depthwise convolutions. This is due to the differing NHWC and NCHW formats, referring to the axis ordering of the input, output, and intermediate tensors in memory. The two tensor formats are illustrated in Equations (3.4) and (3.5), showing how the memory offsets are affected for a tensor element in batch b, at y-coordinate h, x-coordinate w, and channel coordinate c.

$$offset_{NHWC}(n, h, w, c) = nHWC + hWC + wC + c \tag{3.4}$$

$$offset_{NCHW}(n, h, w, c) = nCHW + cHW + hW + w \tag{3.5}$$

PyTorch did not support channels-last (NHWC) tensors at the time of this experiment, and it is unclear if they have been implemented properly even as of April 2021. Inversely, TensorFlow Lite only supports channels-last (NHWC). Even though TensorFlow itself supports both NHWC and NCHW (channels-first), it cannot automatically re-arrange the weight channel ordering during conversion to TensorFlow Lite, making a successful NCHW PyTorch-ONNX-TensorFlow conversion useless for this purpose.

We concluded that ONNX should not be used for PyTorch to TensorFlow Lite

conversion, at least at the time of testing (December 2019). As such, manual conversion from PyTorch to TensorFlow was carried out, by fully re-creating the network structures in TensorFlow, and channel-shuffling the weights of the PyTorch layers and copying them to the TensorFlow layers.

3.7.2 Initial results

For both networks, results on the test set were generally good, with the 5-pixel accuracy (the fraction of bounding box coordinates within 5 pixels of the ground truth) being around 90% for both networks, for both the 224×224 and 512×512 input sizes. However, the precision of both networks proved insufficient during testing on unseen data, including on-device testing. The output points did not form a proper 3D bounding box, making it impossible to extract the pose through perspective-and-point. The points were only sufficiently precise to give the rough 2D positions of the two ends and the centroid of the curler, which effectively reduced it to a 2D pose estimator.

Loosely based on the work of Suwajanakorn et al. (2018), we attempted to achieve better precision by increasing the number of points predicted, and placing the points on the surface of the curler itself rather than at the corners of the bounding box. This would cause the points to be closer to the cell outputting their positions, in theory reducing the error. While this method failed to improve precision, this is believed to have been caused by coding errors, so the results are deemed inconclusive.

3.8 Improved pose detection scheme using tracking

As a further attempt to increase accuracy of the pose detection, a tracking scheme was used, consisting of two architecturally identical networks, but with differing input sizes and training parameters. The first is the detection network, with an input size of $X \times Y = 384 \times 224$, to roughly match the landscape 16x9 aspect ratio required during use. It is trained on the entire input frame, with more aggressive random scaling and XY offset (*jitter*) augmentations. The second is the tracking network, which is trained on a square crop around the 9 keypoints of the 3D bounding box (8 corners and the object's 3D centroid). Jitter and scaling augmentations were done within closer limits on this second network. Given its improved latency and structure, only the MobileNetV2-6D architecture was tested in this scheme.

The tracking scheme follows a simplified version of that in BlazeFace (Bazarevsky et al. 2019). This uses the detection network once and the tracking network on every subsequent frame, the latter running on a crop around the previously detected keypoints.

In the BlazeFace scheme, the object is first detected in the whole frame with an object detection network. Here, we use the 16x9 version of the same model trained on uncropped frames, but this could be replaced with any sufficiently small and fast bounding box detector, such as MobileNetV2-SSD. After the detection stage, a 224×224 square crop of the frame encompassing the 2D/3D bounding box (with additional padding to account for movement) is used as the input of the tracking network. This is then repeated, but using the tracking detector outputs to get the next input crop region. If the tracked detection confidence drops below a threshold, the full-frame detection network is run again to re-detect the object.

The crop region selection could be improved by fitting multiple previous tracking detections to prognosticate the future position. This would improve detection at low framerates or during more rapid motion. For our purposes, this was not necessary, as the tracked object was always moved slowly during use.

3.8.1 Results using the tracking scheme

The system was trained and used to track handheld curlers from an RGB video stream, with both the detection and tracking networks being trained for 100 epochs. The results on isolated frames and frame crops (done without tracking on video) are summarized in Table 1, and some visual results are shown in Figure 14.

Table 1: Average per-keypoint precision and orientation error after perspective-and-point, given as angular offsets from the ground-truth axes. The 224×224 input-sized networks are the tracking (square crop) networks, while the 384×224 are the full-frame detection networks. The test-time inference was done with the same augmentations as the training, so values may be lower than during normal *in the wild* use.

Backbone	Input	2D kp. error	X-axis error	Y-axis error	Z-axis error
MobileNetV2	224×224	8.88 px	20.438°	20.758°	21.749°
MobileNetV2	384×224	8.29 px	35.445°	34.924°	29.522°

The results show the increased precision when using the tracking network, as opposed to just the full-frame detection network. In Table 2, we evaluate the latency of these networks on several phones, on both the CPU and GPU, using the Android benchmarking tools offered as part of TensorFlow. Given the additional strain of running the rest of the Unity app, the in-app performance is slightly worse. On iOS, while precise latencies were not recorded, the networks were always faster on an iPhone XS than on the OnePlus 6.

While neither full-frame model can accurately predict pose on unseen data, as shown in the non-tracking tests, it is sufficiently accurate to provide a region of interest

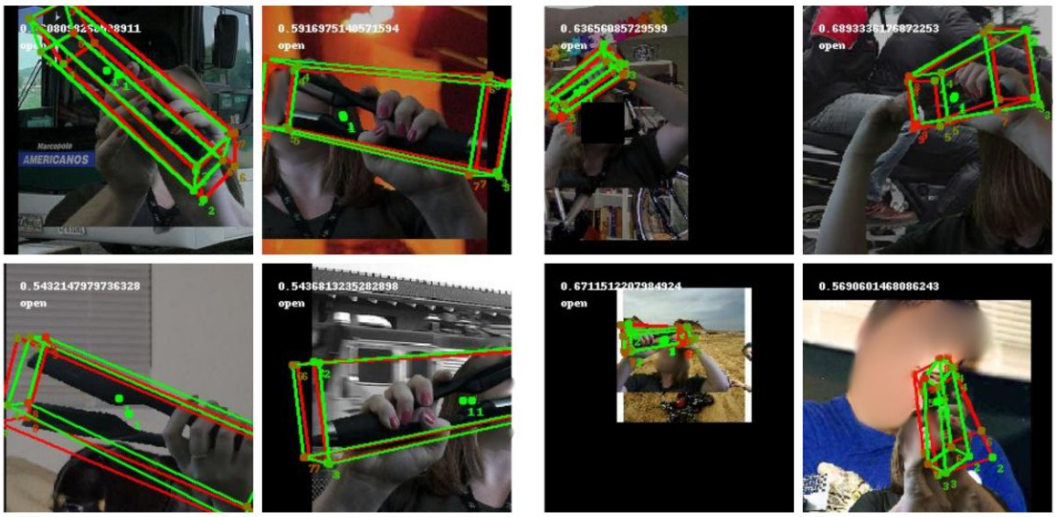

Figure 14: Some evaluation results for the tracking network (left four), and detection network (right four, run in these examples at 224×224 on square inputs). Ground truth is shown in green, prediction in red. Background images are from the Pascal VOC dataset (Everingham et al. 2010).

to be tracked. To demonstrate and evaluate the tracking scenario itself, we use a group of 597 unseen VIVE-annotated frames and isolate frame sequences with some degree of temporal coherence (that is, the time difference between subsequent frames is less than 0.5 seconds). We use a confidence threshold of 0.4 for the tracking network, under which we fall back to the full-frame detection network. The results are shown in Table 3.

In these sequences, the subject was far from the camera, leading to high difficulty in estimating pose by the full-frame network, as shown by the high axis errors. Still, it was generally able to provide a crop region on which the tracking network could provide a reasonably accurate result for our real-life purposes in the app. Except in instances of high occlusion, the tracking network was able to follow the curler with higher-confidence and more accurate predictions, showing the superiority over the non-tracking method in this case.

The failure of the full-frame network in providing pose shows that it can be replaced with a more performant bounding box detector without sacrificing relevant information.

Table 2: The latency of the networks, on a newer OnePlus 6 model (Android P, 2018) and an older Nexus 5x (Android 6, 2015), denoted OP6 and N5x, respectively. The feature extractor of the original single-shot-pose network (Darknet-19) with the Darknet-19 is provided for comparison. The whole network (naturally slower) could not immediately be converted to TensorFlow Lite due to certain nonstandard layers in the Darknet framework. The 224×224 input network is the tracking (square crop) network, while the 384×224 input network is the full-frame detection network. The CPU times use 4 threads, and use 32-bit floating point inference. Tests with post-training quantized models (8-bit integer weights and tensor calculations) ran at roughly 0.65-0.7 the latency of the floating point models. The relative performance gains generally decreased as the models got smaller and faster.

Backbone	Input	OP6 GPU	OP6 CPU	N5x GPU	N5x CPU
MobileNetV2	224×224	10.41 ms	22.70 ms	62.43 ms	87.24 ms
MobileNetV2	384×224	14.06 ms	38.36 ms	106.69 ms	153.64 ms
tiny	224×224	16.47 ms	59.97 ms	160.53 ms	225.50 ms
tiny	384×224	22.95 ms	88.90 ms	323.08 ms	336.23 ms
Darknet-19	224×224	32.96 ms	136.40 ms	436.48 ms	400.78 ms
Darknet-19	384×224	49.21 ms	220.13 ms	618.63 ms	595.44 ms

Table 3: Average errors of the tracking and detection networks during tracking on unseen video, both with MobileNetV2-6D architecture. The 224×224 input-sized networks are the tracking (square crop) networks, while the 384×224 are the full-frame detection networks. Mean intersection over union (mIOU) is provided for the detection network only, as it is relevant to the quality of the tracking crop region.

Input	mIOU	X-axis error	Y-axis error	Z-axis error
224×224	-	37.529°	37.538°	21.617°
384×224	0.817	82.006°	89.553°	58.145°

3.8.2 Evaluation on the YCB dataset

To compare with other networks, we train and evaluate the tracking network on the YCB dataset (Xiang et al. 2018), since it offers per-frame poses in videos. Due to time constraints, we do not train or evaluate the detection network, but merely reset the tracking to the ground truth position whenever the tracking network loses the object. We use only the real training data, without generating any new virtual data from the textured 3D models of the objects. This may have lowered accuracy.

Benchmarking is done using the ADD and ADD-S metrics, as defined in Xiang et al. (2018). The ADD metric is the mean distance between the ground truth 3D vertex positions of the 3D bounding box, and the predicted 3D corner locations (reprojected to 3D from the 2D corner predictions using the pose from the perspective-and-point solver). The ADD-S metric uses the mean distance between the closest pairs of 3D

vertices of the ground truth and predicted 3D bounding boxes, to account for ambiguous vertex matching for symmetric objects (such as *eggbox* and *glue*). As was done for PoseCNN (Xiang et al. 2018), the work that introduced the dataset, we use a threshold of 10 cm to calculate the accuracies using these metrics.

Results are shown in Table 4 and Table 5, for a single network trained to detect all objects, but evaluated on one object at a time. Even when using tracking, our method is generally shown to be weaker than PoseCNN and the state-of-the-art PoseRBPF++ (Deng et al. 2021), and appears to be highly susceptible to occlusions and truncations (which cause the low or zero accuracies). While this may be partially due to lack of occlusion augmentations during training on YCB, single-shot methods such as ours have been noted to be more sensitive to occlusions and truncations than dense segmentation methods (Peng et al. 2018). However, given that PoseCNN and all other 6DoF pose networks we have identified are unsuitable for smartphone inference, our network remains to our knowledge the best current option for 6DoF in this hardware domain.

3.8.3 Interpretation of overall results

Generally, the more significant errors observed were rotational, as the 2D positioning of the curler could be detected accurately enough even with the detection model. We note that the Z-positioning of the object (the distance from the camera) cannot be properly determined if the bounding box is malformed, as the perspective-and-point method will not be able to resolve the pose with such bad inputs. While the detection model also failed in this regard, this was less important for our particular application. The tracking model showed better accuracy in both rotation and position along all axes.

When comparing the results on the test set (Table 1) and the unseen video (Table 3), there is a large difference between the rotational errors of the full-frame (detection) model. This is due to the curler in the unseen video being further away from the camera, on average, than in the test set. During real-life use, the average distance of the curler from the camera more closely matched the test set than the unseen video, so the test set results are a better approximation of real-life accuracy of the full-frame model.

However, the unseen video results also show a reduction in accuracy of the tracking (224×224) model, as again seen in Tables 1 and 3. This is due to the greater similarity between the test set and the training set, than between the unseen video and the training set. As such, the tracking model accuracies for the unseen video are more representative of real-life use.

Table 4: ADD accuracies for PoseCNN and our MobileNetV2-6D. We use the RGB results of PoseCNN (without the Iterative Closest Point method), and the 200-particle PoseRBPF++.

Object	PoseCNN ADD	PoseRBPF++ ADD	OURS ADD
002 master chef can	50.9	63.3	63.9
003 cracker box	51.7	77.8	1.0
004 sugar box	68.6	79.6	60.3
005 tomato soup can	66.0	73.0	68.8
006 mustard bottle	79.9	84.7	57.1
007 tuna fish can	70.4	64.2	39.7
008 pudding box	62.9	64.5	0.0
009 gelatin box	75.2	83.0	48.5
010 potted meat can	59.6	51.8	57.3
011 banana	72.3	18.4	0.6
019 pitcher base	52.5	63.7	56.7
021 bleach cleanser	50.5	60.5	50.8
024 bowl	6.5	28.4	0.0
025 mug	57.7	77.9	33.1
035 power drill	55.1	71.8	18.8
036 wood block	31.8	2.3	0.0
037 scissors	35.8	38.7	4.1
040 large marker	58.0	67.1	7.4
051 large clamp	25.0	38.3	0.0
052 extra large clamp	15.8	32.3	0.0
061 foam brick	40.4	84.1	0.6

When using tracking, for the purposes of the training app for correct curler use, the tracking model accuracies are sufficient. Since the app mainly only needs to check that a sequence of actions is being done with the curler, there is a fair amount of leeway for pose errors. Furthermore, when using in the app, the user is not presented with the exact pose output of the model. Instead, a virtual *ghost* model of the curler is overlaid on the screen, showing the correct actions required, and the detected pose is only roughly compared with that of the *ghost* model, to give the user feedback on the correctness of their actions. Regarding the more difficult requirement for rotation along the long axis, the tracking model is capable of detecting the twist motion required for creating certain hair styles, which again renders the method sufficiently accurate for our purposes.

For other applications requiring pose estimation of an object, there is often less room for error. For example, in the case of overlaying a texture on a pose-detected object, or placing a virtual character on the object, inaccuracies lead to jitter in the

Table 5: ADD-S accuracies for PoseCNN and our MobileNetV2-6D. We use the RGB results of PoseCNN (without the Iterative Closest Point method), and the 200-particle PoseRBPF++.

Object	PoseCNN ADD-S	PoseRBPF++ ADD-S	OURS ADD-S
002 master chef can	84.0	87.5	84.2
003 cracker box	76.9	87.6	1.4
004 sugar box	84.3	89.4	72.2
005 tomato soup can	80.9	83.6	72.6
006 mustard bottle	90.2	92.0	65.8
007 tuna fish can	87.9	82.7	49.5
008 pudding box	79.0	77.2	0.0
009 gelatin box	87.1	90.8	70.4
010 potted meat can	78.5	66.9	64.1
011 banana	85.9	66.9	13.9
019 pitcher base	76.8	82.1	60.0
021 bleach cleanser	71.9	74.2	54.7
024 bowl	69.7	85.6	0.2
025 mug	78.0	89.0	46.6
035 power drill	72.8	84.3	26.9
036 wood block	65.8	31.4	3.7
037 scissors	56.2	59.1	25.9
040 large marker	71.4	76.4	38.5
051 large clamp	49.9	59.3	1.5
052 extra large clamp	47.0	44.3	16.7
061 foam brick	87.8	92.6	4.0

positioning of the virtual elements, which is illusion-breaking. While a tracking scheme is still required if the objects to be detected are small or far from the camera, more precise models than our curler networks would be required for these tasks. This is also reflected in most of the results on the YCB dataset (Tables 4 and 5), which show low pose accuracy on some objects, and complete failure on others, showing that improvements are required for usable general-purpose pose detection.

Furthermore, for fast-moving objects or a rapidly rotating camera, as may appear in autonomous driving or headset-based applications, our tracking method would be likely to fail. This is because the object may move outside the bounding box provided by the detection network before the tracking network can start inference. Such applications would require a more complex predictive tracking system based on multiple past object detections, to determine the region that the tracking network will operate on.

3.9 Alternative method using sparse segmentation and dense alignment

As mentioned in 2.5, in the search for an improved pose estimation method, we tested the approach in Hu et al. (2019) on the curler dataset. In this method, the pose model outputs a sparse segmentation map for the curler, and for each pixel in the map, outputs a prediction for the position of each corner point, as well as a confidence value for each prediction. By selecting the 10 highest-confidence values for each keypoint, the pose is then retrieved through the RANSAC variant of the perspective-and-point implementation of OpenCV. Unlike the single-shot-pose version, only the eight corners were detected, not the centroid as well.

The purpose of generating the segmentation map is to handle occlusions better than methods that directly attempt to give the corner point positions, like the single-shot-pose method. Given the hand and hair occlusion issues of our curler detection project, this method could therefore be well suited for our purposes.

In its original form, this alternate model, referred to as segmentation-driven-pose, used a single encoder and two decoders: one for the segmentation, and one for the keypoints. It is based on the Darknet-53 architecture used in YOLOv3 (Redmon and Farhadi 2018), with the duplicated decoders reduced to 2 upsampling layers instead of 5, as the segmentation mask required is coarse (low-resolution), and the YOLO-based object detection decoder already does not require very high resolution.

While the Darknet-53 based architecture is unsuitable for real-time inference on smartphones, due to its depth and use of standard convolutions instead of depthwise separable, we trained it on our curler dataset to check its original precision on the test set and unseen data. Due to the manual conversion required to convert the PyTorch model to TensorFlow Lite, exact latencies for the original form were not benchmarked. For a rough estimation, a YOLOv3 model of the same input size (608×608), as well as a reduced 224×224 size to match the tracking network of the single-shot-pose method, were converted to TensorFlow Lite and benchmarked on the CPU and GPU of a OnePlus 6 Android smartphone. As is evident from Table 6, the models are too slow to be suitable, as expected, given that real-time execution is desired.

For our dataset, we maintained the different bounding boxes for the open and closed curler classes. As with the single-shot-pose model, this may have decreased the quality of the results, but this could not be verified due to time constraints.

To compare the segmentation-driven-pose model with the single-shot-pose model fairly, a version of the latter with the same 608×608 input size was trained. The new method gave slightly worse results on the test set, as shown in Table 7 and Figure 15,

Table 6: TensorFlow Lite YOLOv3 latencies on a OnePlus 6 (GPU uses the OpenCL delegate, CPU uses XNNPACK with 4 threads and no quantization). The lack of exact latency values on the GPU is due to a few operations near the end of the models not being supported on the GPU. The networks were therefore tested with the unsupported layers removed in the GPU case.

Model	Input size	GPU time (ms)	CPU time (ms)
YOLOv3	608×608	>567	2631
YOLOv3	224×224	>106	395

where the latter focuses on the failure cases of the segmentation model.

Table 7: Average per-keypoint precisions, both pixel-relative and input size-relative, of the segmentation-driven-pose method and a full-frame variant of the MobileNetV2-backboned single-shot-pose method, with the same input size. No augmentations were used for either model during testing, to ensure parity.

Backbone	Input	2D kp. error	Input-normalized 2D kp. error
segmentation-driven-pose	608×608	22.86 px	0.0376
MobileNetV2-6D	608×608	16.14 px	0.0266

In practice, when using unseen data, it was observed that this method was significantly less accurate than single-shot-pose, and more susceptible to occlusions, truncations, and background elements. This is shown in Figure 16, compared to the lower-resolution 384×384 detection version of single-shot-pose, used in the tracking system. Despite the lower resolution of this single-shot-pose variant, it visibly and consistently outperforms the segmentation-driven-pose model on this unseen data.

To check the effect of the perspective-and-point algorithm implementation, we tried using various numbers of points for the corners, and also tried using a global confidence cutoff (through which only high-confidence points were considered). The latter tactic was attempted due to the potential benefit of removing noise in the detection, with the potential drawback of leaving too few points for the RANSAC perspective-and-point algorithm. In practice, the results on unseen data were too poor for any improvement with any perspective-and-point method to be observed, and for seen data, the original PnP scheme performed the same as or better than the alternate variants.

Detection errors were observed to be mitigated somewhat by using the same tracking method described for the single-shot-pose method previously implemented, although not to the extent required for it to outperform the single-shot-pose tracking model on unseen data during use. This improvement is likely due to there being fewer background elements to confuse the model in the tracked crop, which increased the

Figure 15: Some results of the non-tracking version of the 608 × 608 input size segmentation-driven pose network (white), compared to the output of the 608 × 608 input size single-shot-pose network (green), and the ground truth (red). Images show the keypoints without the perspective-and-point step applied, as to not mask the strengths or weaknesses of the models themselves. Images a-c show good results for the segmentation-based model, outperforming single-shot-pose in image b (in which the latter is confused by a background element) and c, despite both b and c having the curler partially out of frame. Images d-f show failure cases of the segmentation model, in which the single-shot-pose model gave better results. These may be due to the segmentation maps picking up parts of the dark shirt (d,e) and the dark watch (f), which are the same color as the curler. Images g-i show larger failures of the segmentation model, despite the single-shot-pose variant performing relatively well. These again appear to be caused by background elements confusing the network, which is compounded in i by the curler being largely out of frame (truncated).

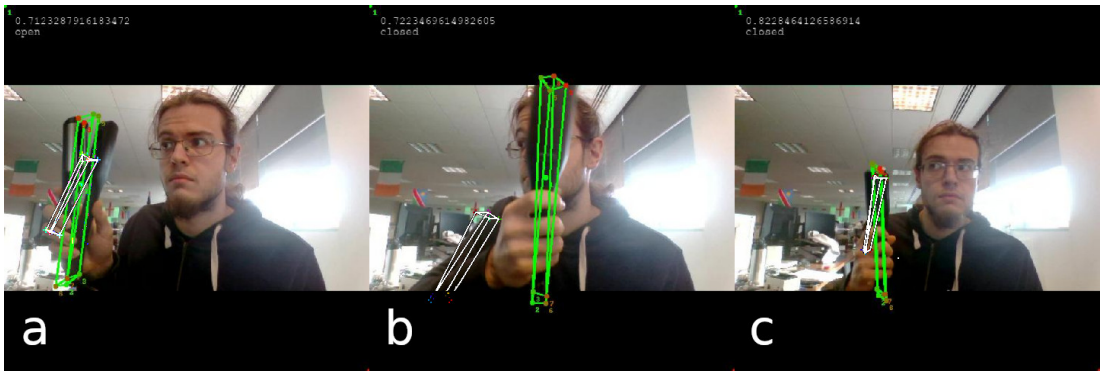

Figure 16: Some results of the non-tracking version of the segmentation-driven pose network (white, 608 × 608 input size) compared to the non-tracking version of the single-shot-pose version (green, 384 × 384 input size), when using square-padded unseen data. Results are virtually always considerably worse, showing general inaccuracy (a), sensitivity to background elements like the sleeved arm (b), and sensitivity to hand occlusions, reflected in the bounding box only encompassing the unoccluded part of the curler in (c), despite robustness to occlusions being a stronger point of segmentation-based 6DoF pose detection methods, according to the original work and similar approaches. The perspective-and-pose step has not been enforced in this image, as it would be unable to correct the bounding box pose due to the very low accuracy of the network-output keypoints.

accuracy of the tracking model. These were initial results, and were not pursued to a greater degree due to the curler project being put on hold indefinitely, and not restarted before the end of the degree. Initial results for the best cropped input (tracking) model are presented in Table 8 and Figure 17 for the test set, and Figure 18 for unseen data.

Table 8: Average per-keypoint precision of the crop-input segmentation-driven-pose model, without perspective-and-point, compared to the crop-input single-shot-pose model.

Backbone	Input	2D kp. error	Input-normalized 2D kp. error
segmentation-driven-pose	256 × 256	19.48 px	0.0761
MobileNetV2-6D	224 × 224	7.796 px	0.0348

3.10 Ethical considerations

Given that in the training dataset, the curler was almost always hand-held, the trained networks may learn to associate the location of a hand with the location of the curler. As a result, it is possible that the network may have different accuracies depending of

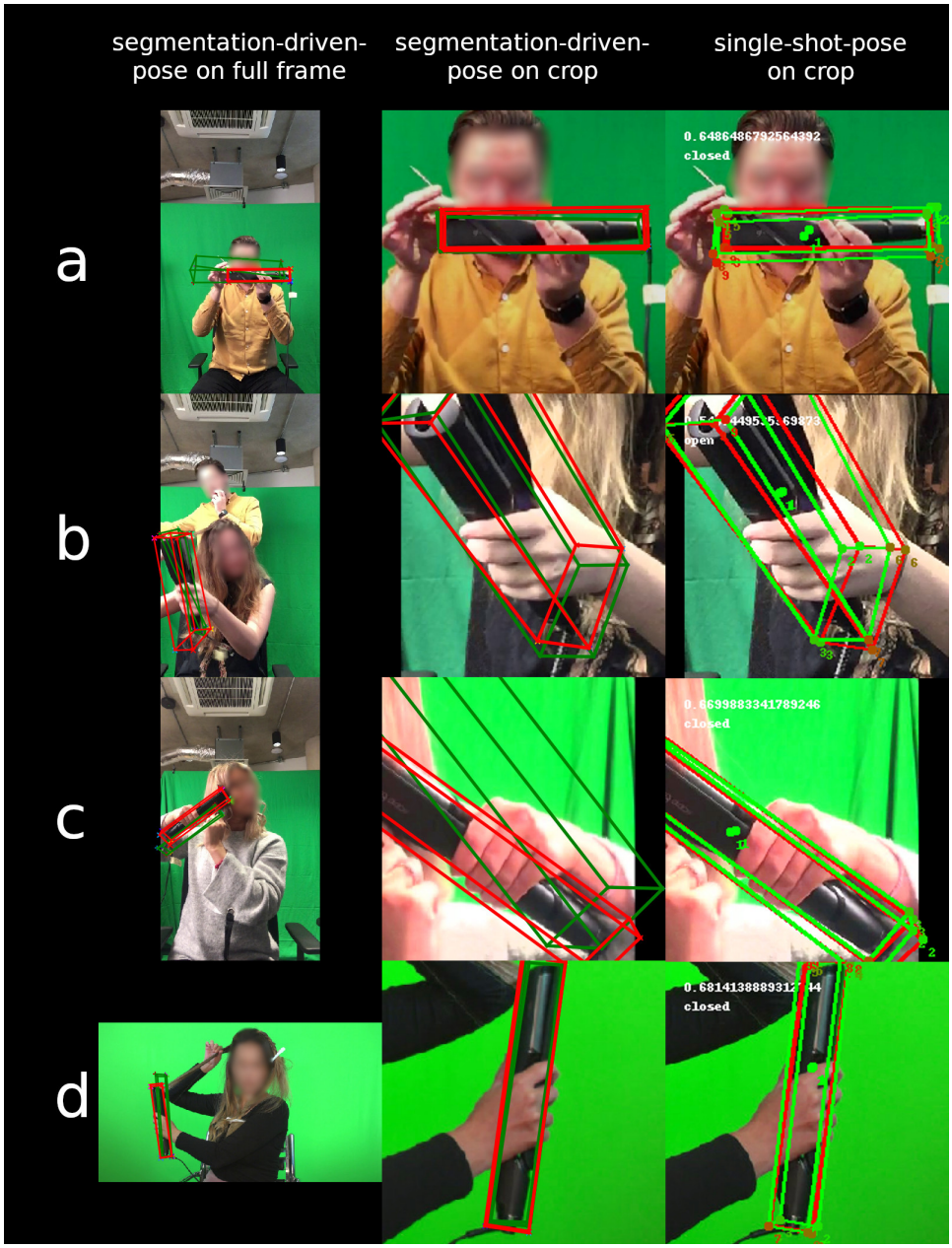

Figure 17: Results of the cropped-input segmentation-driven-pose model (the tracking model, with a 256×256 input size, middle column), compared to the full-frame segmentation-driven-pose model (608×608 input size, left column), and the crop input single-shot-pose model (224×224 input size, right column). Generally, the crop segmentation-driven-pose is more accurate than the full-frame version, but the improvement is not as great as for the single-shot-pose model, possibly due to the greater change in input size. It can be seen in (c) that the output can instead be worse if the input crop is not exact (with truncations). The crop (tracking) variant of single-shot-pose is still more accurate than either segmentation-driven-pose model.

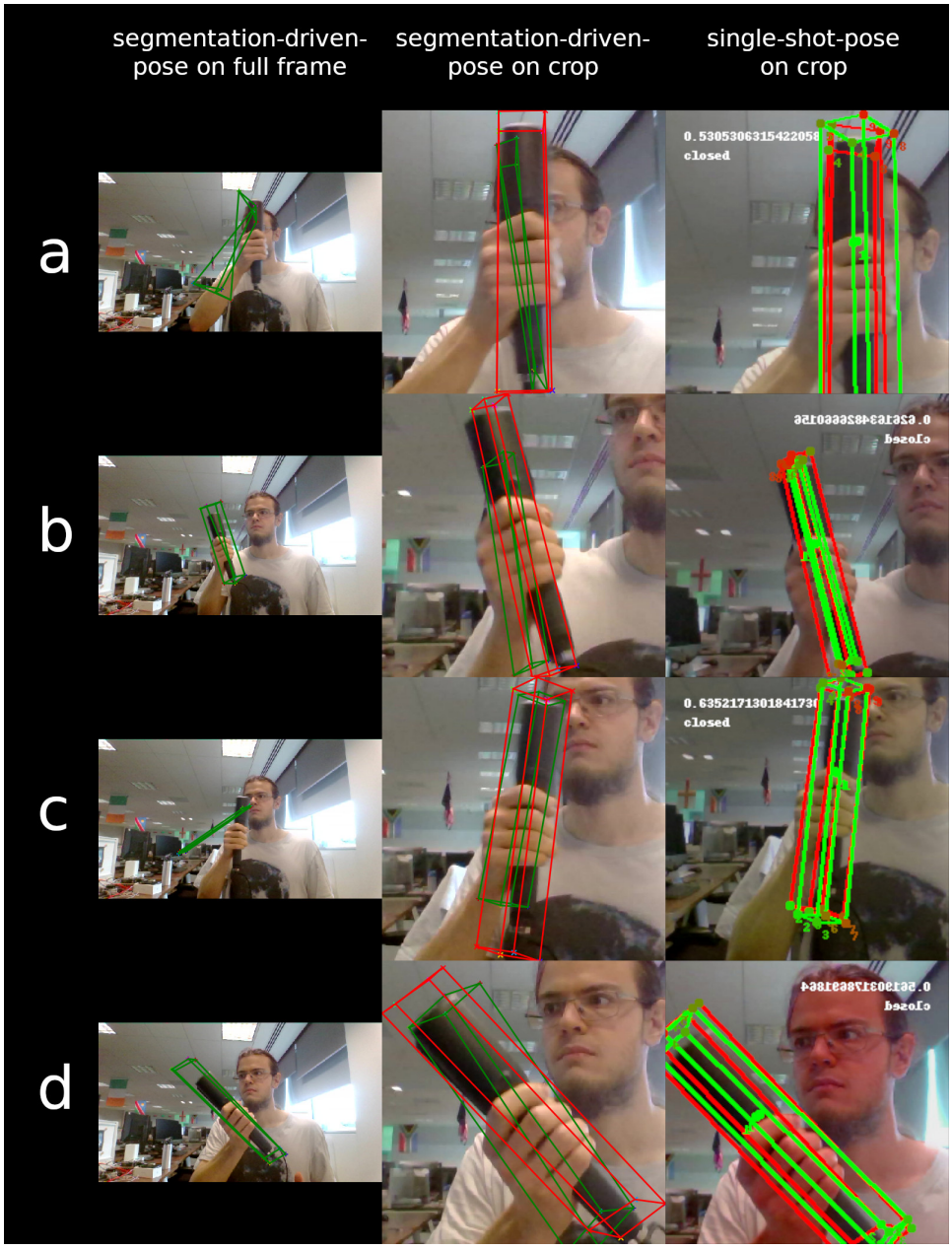

Figure 18: Results of the cropped-input segmentation-driven-pose model (the tracking model, with a 256 × 256 input size, middle column), compared to the full-frame segmentation-driven-pose model (608 × 608 input size, left column), and the crop input single-shot-pose model (224 × 224 input size, right column). Again, the crop segmentation-driven-pose is generally more accurate than the full-frame version, but the improvement is not as great as for the single-shot-pose model, possibly due to the greater change in input size. The crop (tracking) variant of single-shot-pose is still more accurate than either segmentation-driven-pose model.

the skin tone of the hand, either due to skin tone biases in the input dataset or the lower contrast between dark skin tones and the dark-colored curler, which could lead to the network confusing the hand features with curler features.

This could not be verified, since the training dataset only had white and light brown skin tones available, given which people the contracting company was able to recruit for the dataset creation. In addition, no unannotated images with dark-skinned users holding the required curler model existed, that could be used for visual comparison of output quality.

In the future and for similar projects, this could be mitigated either with a more diverse dataset, or by complementing the dataset with CGI images, using photorealistic humans and the well-textured CAD model of the curler. The virtual human tactic was used in some later facial attribute estimation projects at the placement company, but not for any of the projects in this thesis.

In terms of applications to areas other than beauty, the method could be used to track the pose of any object on low-powered devices or otherwise, although it may require improvements for accuracy. With such improvements, it could be used for autonomous driving in small unmanned ground vehicles, or for various augmented reality projects requiring the tracking of real objects. The tracking method could also be used for other tasks such as 2D object tracking and semantic segmentation, in any domain where it might be required.

3.11 Conclusions

By using a simple tracking scheme, we were able to retrieve the six degree of freedom pose of a custom object, in real-time, on both iOS and Android smartphones, with acceptable accuracy. This was accomplished by combining the lightweight and low-latency MobileNetV2 (Sandler et al. 2018), and the single-shot-pose 6DoF pose detection network (Tekin et al. 2018). While the object tracked here was a hair curler, the method can be extended to any object, handheld or not. We were unable to achieve better accuracy using the segmentation-based pose model of Hu et al. (2019).

Future work could involve using a better object detector for the detection network, and improving the pose detection model, potentially by using the improved YOLOv3 architecture (Redmon and Farhadi 2017) as a base, or by modifying a single-shot multibox detector (SSD) model (Liu et al. 2016b) to output keypoints, similarly to how single-shot-pose modified YOLOv2. In addition, motion along the difficult roll angle of the curler could be detected by tracking the hand keypoints as well as the curler, similarly to the approach in Tekin et al. (2019).

Chapter 4

Face Makeup Transfer with Generative Adversarial Networks

While many virtual try-on or photoediting apps, such as YouCam Makeup (PerfectCorp 2021) and Perfect365 (Perfect365 2021), allow for a broad range of customization for the types of makeup colors and styles applied to the user's face, they require a significant amount of manual interaction. The end user needs to choose the various colors or details of the style, or limit themselves to a list of developer-predefined styles. Furthermore, a user may want to try and replicate a particular style employed by a celebrity or makeup artist, without figuring out how to craft the details. In such cases, it would be useful to have a system that simply allows for the scanning of a face wearing the target makeup style desired by the user, and the application of that style to the user's face. This is the goal of face makeup transfer, which this chapter attempts to implement on mobile devices, as they are the main target device for virtual try-on apps.

As mentioned in Subsection 2.6.2, this task can be achieved through machine learning with generative adversarial networks, using the technique from BeautyGAN (Li et al. 2018b). This approach takes the user's face and a face wearing the target style as inputs, and outputs the user image wearing that style. To achieve the effect, it uses GAN loss to ensure the output is a believable instance of a face with makeup, histogram loss to ensure the colors and general location of the makeup matches the target style (for the eye shadow, lips and skin foundation), and perceptual loss to ensure this output otherwise matches the input face of the user without makeup. It furthermore enforces cycle consistency loss similarly to CycleGAN, as shown in 2.6. This is illustrated in Figure 19.

In the BeautyGAN scheme, there are two I_{src} and I_{ref} inputs to the GAN network, where the former is the user's face (with no makeup) and the latter is a face

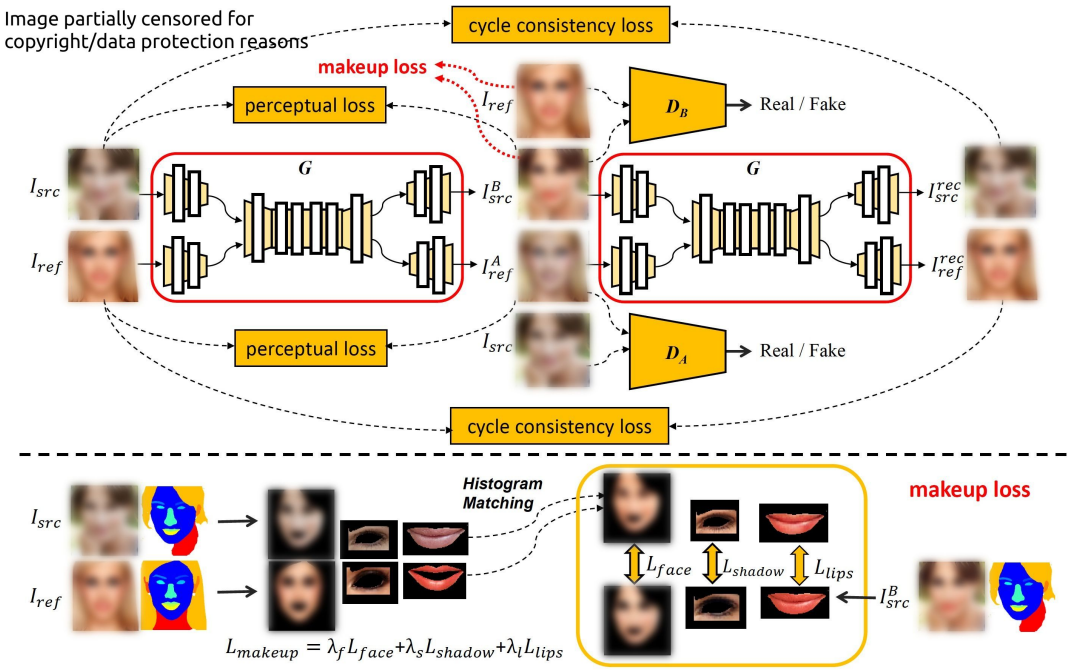

Figure 19: The BeautyGAN training scheme. Figure from Li et al. (2018b).

wearing the target makeup style. The network gives the outputs I^B_{src} and I^A_{ref}, respectively, which are the user's face wearing the makeup style, and the target face with the makeup removed. For these two outputs, respectively, the discriminators D_A and D_B are trained to enforce the similarity to real non-makeup (user) and makeup (style target) images, respectively. Since this does not by itself enforce that the user's face has the same makeup style as the target, this is instead enforced by histogram loss (*makeup loss*) over three regions (skin, lips, and around eyes), which ensures color consistency. To ensure that the input faces stay the same outside of the makeup addition or removal, perceptual loss is used between inputs and outputs. I^B_{src} and I^A_{ref} are then reused as inputs for the generator network, giving the outputs I^{rec}_{src} and I^{rec}_{ref}, respectively. The similarity between I_{src} and I^{rec}_{src}, as well as I_{ref} and I^{rec}_{ref}, is then also enforced by perceptual loss. This is the same cycle consistency scheme used by CycleGAN, except using perceptual loss instead of L1 loss, and using two inputs and outputs instead of one of each, to allow for a settable style instead of CycleGAN's fixed one.

Perceptual loss was originally introduced in Johnson et al. (2016). It was shown to be an improvement upon L1 and other direct losses. Instead of calculating the difference in the image pixel space, it uses the distance in the high-level feature space of networks trained on image datasets. In the case of BeautyGAN, a 16-layer VGG

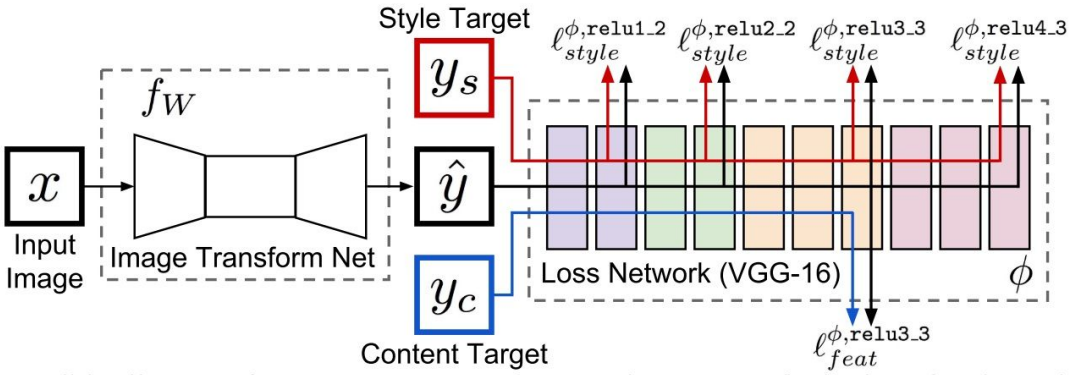

Figure 20: Perceptual losses at different feature depths, as studied in Johnson et al. (2016) (note that the *relu_4_1* layer used in BeautyGAN is not shown). Image from Johnson et al. (2016)

network pretrained on ImageNet (for classification) is used. The perceptual loss in this case is the L1 distance between the *relu_4_1* feature layers when the two images being compared are inputs to the VGG16 network. This is illustrated in Figure 20.

We base our code on the official PyTorch implementation of BeautyGAN (Jiang 2021), which has an architecture slightly different from the the Li et al. (2018b) paper, but shows similar positive results. Several modifications were attempted, to create a variant more suited for integration into the company's beauty apps, and to try to fix certain observed output issues.

4.1 Restriction of BeautyGAN makeup transfer to user-defined regions

In BeautyGAN, there are three face regions treated separately when applying the histogram loss: the eye region, on which the eye shadow and eye liner is applied, the lip region, on which the lipstick is applied, and the skin region, on which the blusher and other skin products are applied. These loss components are illustrated in Figure 19 as L_{shadow}, L_{lips} and L_{face}, respectively.

Since a user might want to transfer only one or two of these three makeup types, some attempts were made to modify BeautyGAN to allow for selective transfer of makeup regions. While the simplest approach would be to merely select the regions using clipping masks on a face mesh fitted to an image of the user, eye shadow and blusher are often smoothly blended at the edges. As a result, these regions may be considered as semitransparent overlays, which when cropped would contain some of the

underlying skin color, which would cause issues if the destination face has a different color or brightness. Furthermore, at the time that this part of the research was carried out, the company's Unity implementation of the face mesh fitter was too limited and approximate for this approach. As such, attempts were made to modify the BeautyGAN network to directly output the user's face with only the selected makeup regions applied.

4.1.1 Vector-encoded region method

Initially, an effort was made to encode the desired regions as a 3-element vector, which was expanded in the X and Y directions to give a $(X, Y, 3)$ shape, such that each element (i, j, k) was 1 if region k was to have the makeup applied, and 0 if it was to be left unmodified. The resulting tensor was fused onto the result of the merging of the two input branches. Aside from the required modification to the subsequent convolution, to keep all following intermediate tensors have the same shape as before, there were no further changes to the network. The training scheme was modified to use a randomized region vector with at least one nonzero element, and the number of iterations in an epoch was increased 7 times, corresponding to the 7 possible region vectors.

This method was however unable to learn to only focus on desired regions based on the input vector. It simply resulted in a more faded application of makeup on the user's face, which would be the same expected result as randomly not applying histogram loss to some regions during training, without any input region vector at all.

4.1.2 Filter separation method

Another region-based method attempted involved using separate parts of the filters in the convolutions of the trunk of the model (that is, the part between the input and output branches) for each region. This was done simply by replacing each convolution with three parallel convolutions with the same input, with filter count F_1, F_2, F_3, such that $F_1 + F_2 + F_3 = F$, with F being the number of filters of the original convolution. The outputs of these three convolutions were then concatenated along the channel axis, which allowed the architecture of the model outside these split convolutions to stay the same. In order to ensure that each convolution only affected the features of its respective region, the training of the convolutions for undesired regions was frozen at each training step (meaning that the weights of the convolutions were not updated during backpropagation), depending on the corresponding element of the Boolean region vector at that step. The region vector is naturally still required as an input, and was

inserted in the same place as for the previous region vector-based method, with the same expansion and channel merging technique described in the previously attempted method.

Implementation and testing of this method were done by a different company employee, and while the results were reported to be unsatisfactory, explicit visual results and proper details of implementation and testing were not communicated before termination of the project. Noted observations were the unwanted regions still being slightly affected, and the desired regions having lower output quality, probably due to the smaller amount of convolution filters applied to each region. It was further observed, however, that increasing the amount of filters used by each region (such that $F_1 + F_2 + F_3 > F$) did not sufficiently improve the results.

Given the lack of explicit results, in-depth testing of this method is left as future work, including several potential improvements described in Subsection 4.8.2.

4.1.3 Multiple generator method

A more crude but straightforward method to allow for user-chosen regions is to train a version of BeautyGAN for each possible choice of enabled and disabled regions. Since all regions being disabled is naturally excluded as an option, this results in seven BeautyGAN variants, in the case of three separate regions. To achieve this exclusion, the histogram loss associated with the unwanted regions is removed from the overall loss. Given the failure of the input-vector based method, this simpler version was attempted in order to obtain a minimum viable method.

When using this method, several issues were noted for the single-region variants, in the case of the eyes and lips. In the case of different skin brightnesses or colours between the non-makeup (user) and makeup (style target) images, a severe brightness change was observed around the lips or eyes, as in Figure 21 and Figure 22. In the case of the eye shadow, this is due to the histogram loss partially taking the skin color into account, due to the aforementioned issue of semitransparent makeup regions and soft, blended makeup edges. While the lip regions should in theory not be affected by this, in practice the same issue was generally observed. This is believed to be due to the inability of the model to precisely segment the lip and eye regions precisely, leading to color changes outside these areas. It should be noted that without an improved loss function that addresses these issues, no region-aware version of BeautyGAN, whether using a single model or seven, could escape this brightness changing effect.

While in the case of the lips, this could be corrected with a lip segmentation network, this largely defeats the purpose of using BeautyGAN for lipstick transfer, as this can be approximated more simply through histogram or tone matching.

Figure 21: A result from trying to only transfer the lip makeup. The lighter skin around the lips is the aforementioned brightness changing effect. It can be noticed that the overall skin color is also slightly changed, despite only the lip region being set to be changed. Inputs from MT dataset (Li et al. 2018b).

Figure 22: A result from trying to only transfer the eye makeup, showing a similar effect to Figure 21, but around the eyes instead of the lips. Inputs from MT dataset (Li et al. 2018b).

For the eye shadow, this brightness change effect shows that the issue of cropping semitransparent regions remains. To some extent, this could be combatted using seamless compositing, such as with OpenCV's seamless clone function based on Poisson blending (Pérez et al. 2003), as attempted previously for mirroring the target style input. This was not looked into due to discontinuation of the project.

4.2 Instance normalization issues

The instance normalization layers of BeautyGAN became a research focus in this chapter, both due to the artifacts it created in the output, as well as the difficulties encountered when attempting to implement instance normalization on the GPU in TensorFlow Lite.

The latter emerges from instance normalization not using the running mean and variance as batch normalization does, which at inference time turn batch normalization into an easily parallelizable linear GPU operation (Ioffe and Szegedy 2015), which can be *fused* into the weights of the preceding biased convolution. Instance normalization, by contrast, requires the use its exact form during both training and inference (Ulyanov et al. 2016). This makes it slower than inference-time batch normalization, and more difficult to implement on the GPU in TensorFlow Lite. This is because there is no instance normalization operation implemented for any GPU delegate, probably due to the calculation of the exact mean and variance being atomic, as for training-time batch normalization. A GPU-supported TensorFlow Lite instance normalization implementation is described in 4.3.2, using a combination of supported layers, in an attempt to allow for faster BeautyGAN inference on smartphone GPUs.

4.2.1 Droplet artifacts from instance normalization

More recently, instance normalization has been avoided in some works such as ESRGAN (Wang et al. 2019d) due to *droplet* artifacts, caused by high-signal features permeating the normalization layers (Figure 23). Perhaps more notably, the adaptive instance normalization (AdaIN) layers in StyleGAN (Karras et al. 2019) were observed to produce similar artifacts (Figure 24), as observed in Karras et al. (2020). In this work, we also observed similar artifacts in the outputs of the BeautyGAN network, but generally in the form of dark spots, likely strong negative signals instead of the positive ones in StyleGAN (Figure 25).

As expected from ESRGAN and StyleGAN, the BeautyGAN droplets vanished once the instance normalization layers were removed. However, use of no normalization at all resulted in a poorer result outside the artifact region. Using normal batch normalization again resulted in the same artifacts, as well as reduced output quality.

With regards to droplet artifact prevention, the most promising form of normalization we encountered was StyleGAN2's modulation-demodulation normalization, as it completely removed these artifacts. This form of normalization has also been successfully used in a more lightweight version of StyleGAN2 (Belousov 2021), which replaced the normal convolution operations with depthwise separable convolutions, similarly to our attempts in Subsection 4.3.1.

Alternative normalization methods such as this could not be looked into due to the project ending, but are deemed to be valuable future work for creating droplet-free variants of BeautyGAN with good output quality.

Exp_64RB
Deeper network with 64 RBs

Exp_BNinHR
with BN in HR space

Figure 23: Droplet artifacts in ESRGAN experiments when using batch normalization. Image from Wang et al. (2021b), based on Wang et al. (2019d).

Figure 24: Droplet artifacts in the original StyleGAN. Image from Karras et al. (2020).

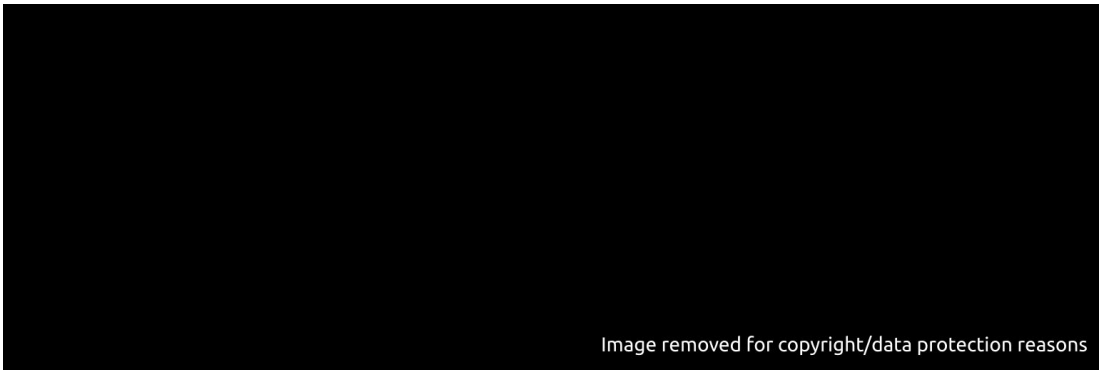

Figure 25: Artifacts observed in outputs of the original BeautyGAN architecture (observable in the top-left portions of the two middle images from each row). The reconstructed original images (the two rightmost on each row, created for cycle consistency) show even more prominent artifacts in the same location. Original input images (the four leftmost) are from the MT dataset of the original BeautyGAN paper (Li et al. 2018b).

4.3 Optimizations for lower latency on smartphones

Before our experiments with instance normalization layers shown in 4.3.2, we tested the original BeautyGAN network with these layers removed, in order to obtain a rough estimation of this network's on-device latency. We found that the speed was acceptable for single-image inference in the case of higher-end phones, as shown in Table 9. In order to decrease latency on lower-end phones, and potentially obtain real-time inference on higher-end ones, several optimizations were tested in this work. These are detailed below.

4.3.1 Latency-improved generator

The first optimization involved replacing the convolutions in the BeautyGAN generator network with bottleneck residual blocks with depthwise convolutions, as used in MobileNetV2 (Sandler et al. 2018). The network contained two transpose convolutions at the end of the trunk, which were left unmodified in all the variants used in these latency improvement tests. The transpose convolutions were later only replaced with normal (not depthwise separable) convolution residual blocks and an upscaling layer, solely with the purpose of testing if certain artifacts in the output were mitigated (see Section 4.5). Latency benchmarking for these versions without transpose convolutions could not be carried out due to time constraints.

As the base code was written in PyTorch, we manually re-created the trained networks in TensorFlow to allow for conversion to TensorFlow Lite, in the same manner as in Subsection 3.7.1. To reiterate, this is done by creating equivalent TensorFlow layers with NHWC ordering, and converting the convolution weights from PyTorch NCHW tensors to Numpy NHWC arrays and loading them into the TensorFlow layers. As mentioned previously in Subsection 3.7.1, while TensorFlow supports both NHWC and NCHW axis ordering, TensorFlow Lite only supports NHWC, and the TensorFlow to TensorFlow Lite converter is unable to automatically reorder the axes, so the axis reordering must be done manually.

The resulting latencies of the networks are shown in Table 9. Comparison between the output quality of the different network architectures is done in Subsection 4.6.2, showing improvements for some of the latency-optimized variants.

4.3.2 GPU-friendly instance normalization layer implementation

When replicating an instance normalization layer through available TensorFlow Lite GPU operations, summing over the height and width (HW) dimensions can be accomplished through a series of average pooling operations, where the final output is of

Table 9: Latencies of original BeautyGAN architecture (without inference-time instance normalization) and the modified version with depthwise separable convolutions (also without instance normalization). The Android benchmark binary and iOS benchmark app were available from the official TensorFlow repository (Google 2021l). The Unity app failed to run the model on the Android GPU for both OpenGL and OpenCL, probably due to an implementation bug, not lack of sufficient GPU resources.

Platform/Regime	**Latency (ms)**			
	CPU OG	CPU Mod.	GPU OG	GPU Mod.
MacOS Unity Player	2240	1067	600	493
iOS Unity	1860	crash	115	-
Android Unity	2750	1335	-/-	-/-
iOS Benchmark app	1860	3293	115	76.36
Android benchmark (CL/GL)	x	704	x	161/484

shape $(B, 1, 1, C)$, followed by a multiplication by $H \cdot W$. For example, for a series of length 1, this involves a pooling operation of kernel size (H, W). This is basically an application of GPU sum reduction techniques, but with restriction to existing TensorFlow Lite GPU ops. While this technique only applies if the batch size is 1, this is the case for most machine learning models during inference.

An example network for the ops replicating an instance normalization is shown in Figure 26, using a pooling series of length 2. Since an instance normalization requires two summations (one to get the mean, one to get the variance), there are two pooling series with the same input size.

As of March 2021, there is still no dedicated TensorFlow Lite operation for instance normalization for mobile GPUs, so the research carried out in this section remains relevant for attempts at real-time GAN inference.

The latency of instance normalization graphs with different pooling series lengths is tested on a OnePlus 6 GPU using the OpenCL delegate, and an iPhone XS using the Metal delegate, and compared to the CPU results on these same platforms (using 4 threads, with XNNPACK enabled). The input shapes tested were the same as the input shapes of the instance normalization layers that appeared in the 256×256-input BeautyGAN model ($64 \times 64 \times 64$, $128 \times 128 \times 128$, and $256 \times 256 \times 256$). All possible pooling series were tested on Android, but only three series for each input size on the iPhone XS. This was due to the lack of ability to batch test TensorFlow Lite models on iOS, given the differing benchmarking applications, requiring that they instead be tested one by one. The results for both smartphones are shown in Table 10, including the average Android latencies for each input size.

With respect to the optimal chain of average pooling operations used, the results on both Android and iOS GPUs showed that the single-pooling chain generally has the

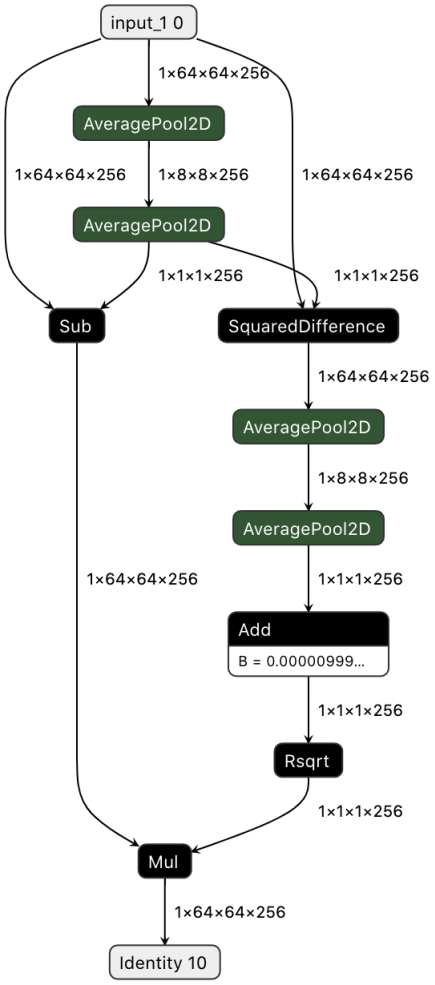

Figure 26: Graph of TensorFlow Lite GPU-compatible instance normalization implementation, rendered with Netron (Roeder 2021).

Table 10: Latencies for the instance normalization implementation on iOS (iPhone XS) and Android (OnePlus 6), with various pooling op sequences, on the GPU (Metal and OpenCL delegates, respectively) and CPU (4 threads, with XNNPACK and no quantization). The pooling factors represent the x and y kernel sizes in the sequences of average pooling operations. Sequences are left identical for both the sum and sum of squares calculations.

Input size	Pooling factors	iOS GPU latency (ms)	iOS CPU latency (ms)	Android GPU latency (ms)	Android CPU latency (ms)
64x64	2,2,2,2,4	9.557	4.105	25.533	2.502
64x64	2,2,2,2,2,2	9.658	4.465	27.653	2.376
64x64	64	10.120	3.885	27.921	4.587
64x64	average	-	-	27.425	2.459
128x128	2,2,2,2,2,4	13.204	9.691	37.370	5.399
128x128	2,2,2,2,2,2,2	17.339	9.630	37.449	5.404
128x128	128	25.795	8.326	44.305	8.864
128x128	average	-	-	37.852	5.189
256x256	2,2,2,2,2,2,4	22.553	23.142	57.644	12.755
256x256	2,2,2,2,2,2,2,2	26.628	23.211	59.247	12.636
256x256	256	32.249	20.754	88.930	14.348
256x256	average	-	-	67.055	12.646

highest latency. Since the longer pooling chains take more advantage of parallel sum reduction, it was expected that they would have the lowest latencies, but this pattern was not observed experimentally. This is probably due to the counteracting effect of the increased cumulative overhead of using multiple TensorFlow Lite GPU operations.

The tests also demonstrate the unexpected result of the GPU implementation being slower than the CPU one, which is likely due to the overhead of using multiple separate GPU operations for the instance normalization. This is somewhat similar to the slow multi-operation depthwise convolution equivalent resulting from the automated PyTorch-TensorFlow conversion attempts in Subsection 3.7.1. This raises the question as to whether falling back to the CPU solely for the execution of the instance normalization layers would result in faster overall performance of the model, despite the overhead of the GPU-CPU and CPU-GPU tensor memory copies required to do this. However, TensorFlow Lite does not perform CPU fallback this way, instead executing the model on the GPU until the first unsupported op, and then executing the entire remainder of the model on the CPU.

These results show that while instance normalization can be accomplished on any GPU delegate of TensorFlow Lite through this method, it is likely too slow for any significant improvement over CPU inference, especially on Android. As such, integration into the BeautyGAN model was not done. Better GPU inference speed for instance normalization layers could potentially be accomplished by implementing them

as a single custom layer, similarly to the efforts in Appendix B. This is left as future work, and detailed in Subsection 4.8.4.

Aside from latency improvements, certain issues in the original BeautyGAN's outputs offered opportunities for visual improvements as well. One issue with face shadows is addressed and mitigated in Section 4.4, and another, regarding checkerboard-like artifacts in the output, is addressed in Section 4.5. The other observed issues are noted in Subsection 4.6.4, with their resolution left as future work.

4.4 Mirrored inputs for mitigating effect of face shadows

A very noticeable output issue consistently arose when the input face with the target style was partially in shadow. This caused the output user's face to have a dark patch corresponding to that shadow, affecting mainly the skin and often one eye (see Figure 27, third column).

This can be mitigated by mirroring the style target face, and keeping the half with the higher overall intensity. We implemented this by using the PRNet face mesh fitter (Feng et al. 2018a,b), which allows a morphable 3D model of a face to be aligned to a 2D image or video automatically, using a convolutional neural network. This allows for face swapping by fitting a face mesh to a source and target image, projecting the source face onto the mesh, and compositing the transformed mesh onto the second image. By setting both the source and target image to the style target, and mirroring the remapped texture associated with the PRNet fitted face mesh, the shadowed part of the face can be replaced with the lighter side.

In the initial mirroring attempt, Poisson image editing (Pérez et al. 2003) was attempted when pasting the mirrored face back into the image. This prevents hard edges between the mirrored half and the original image, by seamlessly compositing the two. However, this often caused the shadow to be maintained, due to the original image being dark in the region near the mirrored half. As a result, the mirrored face was simply pasted back on with no blending.

When using the same BeautyGAN network with the mirrored style target image as an input, it appeared to improve the shadow issue in most cases, as shown in Figure 27. This improvement occurred without retraining BeautyGAN on the mirrored images.

While maintaining only the brighter half of the target style face tends to brighten shadows in the user's face image, generally this has a lesser negative effect on the perceived quality of the image. A possible exception can be seen in the bottom row of Figure 28, in which the final output is brightened too much, particularly on the right side of the image. This could possibly be improved by checking if the user's face image

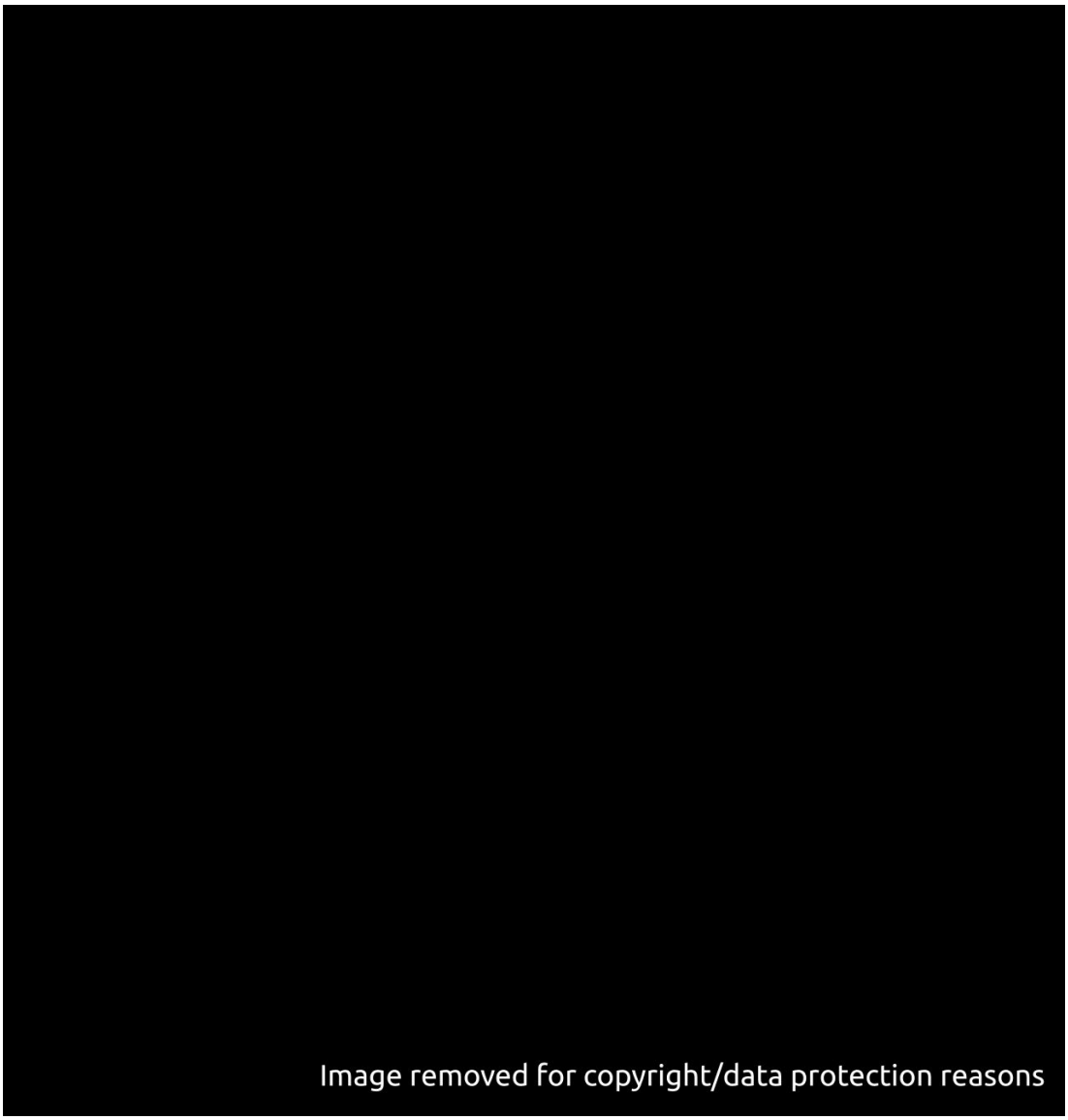

Figure 27: Improved GAN outputs using the mirrored makeup dataset, without seamless cloning. Source and target inputs from MT dataset (Li et al. 2018b).

and the style target image are in shadow on the same side, and not mirroring in this case.

Several failure cases of this technique are shown in Figure 28. As observed in the top row, dark or incorrectly colored patches can occur if the style target face is occluded, for example by a hand or hair. Given that facial occlusion detection outside the lips was limited in our Unity facial segmentation models at the time, this would not have been easy to detect or handle in any mirroring implementation that could be built

on our existing software. Furthermore, it is not trivial to handle the case where one side of the style target face is shadowed and the other is occluded, as occlusion handling in mirroring would likely result in a face with both sides in shadow. This is also observed in the top row of Figure 28, given the dark patch on the left side of the output image, for the mirrored target. Given that making the user choose an unoccluded style target image is deemed to be a reasonable request, finding a suitable solution was not looked into.

Aside from resulting in an overly bright output, the final image in the second row of Figure 28 has a noticeable out-of-place dark region on the bottom right. This is due to the neck being in shadow, which is evidently not handled by the face mirroring. This could be handled by recompositing the face onto a different image, in which the neck is not in shadow. Since the remapped mirrored face is more realistic if the pose of this different image matches the style target, the face could even be pasted on a CGI torso with roughly matched lighting and skin color, to remove any possibility of excessively shadowed regions. While this may complicate execution, the target system being Unity does allow for relatively simple inclusion of a posed 3D head model in the scene. This might also fix the observed persisting shadow issue when trying to use seamless cloning for face recompositing.

A more general issue with the technique is that the makeup in the output image appears to be slightly blurrier when using the mirrored style target, due to the lowered effective resolution of the style target face after mirroring. This can be corrected with higher resolution style target images, as well as improved interpolation when projecting and recompositing the mirrored style target face.

Finally, another issue with this mirroring approach is that some makeup styles are not symmetrical, but this is a relatively rare occurrence.

4.5 Mitigation of checkerboard artifacts in BeautyGAN output

In assessing the output quality of the original BeautyGAN, as well as some variants, we observed checkerboard-like artifacts in the output, as shown in Figure 29. These appeared to increase in intensity as training progressed, becoming noticeable before the generator was sufficiently trained. In all observed cases of these artifacts, they appeared near the eyes or on the lips, as these are the regions most strongly modified by BeautyGAN. Furthermore, the chance of occurrence, as well as the noticeability of the artifacts, appeared to increase if the changes in color required in these regions were large (for example, when trying to apply heavy eye shadow or dark lipstick).

Figure 28: Subpar GAN outputs using the mirrored makeup dataset, without seamless cloning. Source and target inputs from MT dataset (Li et al. 2018b).

Overall, for the original generator architecture, only about 20% of the outputs were not observably affected, while about 30% were heavily affected, with artifacts appearing in more than one of the three regions of possible appearance (left eye, right eye and mouth). With these rates of occurrence, the artifacts would be observable by any user of the app given only a few attempts.

Similar artifacts have been observed in a number of image-generating networks, as shown in Odena et al. (2016), and are attributed to the use of *transpose convolutions* (also called *fractionally-strided convolutions*, or, somewhat incorrectly, *deconvolutions*). We attempted to correct this by replacing the transpose convolutions with an upscaling layer followed by a normal convolution operation, using the original BeautyGAN architecture as the base (without the new MobileNetV2-based depthwise separable convolution blocks).

However, retraining with these changes resulted in slightly different but heavily increased artifacts, as can be shown in Figure 30. As in the case of the version with transpose convolutions, the artifacts appeared to be exacerbated as training progressed. The artifacts appear to always be worse than those in the transpose convolution version, for the same amount of training epochs. We theorized that the instance normalizations could have exacerbated these artifacts, but removing the normalization layers and retraining did not remove the artifacts, showing this to not be the case.

Another effect of replacing the transpose convolutions was an increased incidence of dark droplet artifacts. These were most noticeable in the variant with normal (not depthwise separable) convolution residual blocks, but were also more common in the

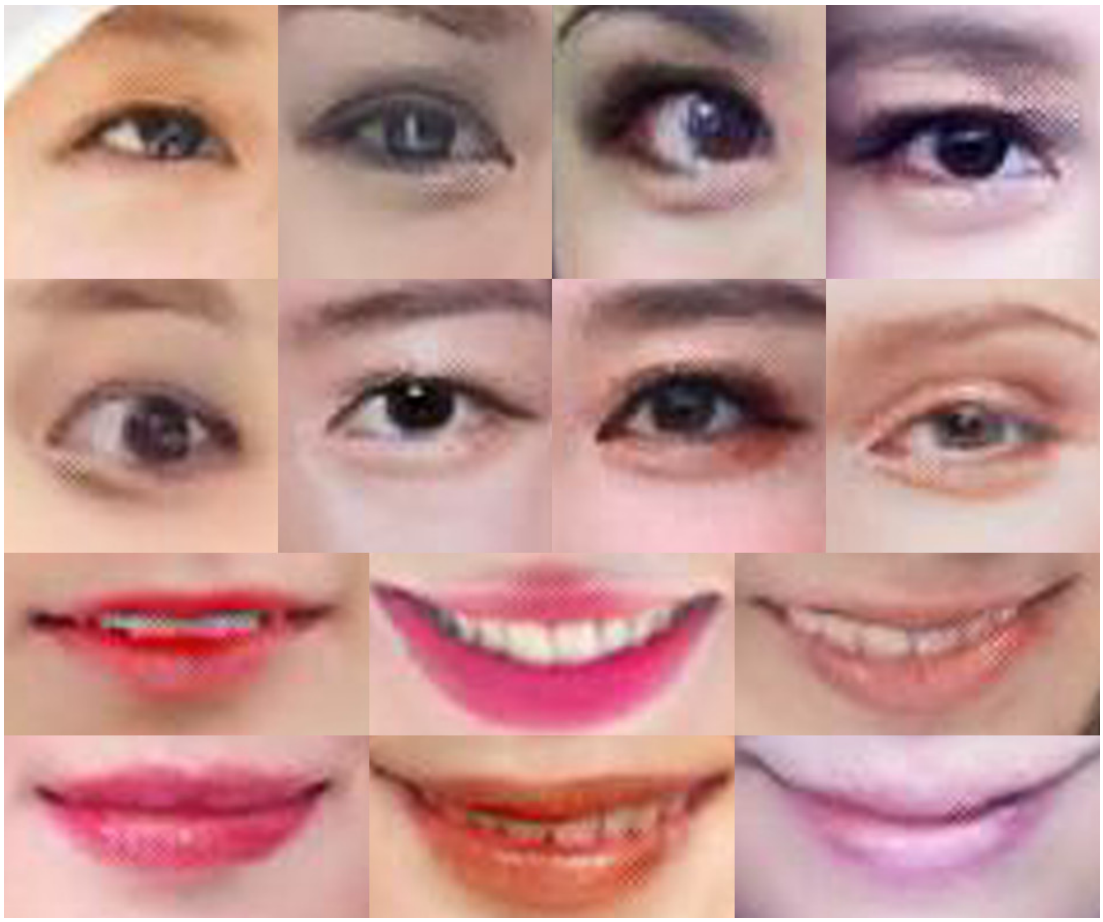

Figure 29: Eye and mouth crops of BeautyGAN outputs showing checkerboard artifacts, originally suspected to be caused by the transpose convolutions. Input images from MT dataset (Li et al. 2018b).

depthwise separable variant with upscaling in the output branches, as observed in the top two rows of Figure 31. While most observable in the reconstructed original inputs, they are also present as completely dark spots in the makeup-transferred outputs.

While simply removing the transpose convolutions did not fix the checkerboard artifact problem, both latency-improved versions using MobileNetV2-style depthwise separable convolution blocks were unexpectedly observed to significantly mitigate the checkerboard artifacts. This is further detailed in the architecture variant comparison in 4.6.2.

Figure 30: Eye and mouth crops of BeautyGAN outputs showing artifacts in the output of the model without transpose convolutions, with exacerbated but somewhat different artifacts. Input images from MT dataset (Li et al. 2018b).

4.6 Output quality assessment and results

4.6.1 Training and evaluation method

The various networks were compared using purely visual comparison, given the failure of our attempts to use automatic numerical methods with the Frechet Inception distance (Subsection 4.6.3). Given the large number of combinations of inputs, at least 50 combinations for each network version were checked. While the sets of inputs were not always the same for each network, for the outputs with issues, the same inputs were tested with all the other networks, for comparison purposes. The training and test datasets were provided with the official PyTorch implementation of the BeautyGAN projects (Jiang 2021), and contained only Asian and Caucasian face images. This dataset contained 335 non-makeup test set images, 780 non-makeup training set images, 816 makeup test set images, and 1903 makeup training images.

In order to optimize for latency and size, lessen output artifacts, or both, and to compare them to the original architecture, four different non-region-aware BeautyGAN versions were tested, based on the proposals in Subsection 4.3.1 and Section 4.5. Two were modified to use depthwise separable convolution blocks instead of normal convolutions, and two with the original normal convolutions and residual blocks. Each pair was formed of a version with the original transpose convolutions at the end of the trunk, and one version with the upscaling and normal convolutions instead. Each network was trained for 300 epochs, and the output was monitored to check that the checkerboard artifacts did not start worsening after a certain number of training epochs, as a bad loss function could potentially cause this.

In terms of qualitative comparison with methods not based on BeautyGAN, this was already undergone in the original BeautyGAN work. As such, only the retrained architecture from the official BeautyGAN PyTorch implementation (Jiang 2021) was used as a baseline for judgment of output quality of our new network variants.

Given that the output quality needs to be judged based on human perception, and given the inability for the automated perception-mimicking Frechet Inception Distance method to be used in our case, no quantitative analysis could be done for this project.

In terms of qualitative analysis, we were unable to do a proper multi-user perception study on the quality of the outputs, as in the 84-volunteer study in the original BeautyGAN paper (Li et al. 2018b). In our case, we were limited to multi-user analysis solely in the form of general output quality impressions, supplied by other company employees for some of the network variants, due to time constraints and abandonment of the project. The remainder of the output quality analysis was based on manual impressions by the thesis author.

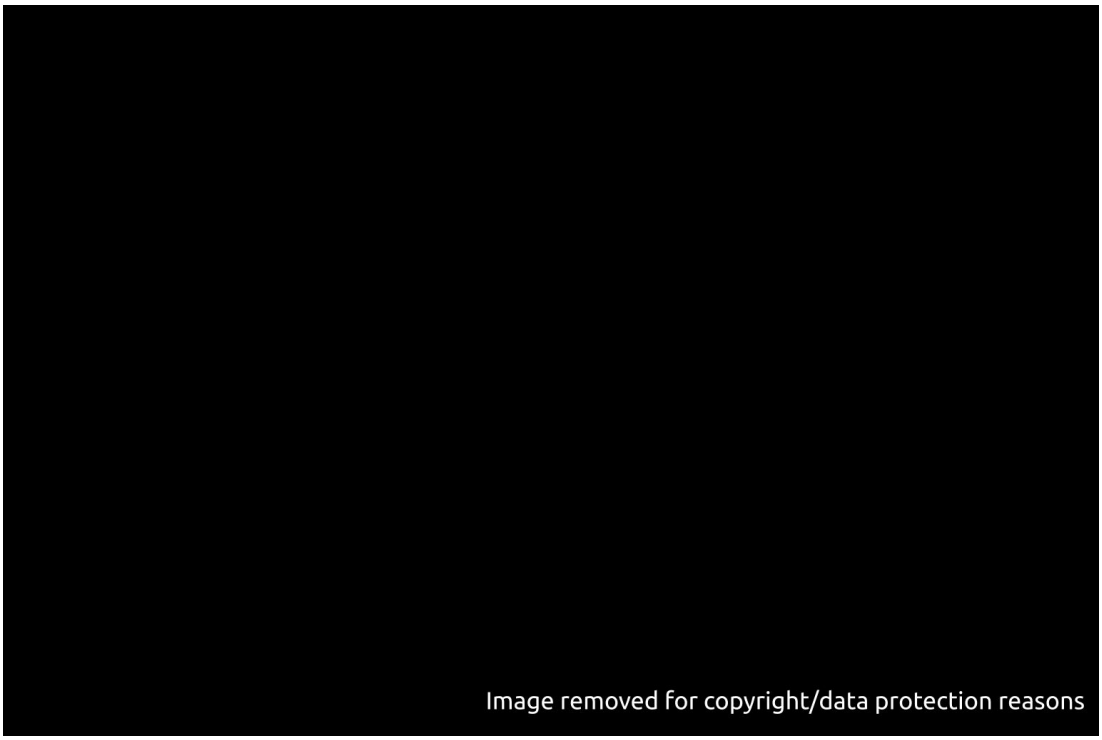

Figure 31: Comparison between the different BeautyGAN variants. The top two rows correspond to the transpose convolution-free variants, while the bottom two have the same output branches as the original architecture. The second and fourth row variants use the depthwise separable convolution blocks, while the other two use the original architecture. Concordantly, the original BeautyGAN architecture is on the third row. Input images from MT dataset (Li et al. 2018b).

Given the variable degree of output quality, as well as the shortcomings of the method, we do not measure a success rate for the variants, but merely note and discuss the observed output issues for different network variants and preprocessing methods, such as target style image mirroring (Section 4.4).

4.6.2 BeautyGAN architecture variant comparison

Since the effects of the transpose convolutions and the upscaling with normal convolution versions have already been discussed, focus is put on the effects of the depthwise separable convolution (MobileNetV2-style) residual blocks used in the modified generator. These are illustrated in Figure 31, with the relevant GAN outputs being on the third column. Close-up views of the eyes, illustrating the presence or absence of artifacts, are shown in Figure 32.

The most notable effect of the depthwise separable convolution versions was the

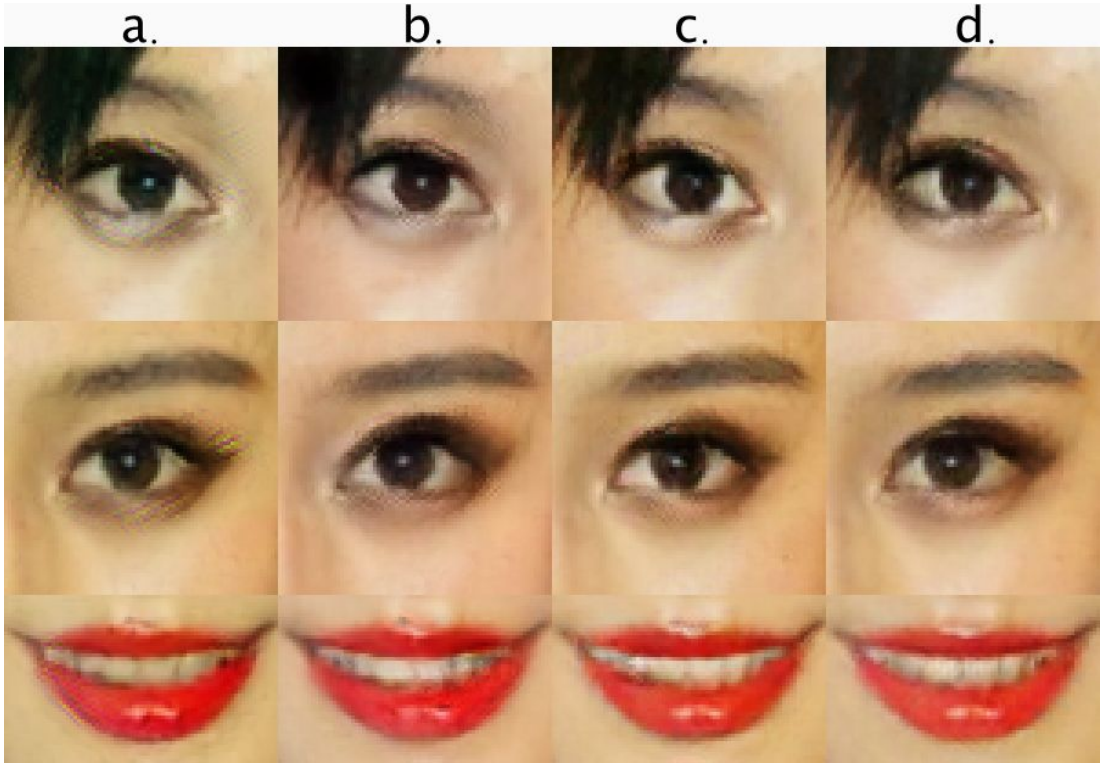

Figure 32: Comparison between the different BeautyGAN variants, with close-up on left eye, right eye and mouth. BeautyGAN variants from left to right: a. with normal convolutions and without transpose convolutions; b. with depthwise separable convolutions and without transpose convolutions; c. with normal convolutions and with transpose convolutions; d. with depthwise separable convolutions and with transpose convolutions. Input images from MT dataset (Li et al. 2018b).

lessened presence of the artifacts mentioned in Section 4.5, for both the transpose convolution versions and the upscaling ones. In the upscaling versions, the artifact severity appears to be reduced greatly in virtually all images, while in the transpose convolution versions, the artifacts appear to be missing completely, although there do still appear to be some faint grid-like structures around the makeup application regions.

While both full convolution variants appears to otherwise give slightly better larger scale results on average, such as improved shading when one or more of the inputs has bad lighting, and slightly less blurriness in the makeup of the output, the depthwise separable convolution version with the transpose convolutions is considered to give the best output overall. This variant is therefore both generally faster (given Table 9) and more accurate than the original architecture.

4.6.3 Estimating output quality with the Frechet Inception Distance

So far in this project, the quality of the output has been determined manually through visual inspection, without any hard numerical quality estimate for more rigorous variant comparison. Although this is in line with quality assessment in the original BeautyGAN paper, some form of automated quantitative measurement would be preferable.

For GANs that solely aim to replicate a given training dataset, like how StyleGAN aims to generate realistic faces based on Flickr-Faces-HQ (Karras et al. 2019), the similarity of the GAN output to the training dataset can be estimated numerically with the Frechet Inception Distance (FID), a scalar value which shrinks as the perceptual difference between two datasets shrinks (Heusel et al. 2017). The FID is based on the value of an internal output of the ImageNet-trained InceptionV3 network (Szegedy et al. 2016), when taking the training dataset and GAN outputs as inputs. If μ_{GAN} and Σ_{GAN} are the covariance matrix and mean vector of the InceptionV3 internal outputs when taking the GAN outputs as inputs, and μ_{real} and Σ_{real} the equivalent for the real training images, then the FID follows Equation (4.1).

$$FID = |\mu_{GAN} - \mu_{real}|^2 + tr(\Sigma_{GAN} + \Sigma_{real} - 2(\Sigma_{GAN}\Sigma_{real})^{1/2}) \tag{4.1}$$

In this case, the 2048-element internal output of InceptionV3 is used, which is immediately before the final fully connected layer giving the ImageNet class probabilities. It needs to be noted that the FID is meant to be calculated using larger datasets, with a number of samples at least equal to the number of elements in the output it uses, here 2048 (Heusel et al. 2017; Johannes Kepler University Linz Institute of Bioinformatics 2017). This was not possible due to the small size of the datasets available to us in this project, meaning that the resulting values are not comparable to FID values from

other works.

The FID is calculated for the variants with MobilenetV2-style blocks, and the variants with the original normal convolution blocks. Both types have two architectures tested: one with transpose convolutions at the end of the trunk, and one with the upscaling and normal convolutions instead. The results are shown in Table 11. For each architecture, four FID values are calculated, using 2002 GAN output images generated from the makeup and non-makeup test sets, and each of four real image sets - the 335 non-makeup test set images, 780 non-makeup training set images, 816 makeup test set images, and 1903 makeup training images.

Table 11: Frechet Inception Distances for the training and test datasets, for all the non-region-selectable BeautyGAN variants trained.

MobileNetV2 blocks	Transpose convolutions	Makeup test FID	Non-makeup test FID	Makeup training FID	Non-makeup training FID
no	no	39.24	45.12	51.93	47.15
yes	no	37.68	43.14	50.86	46.03
yes	yes	33.17	42.47	57.31	53.21
no	yes	37.72	45.53	54.65	50.23

While this suggests that the transpose convolution variants are better than their corresponding variants with upscaling and normal convolutions, it was observed that randomly dividing the combined set of makeup and non-makeup datasets in two gave an FID of 8.37 between the two halves, which can be considered a rough margin of error for BeautyGAN FID calculations. Since this is greater than the difference between any two of the four variants, the results are deemed inconclusive, and the FID deemed unsuitable for BeautyGAN output quality measurement, at least using the datasets available to us. Output quality can therefore only be estimated using manual user perception, as the original BeautyGAN work did for assessment and comparison with prior methods (Li et al. 2018b).

4.6.4 General output issues and limitations

Due to multiple unresolved issues eventually noticed with the output of even the original network, the attempts at making the network more mobile-friendly were discontinued. These are general issues, that affect the model regardless of whether transpose convolutions were used, and whether depthwise separable convolutions were used in the encoder and trunk of the model.

The most significant issue of this approach is the inability of the GAN to extend to unseen makeup styles, which was the main reason this project was discontinued.

This is likely at least partially a result of the small dataset size. Due to this, there were some efforts made to boost the dataset with frames from YouTube videos. Aside from increasing the dataset size, this also decreases the chances of the images being photoedited, as videos are not modified as often or as easily as still images. However, when attempting to restrict the collected images to permissively licensed videos, which are only a small fraction of YouTube videos, not enough high-definition videos were able to be found. Furthermore, since uploaders tend to set the license channel-wide, many of the sampled videos were of the same person, leading to a insufficiently varied face dataset. The dataset quality was further affected by YouTube's relatively low bitrate, which is required for streamed videos. This increased the sizes of faces required, and by extension the minimum video resolution we needed to search for. Also, even if non-permissively licensed videos were to be used, a significant annotation effort would have been needed to separate the makeup and non-makeup images.

Another issue plaguing most of the output images is the apparent blurriness of the makeup on the user's face in the output image. We believe this to be a limitation of the histogram loss, as this only enforces the distribution of the colors in the histogram, not the spatial layout of the colors in the region. As an example, shuffling the locations of the pixels in a region would give the same histogram loss, even though the visual appearance would clearly not be correct in most cases. To mitigate this, an alternative to the histogram loss would be required, that enforces the spatial distribution of the makeup colors on the face. This is similar to the more recent approach in Li et al. (2020), which uses a *UV loss* term for this purpose.

We further observed that many images from the makeup style dataset were heavily photoedited, as is the case for many glam shots. This was much more rarely the case for the no-makeup dataset. This induces a systematic difference between the makeup dataset and the output of the GAN, that no amount of training and accuracy can overcome. This will likely inflate the loss value unfairly and affect the training, and will likely also affect the FID score.

Another important issue can be noticed in almost all the example figures in this chapter: the target style, especially for the original non-region-based BeautyGAN, imposes its own skin color on the user's face. For dark-skinned targets and light-skinned user faces, or vice versa, the BeautyGAN method fails completely. This failure has ethical ramifications, discussed in the next section. This same failure is also observed when the lighting conditions of the user and style images vary, which occurs often, with many user images being poorly lit. A secondary effect of this is the unrealistic face lighting in many of the outputs, since the shading of the target style image is adopted by the output. This remains an issue even when the target style image is mirrored,

sometimes causing the output to appear "flat", and have unrealistically symmetric and bright shading.

Finally, given the slowness of the GPU-runnable instance normalization variant suggested in Subsection 4.3.2, none of the network variants can run in real-time, or benefit from hardware acceleration on smartphones.

4.7 Ethical considerations

The makeup transfer system was a feature of the placement company's makeup apps, which aimed to contain personalized makeup product suggestions with the purpose of driving product sales, and to tune a recommendation engine for this purpose. While this does carry ethical concerns, the author of this thesis was not involved with these parts of the resulting apps, no in-depth ethical analysis can be made.

With respect to fairness in output quality, the datasets available for training the networks contained solely light-skinned people (Asian and Caucasian) for both the makeup and non-makeup images. As the network is unable to transfer makeup properly when the input makeup and non-makeup images have noticeably different brightnesses, it would not be able to transfer makeup if the images had different skin tones either.

The case of both the makeup and non-makeup images having dark skin could be supported if the appropriate training data were added, but it is unlikely that users would be content with being restricted to using a target makeup style that matches their skin color.

These issues cannot be solved without a much more powerful method of extracting the makeup color without retaining any of the skin color. Some form of Poisson image editing could potentially be used for this purpose in future work.

Regarding the value of this research to the greater scientific community, BeautyGAN-like region-based control of GAN influence has applications beyond beauty, for example in the domain of GAN-based art, as seen in tools such as Artbreeder (Simon 2021). By restricting the influence of changes, it could allow for better fine-tuning of outputs through combination of local elements from different generated images, as combining them in latent space would result in an undesirable global mixture of the images.

4.8 Conclusions and future work

Despite the termination of the project, a modified variant of the architecture was able to be created that improved both speed and output quality, by replacing the convolutions

and convolutional residual blocks with depthwise separable convolution blocks, while keeping the transpose convolutions unchanged. While this resulting architecture could not be run on the GPU due to the instance normalizations, with the GPU-supported instance normalization variant we proposed adding too much latency, slower Android phones should benefit from the increased CPU inference speed, given the results in Table 9.

But given the general issues noted with the method in Subsection 4.6.4, mainly poor generalization and blurriness of output makeup, this method is not deemed to be suitable for use in apps in its current form, even with a larger dataset. We discuss several avenues of improvement below.

4.8.1 Alternative approach through makeup overlay generation

For the purposes of beauty apps using our virtual try-on system, another potentially superior way of tackling the makeup transfer task is to use a GAN to deduce a makeup overlay matching the target style, as opposed to generating the entire face of the user with the makeup applied. This would allow the makeup style to be tracked to the user's face, using the same face mesh fitter used for the regular virtual try-on. Since this task was further displaced from the approach in the original work we based our research on, and application of the style onto a single frame was deemed to be enough by the company, this was not attempted. It is considered to be potentially valuable future work.

4.8.2 Potential improvements to the filter separation method for region selection

The failure of the filter separation method in Subsection 4.1.2 may have been due to not properly enforcing the lack of makeup transfer on unwanted regions, as the training was always frozen for the convolutions associated with the unwanted regions. This could potentially be corrected by disabling the effect of the unwanted region convolutions at inference time, perhaps by multiplying their outputs by the respective element of the region vector (zero for wanted, one for unwanted). Alternatively, a two-step backpropagation scheme could be attempted, with the first step optimizing the wanted region convolutions using local histogram loss and non-local perceptual loss, while keeping the unwanted region convolutions frozen. The second step would do the opposite, by freezing the wanted region convolutions and optimizing the unwanted ones with perceptual loss outside the wanted regions. As this does not enforce the global effect of all convolutions working at once, a third step could be added if necessary, that

would freeze nothing and optimize the global effect.

4.8.3 Sub-pixel convolutions for decreased checkerboard artifacts

A different potential solution to the checkerboard artifacts noted in Section 4.5 is using sub-pixel convolution, as noted in Aitken et al. (2017), which has been used in super-resolution tasks. This appears to be implementable using a normal convolution and a depth-to-space operation, both of which appear to be supported both on the CPU and GPU in TensorFlow Lite. This could perhaps improve the output of the original architecture, by changing only the transpose convolutions.

4.8.4 Custom TensorFlow Lite instance normalization operation for faster GPU inference

Given the results of the GPU implementation of instance normalization in Subsection 4.3.2, a better custom implementation as a single GPU op, as in Appendix B, is likely still required for real-time GAN inference on smartphone GPUs. Such an op would also be implemented using parallel sum reduction, but probably use only two GPU buffers using *ping-pong* buffering - the destination buffer of the previous parallel sum becomes the input of the current one, and the source of the previous is re-used as the output of the current parallel sum. This is a technique commonly used in shader programming (Guha 2015).

Chapter 5

Wrinkle and Blemish Detection

In the application of makeup, as well as in beauty in general, significant focus is placed on the obscuring of various skin blemishes, like acne, blackheads, or birthmarks, as well as obscuring the effects of aging, like the presence of wrinkles. The detection of such skin artifacts can be exploited to drive the user to purchase beauty products. This has lead to a significant amount of research on the subject, as well as beauty and photo retouching applications with blemish and wrinkle detection or removal capabilities (see Section 2.7). Such a detector can also be used to estimate an overall skin score, offering the user a simplified overview, for the same purpose.

As a result, the work described in this chapter is focused on automated detection of both facial wrinkles and blemishes (acne, moles, etc.), for potential use in a smartphone beauty application. Given that wrinkle and blemish detection tasks require semantic segmentation, and no suitable datasets could be found and used, purely machine learning-based approaches would have required new, expensive-to-create datasets. As a result, the main focus of this chapter is on traditional computer vision techniques, which do not use machine learning. Some existing machine learning functionality from the Unity and TensorFlow Lite system was however still required.

In this chapter, the work of Elbashir and Yap (2020), based on Jerman's enhancement filter from the blood vessel detection work of Jerman et al. (2016), is extended from just wrinkle detection to blemish detection as well, and implemented and evaluated using forward-facing female faces from the IMDb-WIKI (Rothe et al. 2018, 2015) and FFHQ (Karras et al. 2019) datasets. This was implemented as a C++ Unity plugin, similarly to the Unity and TensorFlow Lite system detailed in Chapter 2.4, and can be run on smartphone CPUs. While the detection step itself uses classical computer vision techniques with OpenCV, the skin region to be tested is retrieved using the skin segmentation network from the Unity and TensorFlow Lite system.

The main experimental contribution was showing that the blob detection from Jerman et al. (2016) can be, to some extent, used for the detection of generic facial blemishes and point-like structures in skin texture. From this, a largely shared implementation of both wrinkle and generic blemish detection was created, detailed in Section 5.2.

As a second contribution, based on the first step of an existing wrinkle detection work (Batool and Chellappa 2015), we propose a variation of Gabor filter banks for wrinkle detection, using an expected direction map. This method is based on our observation that wrinkles, when present, generally follow the same directions from person to person, depending of the part of the face they are on. This approach could be used both as a post-processing step to Jerman's enhancement filter for wrinkles, for the removal of noise, as well as on its own, as described in Section 5.3. This did not lead to a full implementation and was not extensively tested due to time constraints.

5.1 Jerman's enhancement filter method

In the work of Jerman et al. (2016), variations in image intensity with elongated or blob-like shapes, corresponding to transverse and cross sections of blood vessels in angiographic images, are detected based on the eigenvalues of the Hessian matrix. The transverse and cross section detection filters are referred to as the *vesselness* and *blobness* filters, respectively.

As shown in Equation 1 of Jerman et al. (2016), here restricted to a two-dimensional image, the 2D Hessian matrix $H_{1,2}(\mathbf{x}, s)$ of the grayscale image intensity $I(\mathbf{x})$ at coordinate $\mathbf{x} = (x_1, x_2)$ and scale s is given by Equation (5.1), where $G(\mathbf{x}, s)$ is a 2-variate Gaussian defined by Equation (5.2).

$$H_{1,2}(\mathbf{x}, s) = s^2 I(\mathbf{x}) * \frac{\partial^2}{\partial x_1 \, \partial x_2} G(\mathbf{x}, s) \tag{5.1}$$

$$G(\mathbf{x}, s) = \frac{1}{2\pi s^2} exp\left(\frac{-\mathbf{x}^T \mathbf{x}}{2s^2}\right) \tag{5.2}$$

The two eigenvalues of this Hessian matrix, λ_1 and λ_2, are defined so that $\lambda_1 < \lambda_2$.

In Jerman et al. (2016), the case of 3D (volumetric) images is also described, for which the 3D Hessian matrix has a third eigenvalue λ_3. Assuming $\lambda_3 > \lambda_2 > \lambda_1$, the 2D version can be extracted from the general 3D case by setting $\lambda_3 = \lambda_2$.

In Elbashir and Yap (2020), only the detection of elongated structures is addressed, corresponding to wrinkles. This is the same as the two-dimensional vesselness filter from Jerman et al. (2016). This defines a value λ_ρ as in Equation (5.3), where

(x, y) is the pixel location in the image, and σ and τ are the scalar scaling and thresholding controls, respectively. This leads to the final response function in Equation (5.4).

$$\lambda_\rho(s) = \begin{cases} \lambda_2 & \text{if } \lambda_2 > \tau max_{x,y}\lambda_2(x, y, \sigma) \\ \tau max_{x,y}\lambda(x, y, \sigma) & \text{if } 0 < \lambda_2 \leq \tau max_{x,y}\lambda_2(x, y, \sigma) \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

$$V_P = \begin{cases} 0 & \text{if } \lambda_2 <= 0 \vee \lambda_\rho <= 0, \\ 1 & \text{if } \lambda_2 >= \lambda_\rho > 0, \\ \lambda_2^2(\lambda_p - \lambda_2)\left(\frac{3}{2\lambda_2+\lambda_p}\right)^3 & \text{otherwise} \end{cases} \tag{5.4}$$

In the original vascular structure detection paper of Jerman et al. (2016), a variant for 2D *blob* detection (in their case, for detection of cross-sections of blood vessels) is presented. The corresponding equation is shown in Equation (5.6), which differs from Equation (5.4) only in the eigenvalue used, and the new value of λ_ρ (Equation (5.5)) being relative to the new eigenvalue. We have found this to be useful for detection of blob-like blemishes, such as acne and birthmarks.

$$\lambda_\rho(s) = \begin{cases} \lambda_1 & \text{if } \lambda_1 > \tau max_{x,y}\lambda_1(x, y, \sigma) \\ \tau max_{x,y}\lambda(x, y, \sigma) & \text{if } 0 < \lambda_1 \leq \tau max_{x,y}\lambda_1(x, y, \sigma) \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

$$V_P = \begin{cases} 0 & \text{if } \lambda_1 <= 0 \vee \lambda_\rho <= 0, \\ 1 & \text{if } \lambda_1 >= \lambda_\rho > 0, \\ \lambda_1^2(\lambda_p - \lambda_1)\left(\frac{3}{2\lambda_1+\lambda_p}\right)^3 & \text{otherwise} \end{cases} \tag{5.6}$$

5.2 Wrinkle and blemish detection implementation based on Jerman's enhancement filter

5.2.1 Selection of evaluation images

The datasets used for evaluation of the method, as well as optimizing the input parameters, were part of the IMDb-WIKI dataset (Rothe et al. 2018, 2015), which offered precise age, and the FFHQ dataset (Nvidia 2021), which offered higher-quality images, but only approximate age using cloud machine learning inference (from DCGM and Nvidia (2020)). Only female faces were considered, as facial hair is picked up by both

the wrinkle and blemish detectors.

While the IMDb-WIKI dataset offered cropped and aligned faces, these had been resized, making it difficult to filter out low-resolution faces. Therefore, we re-cropped and re-aligned the images from the originals with our minimum size constraints, using an offline face detector and keypoint estimator.

The algorithm was run solely on faces roughly facing the camera. Images were filtered by placing limits on the maximum face pose angle relative to the camera plane normal, as well as the face size (measured as the crop size of the Dlib face detector). We evaluated using 20 degrees and 300 pixels, as well as the more stringent 10 degrees and 500 pixels.

When selecting dataset images, we did not take into account that face wrinkles tend to vary with facial expression. While this is not relevant in use, as the user can be asked to keep a neutral face, it does cause some unwanted variation when running on available datasets. This could be mitigated to some extent through annotation with an emotion detector, and using only the images with a neutral expression. Cloud annotation services like Azure and Google's Cloud Vision AI both offer facial expression estimation (Microsoft 2021a; Google 2021a), so they could be used for this purpose.

5.2.2 Algorithm and code implementation

The original MATLAB code from Jerman et al. (2020, 2016) was ported to C++ using OpenCV (Gary 2008). This was used to create a Unity plugin, which operated on inputs provided by the Unity and TensorFlow Lite system specified in Chapter 2.4, using its face keypoint detection, face mesh fitting, and facial feature segmentation capabilities.

The full procedure is shown in Figure 33. The algorithm takes the detected face a, facial skin segmentation map b, face-aligned region mask c, and eye and mouth keypoints d from the Unity system. The images a, b, c are rescaled based on the keypoint positions (d), to ensure similar scaling in wrinkle and blemish detection, for all input faces. The segmentation map is *eroded* (shrunk from the edges) to prevent unwanted regions from being included due to imprecision in the segmentation map and aligned region mask, giving the eroded segmentation map e. This is intersected with the region mask c, giving the final mask f. The initial, grayscaled image a is then normalized based on the pixels in this region (shown in g), giving the output h. The wrinkle and blemish detection algorithms are applied to the whole image h, giving the outputs i and k, respectively, which can be overlaid onto the original image in Unity, giving outputs similar to j and l.

The original MATLAB algorithm implementation (Jerman et al. 2020), as well as

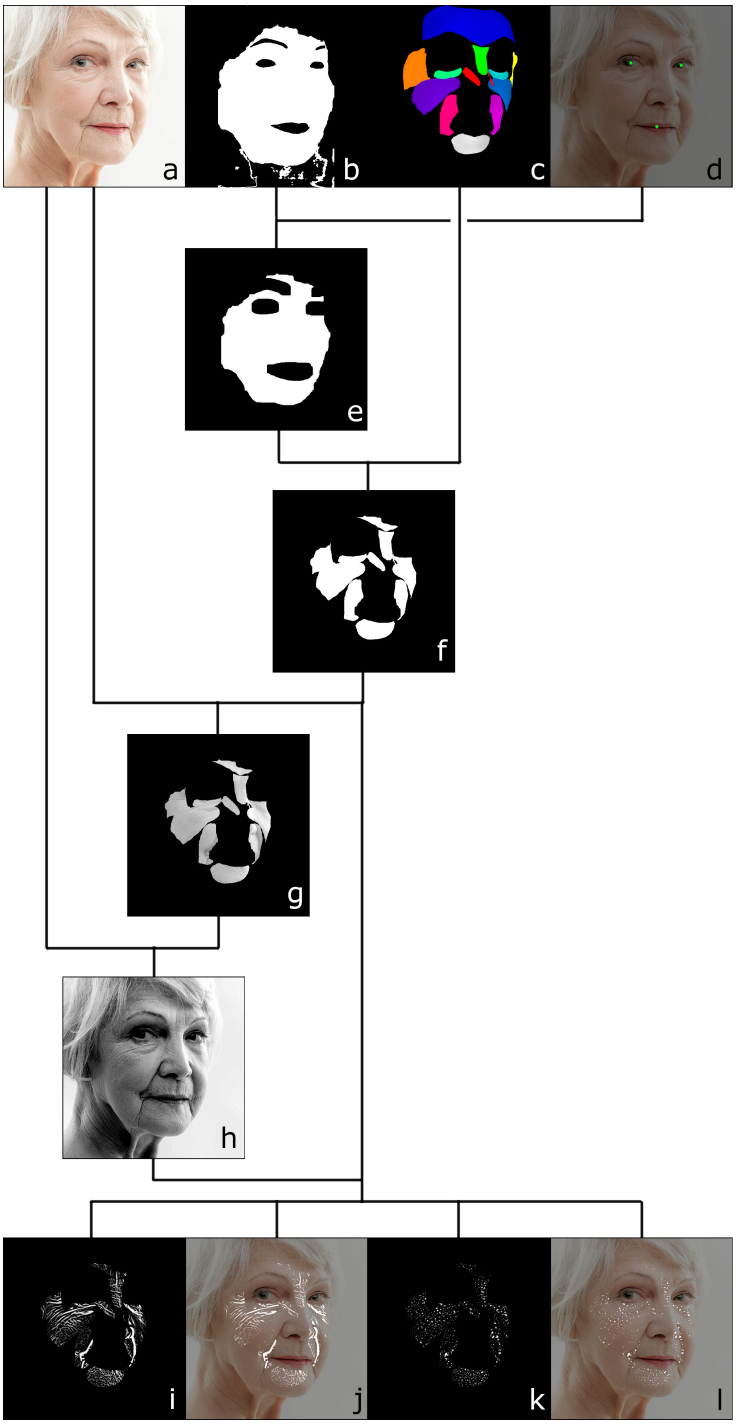

Figure 33: Inputs, intermediate outputs, and final outputs of the wrinkle and blemish detection pipeline. Input image from Yakobchuk (2021).

Figure 34: Result of detecting dark wrinkles on a light background, bright wrinkles on a dark background, and both cases overlaid. Duplicate detections can be observed when both are overlaid. Image from FFHQ dataset (Karras et al. 2019).

the C++ port, allowed for the detection both of dark features on a light background, as well as light features on a dark background, for both elongated and blob-like structures. In general, both wrinkles and blemishes are examples of the former: dark on light. However, under some circumstances, for example when the light falls perpendicular to the wrinkle direction, and the face is somewhat greasy, there may also be a bright line at the wrinkle location. While this may suggest that running the algorithm once for each may give more complete results, in practice it was observed that overlaying the two caused duplicate wrinkle detections, as can be observed in Figure 34. These duplicates are not exactly in the same place, but run parallel, and therefore are not removed when calculating the maximum of both detection outputs, and are non-trivial to remove through any other method. As a result, only the version for dark features on a light background was used for the entire remainder of this research.

As mentioned in the pipeline description, in order to maintain consistent wrinkle and blemish outputs, the scaling of the faces needs to be made similar before the detection filter is applied. To do this, the average keypoint position for the keypoints of each eye, as well as the mouth, are calculated. With these three average keypoints, the image is scaled so that the distance between the mouth keypoint and the midpoint between the eye keypoints is kept constant between images. Since this could cause a very large intermediate image if the face is somewhat far from the screen, the image is cropped to the result of the final region mask before it is scaled. The final wrinkle result is moved and scaled back to the crop area, so that it overlays properly on the original image. We note that, for simplicity and because it was deemed an acceptable constraint based on other wrinkle detection approaches in other beauty apps, the input face is expected to be facing forward, not be tilted, and not be too far from the camera,

so that the face fills the captured portrait image well. If this were to be extended to a real-time version (with the wrinkles being updated on every frame), this might no longer be a safe assumption.

On the Unity side, as shown in Figure 33, we needed to generate the segmentation mask, three keypoints for the eyes and mouth, and a mask for the regions of the face to be used. While the regions are shown color-coded, allowing for separate handling of each region if desired, the algorithm simply took all the regions together.

For the purposes of generating makeup overlays for beauty apps, the Unity and TensorFlow Lite framework had a system in which a morphable face mesh was fit to the face in real-time, and the overlays rendered onto it as textures. By creating an overlay mask texture and rendering it onto the face mesh, the region mask (image *c* in Figure 33) could be fit onto the input face, and the result used as a plugin input.

The keypoint and segmentation results were retrievable without such exploits, allowing all the required algorithm inputs to be provided to the plugin.

5.2.3 Estimation of skin scores

It was observed that other makeup apps, such as YouCam (PerfectCorp 2021), allowed for the calculation of certain skin scores, to gauge overall skin quality. Based on the wrinkle and blemish detection outputs, scores can be created for perceived face skin age, and face skin health, respectively.

As a result, our plugin was modified to allow for rough estimation of face skin age score and face skin health. The former was estimated by using the sum of the output wrinkle mask pixels (from Figure 33i) divided by the sum of the overall input mask pixels (from Figure 33f). When there are many or deep wrinkles (characteristic of old age or high skin age), the output becomes more dense and intense, increasing the aforementioned value. This results in a value between 0 and 1, that increases as the number and depth of wrinkles, and therefore also skin age, increase. Since a user would likely expect skin scores to increase for better skin, the value was subtracted from 1 to give the final skin age score.

The face skin health score was calculated in the same fashion, except this time using the blemish output mask instead of the wrinkle output mask.

To obtain a value for the perceived skin age itself, the skin age score values were plotted linearly against the ages, using the age values provided in the IMDb dataset. The linear relationship, while present, was relatively weak, but still allowed for a rough estimation of skin age.

It was observed that both the face skin age score and the health score tended to be overly close to 1 (above 0.9 in most cases). To rebalance the scores, an alternate

overall scoring system was devised, based on the position of the original scores in a percentile distribution, based on the same portion of the IMDb-WIKI dataset. This resulted in more user-friendly values.

An issue with both these scoring systems is their sensitivity to the σ and τ input parameters. A solution to this would be to run the algorithm on sufficient combinations of single σs and single τs to fit a multidimensional percentile distribution. For simplicity, the other variable input parameters (primarily the keypoint-relative image scaling) should be kept fixed.

In the absence of a machine learning solution, these were the tactics we used to score the face skin age and health. While the scoring methods presented are somewhat crude, the wrinkle and blemish detectors were not implemented in a public release app before the end of the project, so the scoring system was not refined or tested further. As future work, for face age, a simple network taking the output wrinkle mask and trained to output the age values could perhaps offer a more realistic estimate.

5.2.4 Quality of results

Using the Unity implementation to batch process the IMDb dataset subset and the FFHQ dataset for multiple σ and τ values, the outputs are found to be of acceptable quality for a method using only a filter, without line tracing for wrinkles. Some outputs are shown in Figure 35 for wrinkles and in Figure 36 for blemishes, with a close-up of a blemish output in Figure 37. Several common issues with the outputs are noted, and illustrated in Figure 38 for wrinkles and Figure 39 for blemishes.

Given the lack of a tracing step after the filter is applied, which helps filter out short, weak signals corresponding to natural skin texture, the wrinkle outputs are generally noisy. This becomes more noticeable in faces that are smooth, in which skin texture noise is predominant over stronger wrinkle signals, or faces that are badly lit or dark-skinned, since this increases image noise or compression artifacts after the normalization step. One possible tracing method is shown in a later step of Batool and Chellappa (2015), which uses a method similar to the Marked Point Processing model from Batool and Chellappa (2012a,b).

Output noise is even more pronounced in blemish outputs, as shown in Figure 39. As with wrinkles, the noise is more noticeable for badly lit, dark-skinned, and smooth faces. This could be mitigated somewhat by more careful tuning of the τ and σ values used. Further significant improvements are unlikely to be possible without resorting to machine learning methods.

Another issue is the noticeable overlap in the detected wrinkle output and the blemish output, as the paths around or among blemishes can appear vascular to the

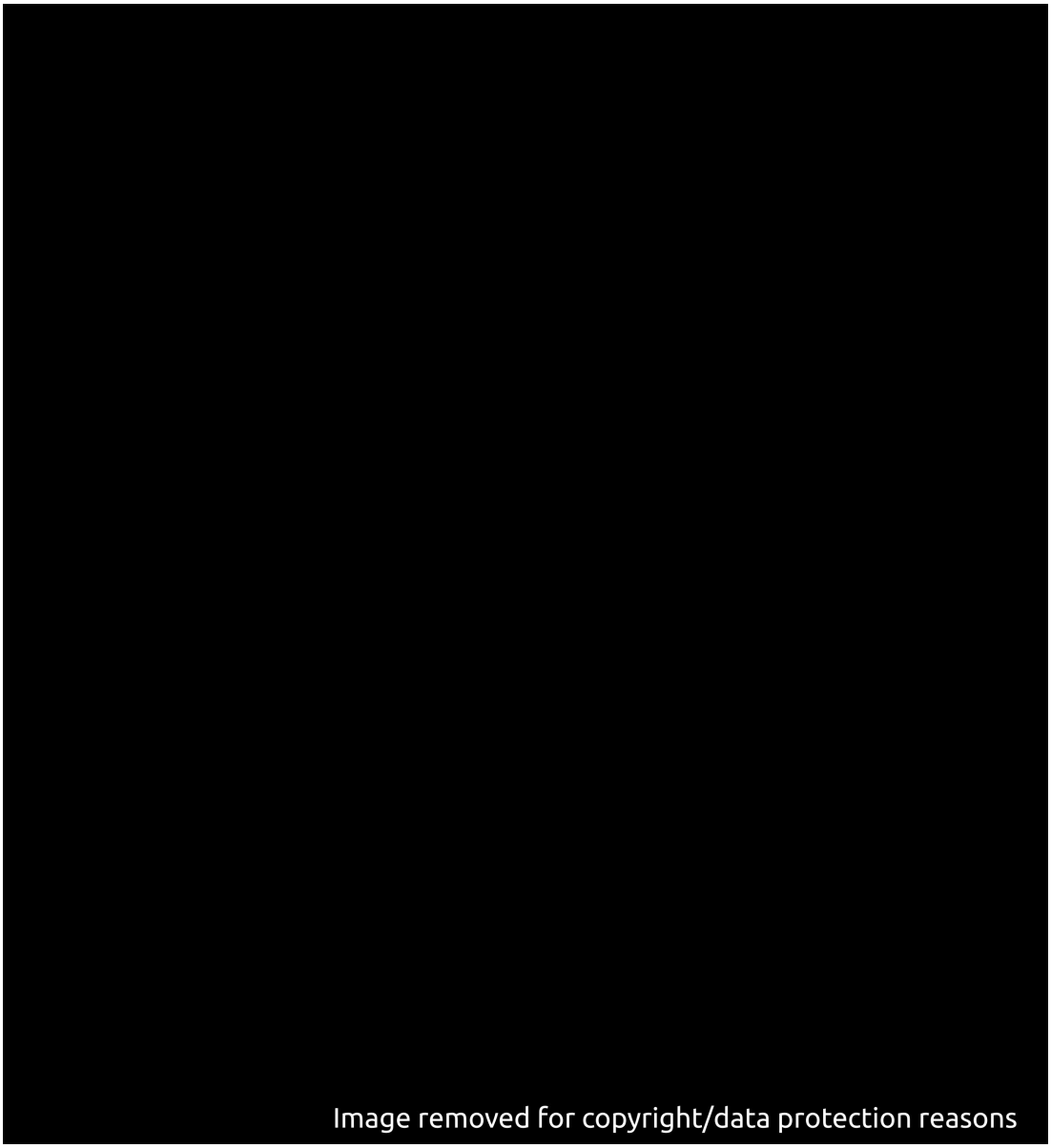

Figure 35: Wrinkle outputs using Unity implementation of Jerman's enhancement filter, for different values of σ, and $\tau = 0.5$. Images from FFHQ dataset (Karras et al. 2019) and IMDb-WIKI dataset (Rothe et al. 2018, 2015).

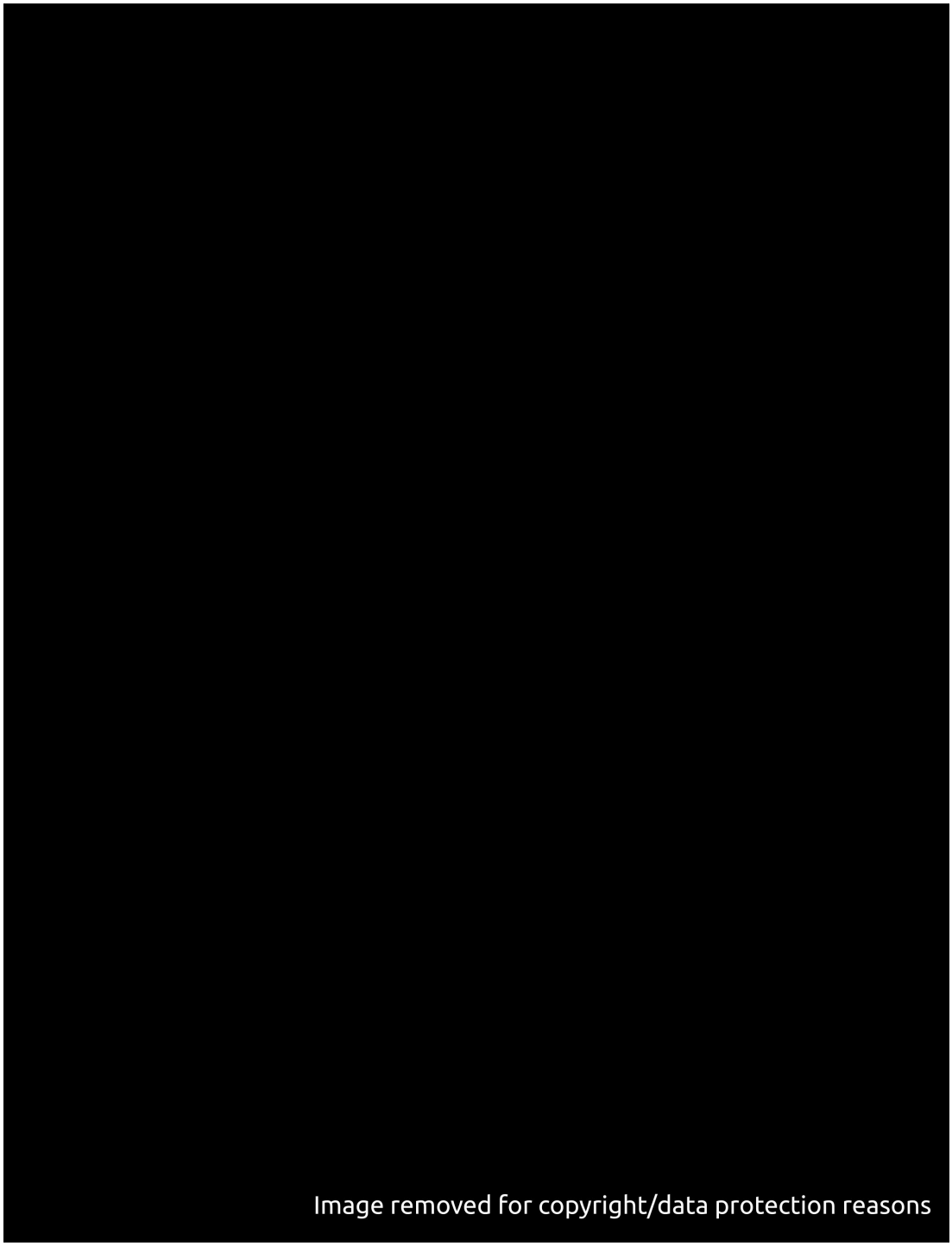

Figure 36: Blemish outputs using Unity implementation of Jerman's enhancement filter, for different values of σ, and $\tau = 0.5$. Images from FFHQ dataset (Karras et al. 2019) and IMDb-WIKI dataset (Rothe et al. 2018, 2015).

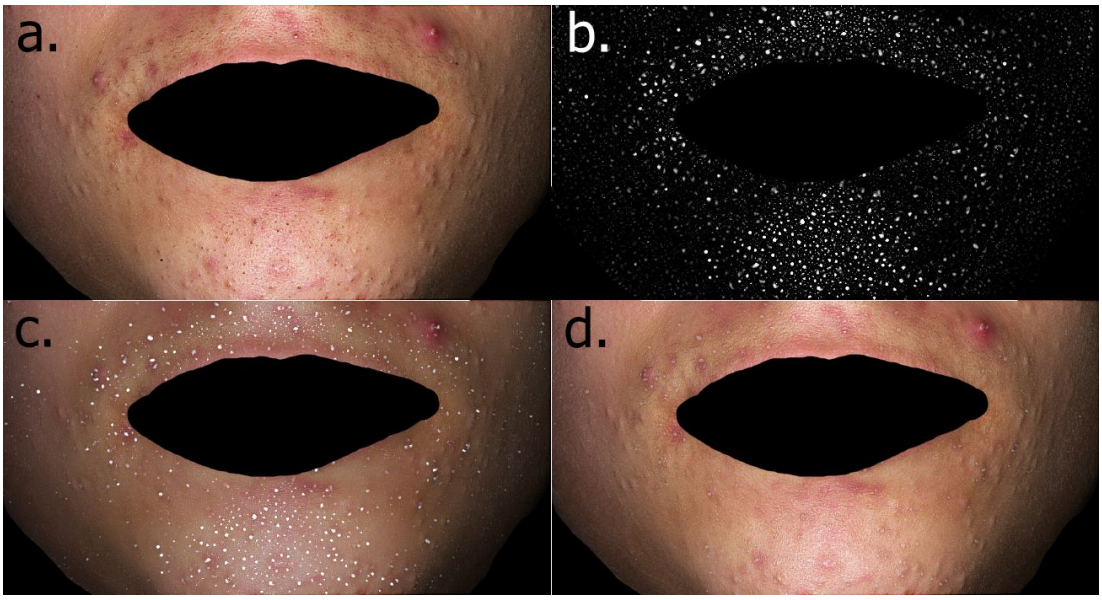

Figure 37: Output of Jerman's enhancement filter implementation in MATLAB, using only a portion of the face. a is the original image. Output b is the pixel-wise maximum of the outputs for multiple σ values (0.5, 1.0, 1.5, 2.0, 2.5), with $\tau = 0.5$. Output is overlaid on the original image in c abd d, with the latter showing how the overlay covers the blemishes (note the absence of the blackheads on the chin). Image from Cliff Dermatology (2021).

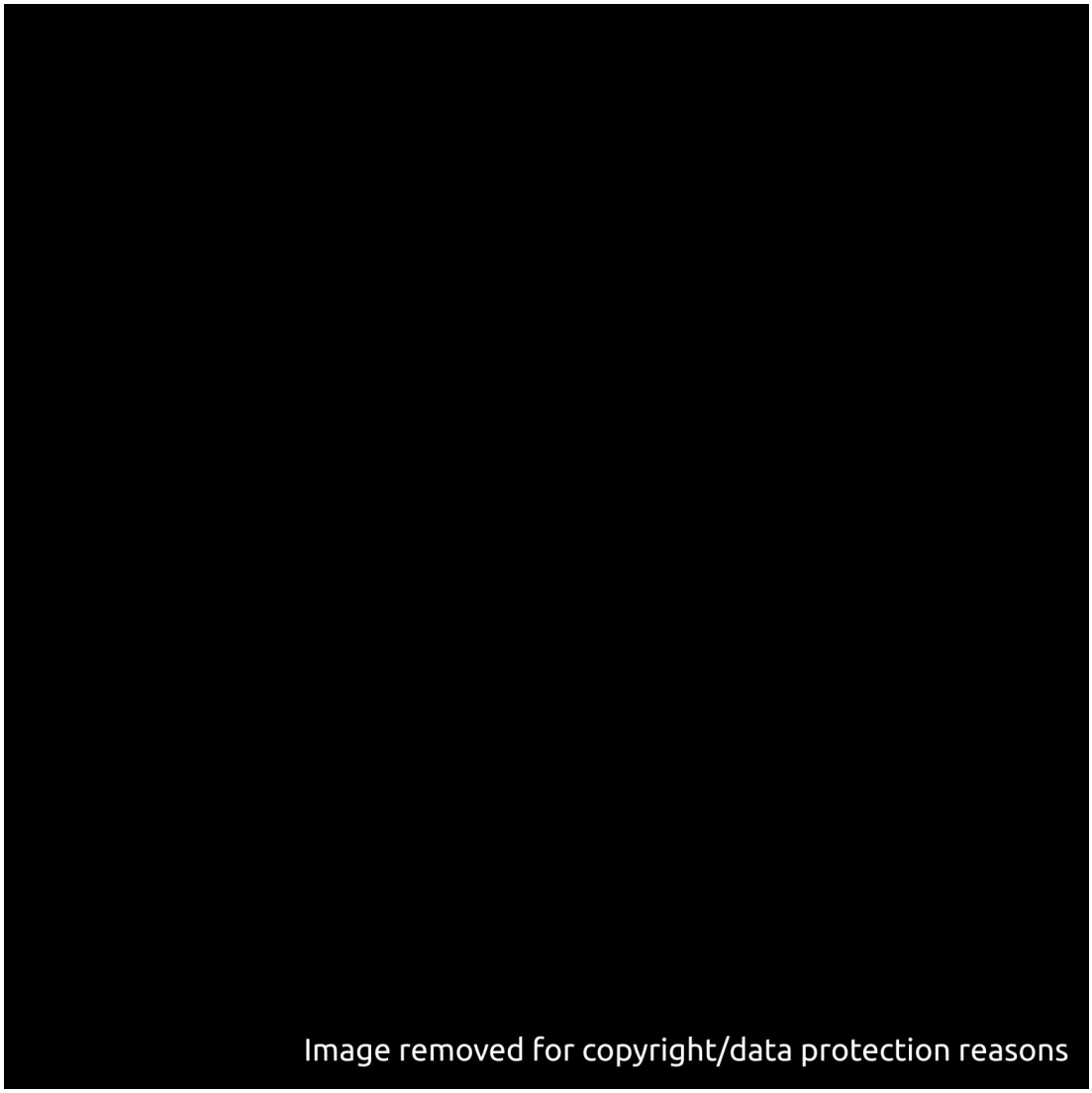

Figure 38: Wrinkle outputs with issues, for $\sigma = 1.5$, and $\tau = 0.5$. Outputs $a - c$ are noisy, due to input blurriness (a), and likely dark, low contrast inputs for b and c, although excess sensitivity may have been caused by the normalization in these cases. Outputs $d - g$ show issues due to occlusion from hair (d,e) or other objects (f and g, the latter being due to failure of the segmentation map). While the outputs in h and i are generally correct, the mask obtained from fitting the face mesh includes the chin, which is incorrectly marked as a wrinkle. This may however be due to the training dataset of the face mesh fitter not containing many child faces, in which case such failures should be rare during real-world use, where adult females are targeted. Images from FFHQ dataset (Karras et al. 2019) and IMDb-WIKI dataset (Rothe et al. 2018, 2015).

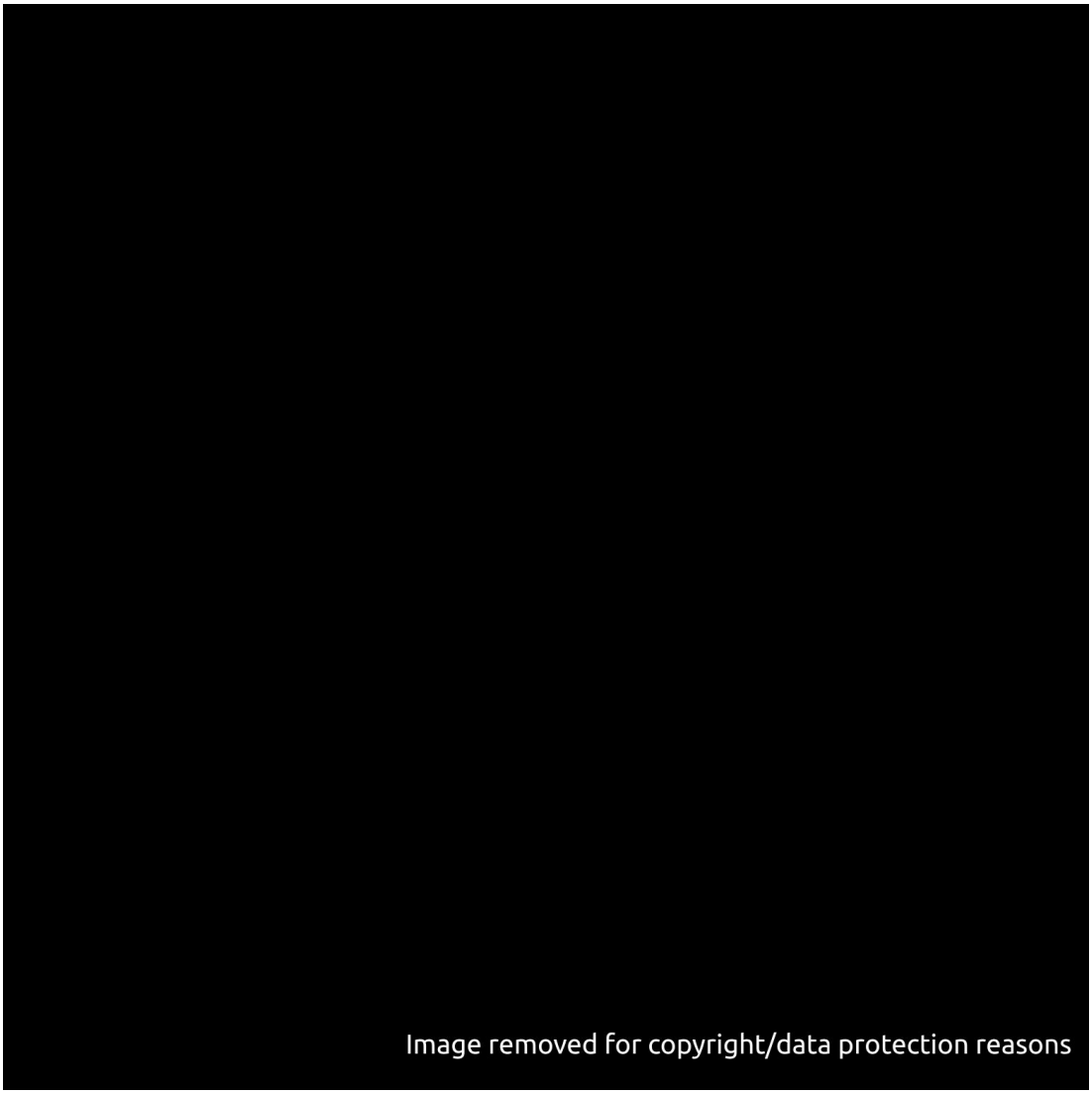

Figure 39: Blemish outputs with issues, for $\sigma = 2.5$, and $\tau = 0.5$. $a - c$ show masking issues, similar to the bad wrinkle outputs in Figure 38 (hat, occluding object, and lower chin, respectively). Outputs d and e show blemish outputs in regions with wrinkles (under the eye on the left side for d, and on the nasolabial folds on the sides of the lower nose in e). Present in most cases is the noise in the output, which is particularly noticeable in outputs $f - i$, where relatively smooth skin wrongly gives an intense output. We note the bad lighting case of h, and the dark skin case of i, which seem to worsen the noise. The noise could be to some extent corrected by increasing the τ value, but this value is difficult to tune well. Images from FFHQ dataset (Karras et al. 2019) and IMDb-WIKI dataset (Rothe et al. 2018, 2015).

algorithm, and corners or thicker points of the wrinkles can appear as blemishes to it. This is particularly noticeable for large blemishes like moles, which appear in the wrinkle output as ring shapes for all σ detection scales, as can be seen in Figure 38d. Since it is possible for both blemish areas to appear in the wrinkle output, and vice versa, attempting to remove false negatives through subtraction of one output from the other will likely cause some true positives to also get removed.

Using the postprocessing method proposed in Section 5.3, some of the noise for the wrinkle outputs could potentially be removed. However, as for the tracing, the method only applies to wrinkles, not blemishes.

5.2.5 General shortcomings of method and future work

This blemish detection algorithm cannot differentiate between different blob or point-like features. While some such features like acne and blackheads are generally deemed to be facial blemishes, freckles are arguably not. Such differentiation would probably take a machine learning approach. As mentioned in Section 2.7, the Jerman filter method detailed could aid in the annotation of a dataset tailored to this purpose, removing the need to tag every blemish, by allowing the human annotator to define large skin regions containing specific types of blob-like facial features, which would then be passed through the Jerman filter to give the final segmentation map for that region. This annotation tactic could also be used to create a wrinkle dataset.

As the method relies on the contrast between the dark wrinkles and the lighter surrounding skin, this makes handling of wrinkles on very dark skin difficult. While this is partially mitigated by the normalization method, the effectiveness on dark skin tones should be more explicitly verified, and the sensitivity of the filter calibrated differently in these cases if necessary.

As mentioned previously, the faces considered were only female, as the facial hair of some male faces showed up as detections for both the wrinkle and blemish filters. The beard could be excluded with a semantic segmentation network, but a simpler method could just use a classifier for beard presence, and restrict the wrinkle and blob detection regions to the upper part of the face, when the beard is present. A third method would be to interpret unusually dense detections on the lower part of the face relative to the upper part as the presence of beard hair, which could function as a beard presence classifier without using machine learning. All these possibilities, as mentioned, are outside the scope of this research, but could be addressed in future work.

5.3 Detecting and boosting wrinkles with Gabor filters and a direction expectation map

In searching for improvements for the Jerman enhancement filter method for wrinkles, we observed that while the amount and distribution of wrinkles on faces can vary greatly, the directions of the wrinkles on people's faces generally follow a known pattern. For example, the wrinkles on the forehead are generally horizontal, as are the wrinkles under the lower eyelid (the *tear troughs*) and those at the outer edges of the eyes (the *crow's feet*). We identify the various types of wrinkles in Figure 40, and mention their expected directions in Table 12. From this, we can generalize to expected directions for the entire face, and create a *direction map* for face wrinkles.

Table 12: General directions of facial wrinkles.

Region	General direction
forehead	horizontal, vertical above bridge of nose
frown lines	horizontal, vertical at top of region
tear troughs	horizontal
bunny lines	diagonal from tear duct to bridge of nose
nasolabial folds	diagonal, from nose to edges of mouth
crow's feet	horizontal
mental crease	horizontal
cheeks	rare wrinkles, no well-defined direction

Since long structures with specific orientations can be detected with Gabor filters (Batool and Chellappa 2015), the direction map can be used to boost wrinkles conforming to the map directions. This could aid in the removal of noise from the result, since most of the noise will not conform to the required direction.

Using a direction map was initially intended to be used as a potential postprocessing step for the Jerman enhancement filter-based wrinkle detector in Section 5.2. However, given the use of Gabor filter banks for wrinkle detection in Batool and Chellappa (2015), it was also looked into as a standalone wrinkle detection method. A Gabor *filter bank* consists of multiple filters with different parameter values, generally for orientation, with the pixel-wise maximum of the filter outputs giving the final bank output.

The theory behind Gabor filters, as well as an overview of their use for wrinkle detection in Batool and Chellappa (2015), is detailed below.

Figure 40: The relevant wrinkle regions. 1 - forehead, 2 - frown lines, 3 - tear troughs, 4 - bunny lines, 5 - nasolabial folds, 6 - crow's feet, 7 - mental crease, 8 - cheeks. Image from Travelwayoflife (2012) (licensed CC BY-SA 2.0), modified by cropping and adding an overlay showing the different wrinkle regions.

5.3.1 Background of Gabor filters for wrinkle detection

Gabor filters are dependent on the direction angle, and are defined as in Equation (5.7) (Fogel and Sagi 1989).

$$g(x, y; \lambda, \theta, \psi, \sigma, \gamma) = \exp\left(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}\right) \cos\left(2\pi \frac{x'}{\lambda} + \phi\right) \tag{5.7a}$$

$$g(x, y; \lambda, \theta, \psi, \sigma, \gamma) = \exp\left(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}\right) \sin\left(2\pi \frac{x'}{\lambda} + \phi\right) \tag{5.7b}$$

where

$$x' = x\cos\theta + y\sin\theta \tag{5.8a}$$

$$y' = -x\sin\theta + y\cos\theta \tag{5.8b}$$

In the discrete domain, these transform into Equation (5.9) (Ramakrishnan et al. 2002).

$$G_c[i, j] = B\exp\left(-\frac{i^2 + j^2}{2\sigma^2}\right) \cos(2\pi f(i\cos\theta + j\sin\theta)) \tag{5.9a}$$

$$G_s[i, j] = C\exp\left(-\frac{i^2 + j^2}{2\sigma^2}\right) \sin(2\pi f(i\cos\theta + j\sin\theta)) \tag{5.9b}$$

The B and C values are normalizing factors, which are here set to 1, with the image normalized to the float $[0, 1]$ range. By having a variable value of θ determined by the direction map at every position in the UV-space (2D surface space) of the face, or every point of a forward-facing face in image space, we can adapt this for use in our direction-based wrinkle detection approach.

Use of Gabor filters for wrinkle detection has been done previously in Batool and Chellappa (2015). Instead of using a coordinate-varying θ, this simply used a filter bank formed of an array of Gabor filters with different θ values, at equal intervals between 0 and 180 degrees. The output of the filter bank is the pixel-wise maximum of all the filters. This allowed for the detection of wrinkles along all directions, given enough filters.

The method in Batool and Chellappa (2015) subsequently applied a tracing step to determine the length and number of the wrinkles, and used a binary dilation step to remove double detections and congestion in the output. These steps are not done

in our implementation, as they appear to lead to high latencies (a reported average of 9 seconds), at least for images of reasonably high resolution.

The direction-based wrinkle detection is similar to the process used in Cula et al. (2013). This relied on a sparse direction map, similar to ours, and the same Gabor filters. However, it relied on controlled, polarized lighting, and estimated the direction map on the fly for each image, by choosing the direction orthogonal to the dominant direction of the Fourier spectrum, as in Lin Hong et al. (1998).

5.3.2 Gabor filter test for stray hair filtering

An initial test of Gabor filters for wrinkle detection is conducted for forehead wrinkles, using the code in Lessel (2021), written in C++, using OpenCV. Since forehead wrinkles are horizontal, we set θ to approximately zero degrees relative to the horizontal.

For the test image in Figure 41a, the directions of the hair strands are diagonal and vertical, so the response for a Gabor filter with a horizontal-relative θ of 0 should be small for the hair, and large for the forehead wrinkles.

This is shown to be the case in Figure 41. The top row shows the response for the input image a, when the Gabor filter angle θ matches the forehead wrinkle direction (b), and the Gabor filter output when θ matches the direction of diagonally oriented hair patch (c). The bottom row shows the result of our Unity implementation of Jerman's enhancement filter (d), the output of the forehead wrinkle-oriented Gabor filter on the Jerman filter output d (e), and the output of the hair patch-oriented Gabor filter on the Jerman filter output (f). On the top-right of the Gabor filter outputs (b, c, e, f), the Gabor filters themselves are shown, scaled 4x in both ther X and Y dimensions. In the filters, background gray corresponds to 0, black to -1, and white to 1.

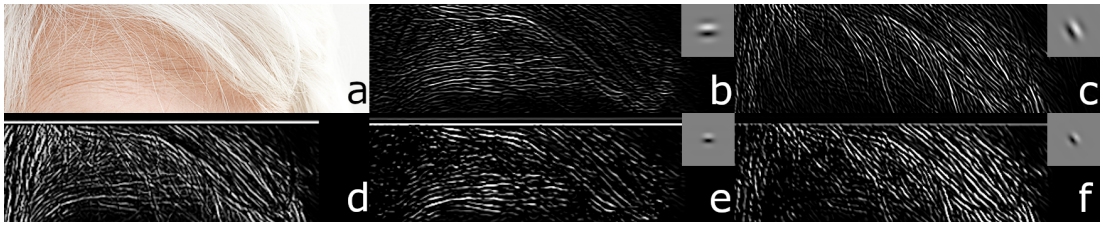

Figure 41: Gabor filter results, both on original image and Jerman enhancement filter result (top and bottom rows, respectively). Input image cropped from Yakobchuk 2021.

Both in Figure 41b and e, this is observed to limit the influence of hairs with angles different from our chosen θ. When the angle is changed to match the direction of a nearby patch of hair, this patch increases in intensity in the output instead (Figure 41c and f). This enables filtration of differently oriented elongated features.

While it is evident that this does not rid us of the need for the hair segmentation map, as much of the nearby hair patches will match the horizontal direction of the forehead wrinkles, it can mute the influence of stray hairs that are much too fine for any reasonable segmentation map to resolve. This has the drawback of some of the features of the Jerman-only result being lost, even when the wrinkle orientations match, especially if the scaling parameters of the Jerman and Gabor filters do not correspond.

This test illustrates the operation of fixed-angle Gabor filters. Use of variable angles based on a direction map is discussed below.

5.3.3 Implementation of Gabor filters with a direction map

Both in the Gabor filter test in Subsection 5.3.2, as well as in many placement company projects, implementation was done in C++ using the OpenCV library (Gary 2008). But in OpenCV it is nontrivial, and more computationally expensive, to create a filter that depends on the coordinates of the pixel in the image, as the $filter2D$ function of OpenCV used for Gabor filters requires a fixed kernel.

An implementation using a 180-filter bank is shown in Subsection 5.3.4, but it is too slow to be used in production. Two improved approaches are described below for the CPU and GPU, respectively, but not implemented as part of this pilot study.

A reasonably fast approximation of the direction map approach with OpenCV can be accomplished with a filter bank and a per-filter influence map. For each filter in the filter bank with angles in the set Θ, the outputs Y_θ are given by Equation (5.10), which are combined through the elementwise maximum. Depending on what is being implemented, the input X is given by the original image of the user's face, the UV-unwrapped face mesh texture, or the Jerman output of either two. For each filter, there is an influence map D_θ which has an effect analogous to the directionality map of the θ-mapped version.

$$Y(x,y) = Y_\theta(x,y) = \max_{\theta \in \Theta} D_\theta(x,y) Gabor(X(x,y), \theta) \tag{5.10}$$

An improved GPU shader version using a single Gabor filter per pixel, instead of a filter bank, is described by the equation (5.11). As inputs, this uses the input image or Jerman filter output X (potentially UV-unwrapped to a mesh texture), and the direction map Θ, generating the output texture Y.

$$Y(x,y) = Gabor(X(x,y), \Theta(x,y)) \tag{5.11}$$

5.3.4 Initial tests for Gabor filters with a direction map

As an initial test, the general directions are annotated manually, based on a single perfectly forward-facing, vertical, heavily wrinkled face, shown in Figure 40. This was created in Houdini (SideFX 2021), as this allowed for the live painting of a direction map to drive an array of arrows, for quick validation of drawn map updates, as shown in Figure 42a. The resulting direction map itself is displayed in Figure 42b.

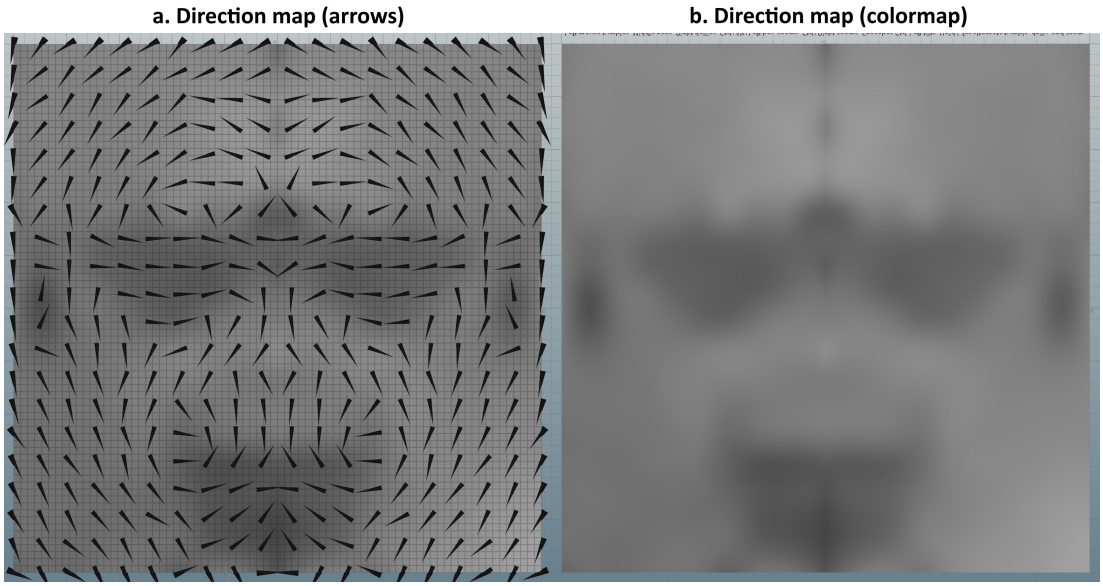

Figure 42: The direction map, shown as a mirrored array of arrows (a) and a scalar map (b), where black is 90 degrees and white is -90 degrees off the horizontal (since positive and negative directions are equivalent, this is the only relevant range). The arrow map is mirrored, so the right side has the positive angular direction being clockwise, and the left side counterclockwise.

A simple but slow way to implement the direction map is to generate the Gabor filter responses for a very large number of equally distanced θ values. Since the direction map gives the required θ value for each pixel, the pixel can be retrieved from the Gabor filter response with the closest θ. For a large enough number of responses, this gives a sufficiently close approximation.

This method was used in the test in Figure 43, for 180 values of θ ($\theta_i = i°$, for $0 \leq i < 180$).

While the result does generate the expected Gabor filter response (Figure 43b), it is comparable to the two-filter Gabor bank response (0° and 90°) in Figure 43c, and less good than the four-filter Gabor bank response (0°, 45°, 90°, and 135°) in Figure 43. As expected, the best result is obtained by using the bank formed of all 180 filters

(Figure 43e).

We note that due to the high number of wrinkles in the face image used, there are many wrinkles that do not follow the direction map, and a better result is likely for more common, less extreme cases. However, given its added complexity, for the method to give an improvement over simply using equally spaced Gabor filter banks, it would need to have comparable latency, or significantly reduce noise in the output.

5.3.5 Discussion

Given that the direction map-based Gabor filter approach was meant as a pilot study, it was not implemented or tested in its full intended form. This full method is described in the following subsubsections.

Addition of directionality map

The wrinkles on specific parts of the face do not always follow a single direction, which can cause the direction map-based approach to miss wrinkles in these areas. The degree to which wrinkle directions vary in each area can be mapped onto the face similarly to the wrinkle direction map, creating a *directionality map*.

Zones in which wrinkles generally follow only the direction specified by the direction map, like the top of the forehead and under the lower eyelids, are referred to as *strong directionality* zones. Some parts of the face do not have well-defined directions, either because there are normally no wrinkles on those parts of the face (like the cheeks), or because the wrinkles can appear along multiple directions, sometimes completely randomly. This is understood as *weak directionality*.

This directionality map could be represented as a 2D matrix of scalar values between 0 (weak directionality) and 1 (strong directionality). When the Gabor filter is applied as a post-Jerman filter effect, the directionality map can be used as a weighting variable between the Gabor and Jerman outputs. Since low directionality would cause any single Gabor filter to remove too much, a near-1 directionality value would favor Jerman. In the case of high directionality, the Gabor filter is more likely to filter out noise than wrinkles from the Jerman output, given the rarity of wrinkles that do not conform to the general direction in that region. In this case, the Gabor filter output would have a greater weight.

When not used as a post-processing effect, the directionality map could be used to inform a multi-filter variant of the direction map-based Gabor wrinkle detector. For a two-filter bank using angles $\theta_1(x, y)$ and $\theta_2(x, y)$, at a point (x, y) in the direction map where the required Gabor angle is $\theta(x, y)$, the bank angles would be $\theta_{1,2}(x, y) =$

$\theta(x, y) \pm (1 - d(x, y))\pi/4$, where $d(x, y)$ is the directionality at point (x, y).

Data-driven wrinkle direction and directionality map

To generate more thorough direction and directionality maps, a data-driven approach can be used. This can be done using a high-quality dataset like CelebAMask-HQ. After cropping and aligning the faces from the dataset using Dlib, the faces are restricted to a direction ± 5 degrees from the camera axis (so that they are forward-facing or close).

In order to determine both directionality and direction, there are two possible methods, both using all or part of the current Jerman-based Unity system for wrinkle detection. In the first method, the segmented and cropped face image is passed to a Gabor filter bank with a large number of filters, where if N is the number of filters, the ith filter is set to a θ angle of $\pi/N * i$. This is similar to the initial step of the wrinkle detection method in Batool and Chellappa (2015), but in this case, the filters with the highest response are used to determine the direction of the angle that will be used in the final filter, while the variance of the responses over all the filters determines the directionality.

The second method is similar, except that the input to the Gabor filter is the output of Jerman's filter, which is more appropriate when trying to use the Gabor filter map as a postprocessing effect. In both cases, in order to account for the differing widths and heights of faces, the facial keypoints detected by the face mesh fitter are used to squash or stretch the resulting Gabor outputs, to obtain universal face maps.

Extension to more face poses

While restriction to forward-facing faces is a limitation many similar beauty apps accept, and not only for wrinkle detection, it might nevertheless be considered limiting by the user. Furthermore, allowing for turned faces, perhaps even from profile, would better expose the wrinkles on the side of the face, allowing for multiple passes for a better map of the user's wrinkles. It should be noted that for our face mesh fitter and keypoint detectors, the greater the angle of the face from forward facing, the greater the errors of the fitter and detector. Since the wrinkle detectors rely on these, it may not be possible to properly handle turned faces through wrinkle detection algorithm improvements alone. Nevertheless, an extension of the Gabor methods is presented, which could allow for a greater array of face angles.

First, the wrinkle map must be changed from a plane to the surface of the face mesh. In order to apply the Gabor filter bank, this curved surface space must be mapped to a flat space independent of the face orientation. This is simply a UV space,

which can be constructed in Blender in such a way as to keep the mappings of equal areas in face mesh space roughly equal in this UV space. This requires that the face image be UV-projected onto the face mesh surface. Since this is not trivial to code as a shader in Unity, it can first be implemented using the Blender Python API, merely for the direction and directionality map construction. Given the reduced resolution of the skin areas not parallel to the camera plane (like the sides of the face in the case of forward-facing faces), these can be excluded within some tolerance by using the dot product between the mesh normals and the camera plane normal, and using a low-pass threshold.

Given the limitations of the face mesh fitter, the fitting and segmentation can be verified manually for the general case with multiple face angles. Additionally, the output of Jerman's enhancement filter was verified manually in both the forward facing face variant, as well as the general case.

Implementing the UV projection step in Unity would also enable the creation of a wrinkle texture tracked to the face, implemented similarly to the virtual makeup try-on textures.

GPU implementation

Since the CPU implementations with OpenCV may be slow, they could be implemented on the GPU as in Equation (5.11). While this could be implemented using a custom TensorFlow Lite GPU operation, as in Appendix B, implementation using Unity HLSL shaders would likely be less complex. This removes the requirement of implementing the custom GPU operations in OpenGL, OpenCL and Metal, as required for TensorFlow Lite GPU, since Unity translates the shaders internally to the languages required.

5.4 Ethical considerations

In terms of fairness in output quality, accuracy was again observed to be lower for darker-skinned individuals, due to the lower contrast between blemishes or wrinkles and normal skin. This could be to some extent mitigated by using an improved normalization system and requesting that the users use good lighting for the input face image. However, it is expected that lower contrast due to dark skin tone will always make wrinkle and blemish detection work slightly worse.

As for the face makeup transfer project, the wrinkle and blemish detection was intended for use in the placement company's beauty apps, which was also intended to make beauty product purchase recommendations, and tune a makeup style recommendation engine. For this project in particular, as many beauty products have the

purpose of hiding wrinkles and blemishes, this system could easily be used to exploit user's insecurity about their appearance in order to drive product sales. As for the face makeup transfer project, the author of this thesis was not involved in these aspects of the apps, so no additional ethical analysis can be made.

Given that all methods presented are very specifically aimed at detecting wrinkles, they have no obvious applications beyond this. However, the detection of wrinkles could have applications outside beauty, for example for driving skin deformation in face rigs for virtual characters, to ensure more human-like facial expressions.

5.5 Conclusions

The Jerman enhancement filter method was shown to give acceptable results for both wrinkles and generic blemishes, but particularly the former. This resulted in the implementation of both detectors in Unity with a C++ plugin. The blemish detector may however have limited application in beauty apps, given its inability to distinguish between different blemish types. A fully machine learning-based approach would be required in this case.

The Gabor filter and direction map method for wrinkle detection was not implemented beyond initial limited tests. The benefits appear to be limited both when used as a postprocessing step for the Jerman filter output, due to the dampening of valid signals, as well as when used as a separate method, due to the increased implementation difficulty and limited benefit over using normal Gabor filter banks, as in Batool and Chellappa (2015). While several future improvements have been suggested, like the directionality map and extension to more face poses, these will further increase the implementation difficulty.

As mentioned in 5.2.5, greatly improved results could be obtained with machine learning if the suitable datasets were available, particularly in the blemish case, and the classical computer vision methods discussed could significantly aid in the annotation of such datasets. This is likely the most valuable future work that could be undertaken based on the research in this chapter.

Figure 43: Results using Gabor filter with direction map, compared to several Gabor filter banks. Original image is shown in a, with the direction map response shown in b. Images c, d and e show the response using Gabor filter banks for $\theta_i = i\pi/N$, $0 \leq i < N$, for $N = 2$, $N = 4$, and $N = 180$, respectively. Base image from Travelwayoflife (2012) (CC BY-SA 2.0), modified by cropping and scaling, and shown with different outputs.

Chapter 6

Hardware-aware Neural Architecture Search for Segmentation in Mobile and Web Apps

As has been observed previously, one of the most challenging aspects of designing networks for use in smartphones is ensuring fast inference, particularly when the models need to operate in real-time, as in many of our use cases. This was further exacerbated by the placement company's desire to offer some of the machine learning functionality of the Unity and TensorFlow Lite system (Chapter 2.4) as part of a web app, capable of functioning on both desktop and smartphone browsers. In this case, TensorFlow Lite cannot be used, but TensorFlow instead allows for inference in web browsers through the TensorFlowJS library (Smilkov et al. 2019). While this also allows for hardware-accelerated inference using WebGL and potentially WebGPU, we observed that the inference speeds, particularly on mobile web browsers, were too slow for real-time operation. This therefore required the design of even faster versions of many of our machine learning models, while sacrificing as little accuracy as possible. As a result, some experimentation was conducted with neural architecture search, using latency and accuracy awareness.

Neural architecture search (NAS) aims to automate the development of improved neural networks by allowing the network architecture to vary, in addition to the weights. The space formed of all possible architectures of the scheme is "searched" through to find the best one, by means of some optimization process. By relying on previously discovered optimal characteristics and building blocks of manually crafted networks, the

prohibitively large space of possible neural network architectures can be significantly reduced, making the search task feasible. This generally leads to a trade-off between search simplicity and likelihood of finding novel, improved architectures.

As mentioned in Section 2.8, neural architecture search methods using reinforcement learning are too computationally intensive to be attempted without access to a significant amount of powerful hardware, which made it unfeasible for use here. As a result, the current most popular approaches, due to their much shorter search time, involve selecting a differentiable search space, so that the search process is similar to the traditional training of a fixed network.

Since it is generally desirable for networks to be both accurate and lightweight, many neural architecture search methods attempt to both increase accuracy and keep the FLOP or MAdd count low (Chu et al. 2020; Lu et al. 2019; Wan et al. 2020). Despite being commonly used as a proxy for low latency and low memory usage, the FLOP count has been observed to be a poor estimate of the former (Wu et al. 2019; Ma et al. 2018). Especially in the case of mobile phone inference, the speed performance of a model is heavily dependant on the hardware and software executing the model.

Given all the observations on available methods in 2.8, the requirement for the base of our neural architecture search efforts is a differentiable architecture search with latency awareness, using a fast but strongly correlated on-device latency proxy.

6.1 Background of the FBNet neural architecture search method

The work in this chapter is based on FBNet (Wu et al. 2019), which is a NAS method for generating classifiers. This uses a relatively small search space, by restricting the candidate models to a sequence of 26 blocks. 22 of these are sampled from 9 candidate blocks, one of which is a no-op (output equal to the input), which allows unneeded or suboptimal blocks in the sequence to be skipped. To avoid having to measure the latency of every candidate architecture, FBNet assumes that the latency of the candidate models is a monotonous function of the sum of the block latencies, which it shows to be generally true in practice. Therefore, to optimize for on-device latency, only the unique blocks need latency benchmarking on the target device. The block latencies are then retrieved by the FBNet scheme at search time, by using a lookup table.

In addition to being differentiable, FBNet is also a one-shot neural architecture search scheme. In one-shot NAS, the search space encompasses all possible subgraphs of a supergraph, and allows for a rough estimate of the efficiency of the possible subgraphs

by simply training the supergraph. This is quite efficient in reducing both the search space and the memory requirements, as for any two subgraphs, any identical portions use shared weights. After the training of the supergraph, the subgraph with the highest overall efficiency is selected as the search result, and is trained until convergence.

At each search step, FBNet uses a single candidate block from each cell, randomly sampled with a certain sampling probability. This probability is given by the softmax function, as in Equation (6.1), where $\theta_{l,i}$ is a value associated with block i of layer l, that is optimized over the search period to be larger for the best blocks. When used for random sampling, this results in a cell output tensor x_{l+1} given by Equation (6.2), where m_l is a one-hot vector indicating the sampled block for cell l, and b_l is the array of block functions, taking the input tensor x_l. Since for a single training step, the value of $m_{l,i}$ is 0 or 1, which makes it discrete, the loss function is not differentiable. FBNet overcomes this by replacing $m_{l,i}$ with a randomly sampled variable from a Gumbel-Softmax distribution, which is a continuous, differentiable function (Equation (6.3)).

$$P_{\theta_l}(b_l = b_{l,i}) = softmax(\theta_{l,i}; \theta_l) = \frac{exp(\theta_{l,i})}{\sum_i exp(\theta_{l,i})} \tag{6.1}$$

$$x_{l+1} = \sum_i m_{l,i} \cdot b_{l,i}(x_l) \tag{6.2}$$

$$m_{l,i} = GumbelSoftmax(\theta_{l,i}|\theta_l) = \frac{\exp[(\theta_{l,i} + g_{l,i})/\tau]}{\sum_i exp[(\theta_{l,i} + g_{l,i})/\tau]} \tag{6.3}$$

Since the search scheme is differentiable, it optimizes for both on-device latency and accuracy by summing the latency and accuracy losses, and doing the supernetwork backward pass. The latency loss l_t is defined as in Equation (6.4), where α and β are constants, by default equal to 0.2 and 0.6 respectively, and T is the sum of on-device latencies of the blocks in the current candidate architecture. The values of α and β can be modified to control the trade-off between low latency and accuracy in the final architecture.

$$l_t = \alpha(\log(T))^\beta \tag{6.4}$$

Generally, a search epoch is formed of two stages, one which optimizes the block weights, followed by a shorter stage that optimizes the θ values of the blocks. These θ values work as an optimization score for the block, and determine the block sampling probability, as in Equation (6.1). For the first zero or more stages, only the first stage is done, as this may result in better θ convergence in later epochs.

After the search converges, the block with the highest θ value is selected from

each layer, resulting in a fixed, optimized classifier architecture, which is trained until convergence.

6.2 Experiments with FBNet variants for segmentation

Since the paper-faithful implementation of FBNet targets classification tasks only, we make modifications to search for and train a segmentation network, partially based on an existing non-NAS lip segmentation network used in the virtual try-on, from the Unity and TensorFlow Lite system from Chapter 2.4. This requires a segmentation decoder, either fixed or formed of searchable cells, as well as UNet-style skip connections (Weng et al. 2019) between the encoder and decoder. We tried several different approaches, detailed in the following sections. The results are compared to the non-NAS lip model, which had the same decoder as the *long encoder, long fixed decoder* model from Subsection 6.2.2, and an encoder of similar length to it.

The code was based on the FBNet implementation from Jh88 (2019). We first describe the non-NAS segmentation model used for comparison in the next subsection, followed by the FBNet-based NAS variants in the later subsections.

6.2.1 Non-NAS model architecture and training

The non-NAS mouth segmentation models were not developed by the author of this thesis, by by colleagues at the company, and are presented only for comparison with our new NAS models. Aside from initial variants, these non-NAS models employed the same strategies as MobileNetV2 (Sandler et al. 2018) to optimize for rapid mobile operation: depthwise separable convolutions and residual network connections. Being segmentation models, they also used a UNet-like structure with skip connections (Ronneberger et al. 2015), as shown in Figure 44.

Several variants of the architecture existed, most notably a long version of input size $(1, 96, 192, 3)$, and a short version of input size $(1, 48, 96, 3)$, although only the former had been observed to give good results. As a result, the larger input size model is used for comparisons with the FBNet-based architecture variants in the next subsections.

This model was trained with Dice loss, which was first implemented for semantic segmentation in medical images by Milletari et al. (2016), as it was shown to give better results than the more commonly used pixel-wise cross-entropy loss. This Dice loss is based on the statistical sample similarity metric known as the Sørensen-Dice coefficient, developed in the 1940s for biology applications (Dice 1945; Sorensen 1948). For binary data, in which the ground truth and predicted segmentation mask pixels

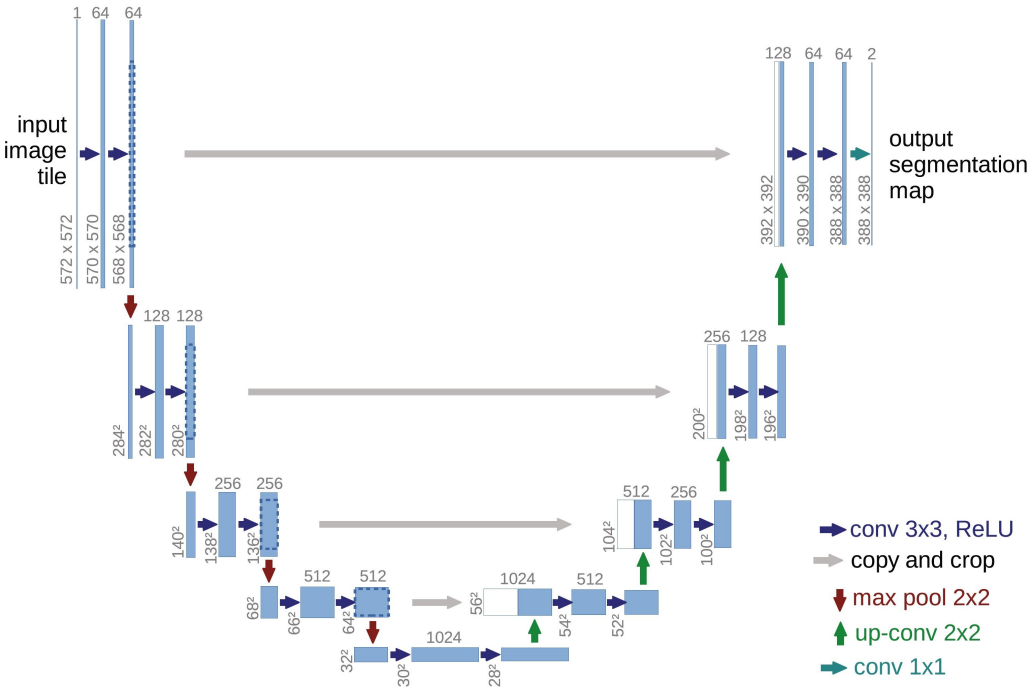

Figure 44: The original UNet architecture. The skip connection (here *copy and crop*) concatenates the max pool input and the up-conv output along the channel axis. Later adaptations and the architecture used in this chapter do not crop the skip connection input. Figure from Ronneberger et al. (2015).

can be considered as sets, it is defined as double the cardinal (number of elements) of the intersection of these sets, divided by the cardinal of the union of the sets. When extended to pixels with associated class probabilities, it is defined as the pixel-wise multiplication of the ground truth class probabilities (usually binary, with one and only one true class per pixel) and the corresponding prediction probabilities, divided by the sum of the pixel-wise squares of each. This is given by Equation (6.5), where i covers the height, width, and class dimensions. Pixel-wise cross-entropy loss for multiple classes is shown in Equation (6.6) for comparison.

$$D = \frac{2\sum_i^N p_i g_i}{\sum_i^N p_i^2 + \sum_i^N g_i^2} \tag{6.5}$$

$$CE = -\sum_{i}^{B,H,W} \sum_{c=1}^{M} y_{i,c} log(p_{o,c}) \tag{6.6}$$

This use of Dice loss was also maintained for our NAS methods, for the training of both the supernet (the architecture search step), as well as the fine-tuning of the final chosen architecture. The description of these methods follows in the next subsections.

6.2.2 Partial FBNet encoder with fixed decoder

Given that the implementation of a searched decoder required more changes than modifying the encoder for segmentation, our first attempt retained the decoder of a short variant of the existing non-NAS network. There was an initial desire to use the decoder from the segmentation variant of MobileNetV3 (Howard et al. 2019), but the effective output resolution was too low for our purposes, so the existing non-NAS lip segmentation models of the company were used instead.

This was appended to part of the backbone of the FBNet classifier (the first 6 blocks), which acted as an encoder. We also added UNet-like skip connections (Figure 44), as required for segmentation networks. Since the decoder is from a non-NAS model, formed of fixed blocks, only the encoder is searched. The resulting supernet architecture is referred to as *short encoder, short fixed decoder*.

As for the non-NAS model, a rectangular input size was used, corresponding to a resized crop of the lip region of RGB face images. We used the CelebAMask-HQ dataset (Lee et al. 2020) for training and testing. Input sizes of 48×96, 96×192, and 144×288 were used, which we refer to as small, normal and large, respectively. Former non-NAS model variants using the small size had not been deemed to have enough output quality to be used as candidates for production. Similarly, our FBNet-based segmentation (FBNet-Seg) architectures also produced unsatisfactory results at this input size, as can be seen in Figure 45.

To generate the latency tables, all the unique blocks of the supernetwork were converted to the TensorFlowJS and TensorFlow Lite formats, using the automatic conversion capabilities of TensorFlow Lite. The TensorFlowJS blocks were benchmarked in Firefox on a mid-2015 MacBook running MacOS Catalina, using a version of the official TensorFlowJS local benchmarking utility (Google 2021m), which we modified to operate on batches of models. The TensorFlow Lite blocks were benchmarked on the GPU of a OnePlus 6 Android smartphone, using the OpenCL delegate, and the official benchmark utility (Google 2021l).

Following our initial few training runs for the small and normal variants, and the subsequent on-device TensorFlow Lite latency results, we observed that the latency awareness in the training did not have a significant enough contribution to justify the observed accuracy hit it caused. For example, in the case of the normal-sized model, the latencies of the latency-aware and latency-unaware *short encoder, short fixed decoder*

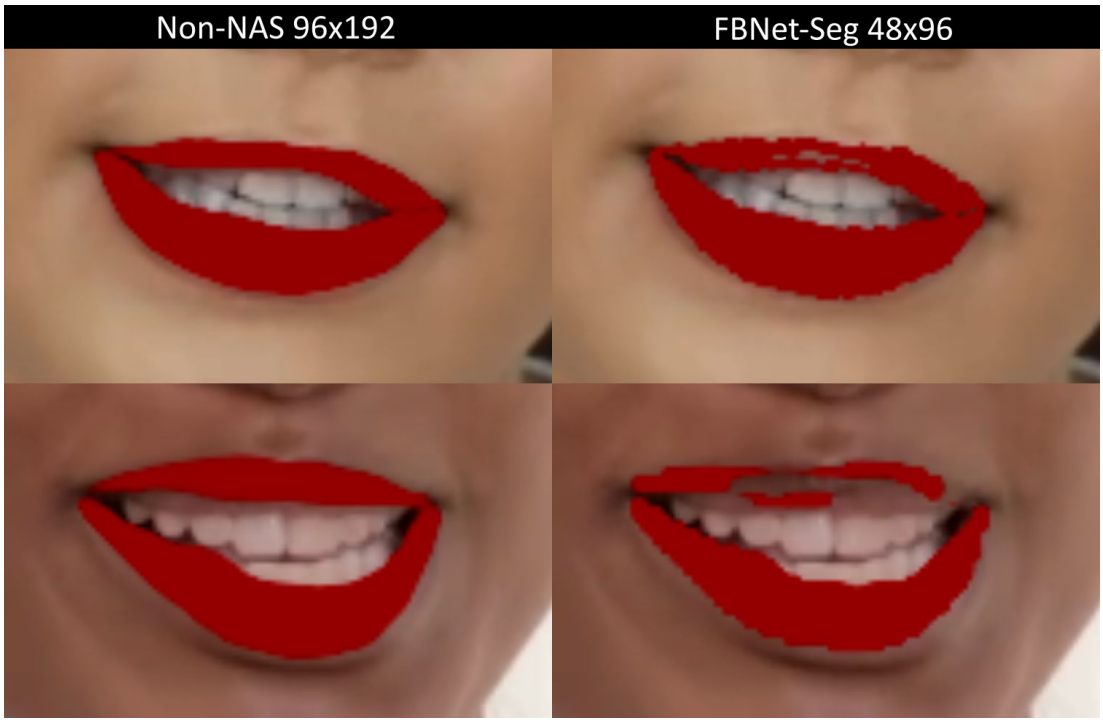

Figure 45: Comparison between non-NAS model of input 192×96 (the minimum quality target) and segmentation FBNet model of input 96×48 with fixed decoder and searched encoder.

models varied between 4 and 5 ms, while the reference non-NAS network had a latency of 9.6 ms. As a result, and given that the latency loss was observed to make up a significant portion of the overall loss, latency awareness was disabled by removing its associated loss, as to not affect the accuracy of the architecture chosen by the search.

After the latency loss was removed, the outputs of the resulting 192 by 96 model were comparable to but slightly worse than the non-NAS model. This was judged mainly through visual inspection on unseen data, as shown in Figures 47 to 50.

Since the resulting 192 by 96 model was significantly faster than the non-NAS model of the same input size, due to it being less deep, the large 288 by 144 input size was also tested. The latencies of the resulting models were still smaller, but comparable to the non-NAS model. These models unexpectedly gave worse performance when tested on unseen data with the Unity and TensorFlow Lite system, when visually judging the outputs. This is deemed to be due either to bad preprocessing in the training stage, either to the mouth crop in the training dataset being smaller than the input size, both of which would lead to blurriness due to upscaling. Since the frame provided by the Unity and TensorFlow Lite system during live use has a higher resolution, resulting in less or no upscale blur, this would cause a higher discrepancy

between training and real-world use than in the case of the normal 192 by 96 input shape model. This was not investigated further due to time constraints, and would have likely required a higher-resolution training dataset to mitigate, which was not available.

Two other fixed-decoder supernet variants were constructed with a slightly different architecture: one with a longer encoder, named *long encoder, short fixed decoder*, and one with both a longer encoder and a longer decoder, named *long encoder, long fixed decoder*. Given the results using the small and large input sizes, only the normal 192×96 input size was tested for these architectures. The *long encoder, short fixed decoder* model was found to be a slight improvement over the *short encoder, short fixed decoder* model, but not enough to be better than the non-NAS model. The *long encoder, long fixed decoder* model, on the other hand, was found to be a downgrade, both in general, as well as for difficult cases (as in Figures 49 and 50).

The numerical results for the final sampled models, for all tested fixed-decoder supernet architectures, are shown in Table 13. The visual results in Figures 47 to 50, showing the Unity test output, are however a more realistic gauge of the quality of the models in real-life use. These reflect the superiority of the non-NAS model, followed by the normal 192 by 96 *long encoder, short fixed decoder* model, followed by the 192 by 96 *short encoder, short fixed decoder* model. This was verified on multiple test videos, by multiple people.

The smoothness of the output of the non-NAS model is unusual when compared to the FBNet-based models, which have much sharper segmentation mask edges (Figures 47 to 50). It is possible that despite attempts to retain the preprocessing code as much as possible, it underwent changes between the training of the non-NAS model and the incorporation of the code into the FBNet-Seg implementation. While this could be more exhaustively verified by retraining the non-NAS models with the same preprocessing code, this was not done due to time constraints.

6.2.3 Partial FBNet encoder with reversed encoder as decoder

Given the insufficiently good results when using a fixed decoder, the decoder was modified to also be searchable. This was expected to result in a more precise model being found by the search. While the associated supernet architecture comes at the cost of a longer search, having about twice as many searchable cells, it still has fewer cells than the original FBNet, which has a relatively low search time. The architecture is referred to as *long encoder, long searched decoder*, and was constructed by using largely the same blocks as the encoder, but in reverse order.

In order to approximate the different types of lip crops that occur in the Unity and

TensorFlow Lite system that the lip network was meant to be used in, the preprocessing step used for all FBNet-Seg versions and the non-NAS lip model allowed for multiple zoom levels for the lips, as shown in Figure 46. As some zoom levels were assumed to be rare in practice, and it is easier for the model to support fewer zoom levels, the lower ones were removed in the *long encoder, long searched decoder* searches, to try to increase the average accuracy. However, this visibly led to a significant number of failure cases during testing on unseen data (as in Figure 48), seemingly indicating that the removed zoom levels occur often enough in practice to be important. As a result, while the output was otherwise comparable to the more accurate fixed-decoder versions, the failures caused it to be inferior overall. The model could not be retrained with the original scaling factors due to time constraints.

Figure 46: Output for the zoom levels used for the non-NAS and fixed decoder models. The 64 and 128 levels were disabled for this version (in which both the encoder and decoder were searched). Augmentations were disabled only during the creation of this example image. Original image from CelebaMask-HQ dataset (Lee et al. 2020).

6.2.4 Overall results

The accuracies and TensorFlow Lite latencies of all the best search results for each supernet variant are shown in Table 13. The success of the models depended mainly on the accuracy on unseen data, so this was visually evaluated on several videos, including two with difficult dark skin tones. Some more notable visual results are shown in Figures 47 to 50.

As previously mentioned, the non-NAS model showed the best output overall, followed by the 192 by 96 *long encoder, short fixed decoder* FBNet-Seg model, and then the 192 by 96 *short encoder, short fixed decoder* FBNet-Seg model. While the superiority of the non-NAS model seems to be partially due to an alternate preprocessing mechanism during training, its segmentation mask appears to be the best even beyond this benefit. This indicates the general failure of the FBNet-Seg methods attempted.

Table 13: Accuracy and latency results for the various FBNet-Seg NAS models, and the existing non-NAS model it was compared to. The latter had a long decoder, similar to the cell length of the long encoder from the NAS models. CPU latencies use XNNPACK and 4 threads, without quantization, and GPU latencies use the OpenCL delegate, both being timed on a OnePlus 6 Android device.

Encoder	Decoder	Input size ($H \times W$)	Accuracy	mIOU	Latency GPU/CPU (ms)
short	short, fixed	48×96	80.59%	34.42%	2.56/1.42
short	short, fixed	96×192	77.19%	38.71%	4.14/5.00
short	short, fixed	144×288	78.17%	39.22%	7.56/13.04
long	short, fixed	96×192	78.23%	38.73%	5.11/6.76
long	long, fixed	96×192	72.44%	30.50%	10.52/10.62
long	long, searched	96×192	68.86%	36.56%	15.55/16.48
fixed	long, fixed	96×192	78.14%	39.20%	9.72/8.61

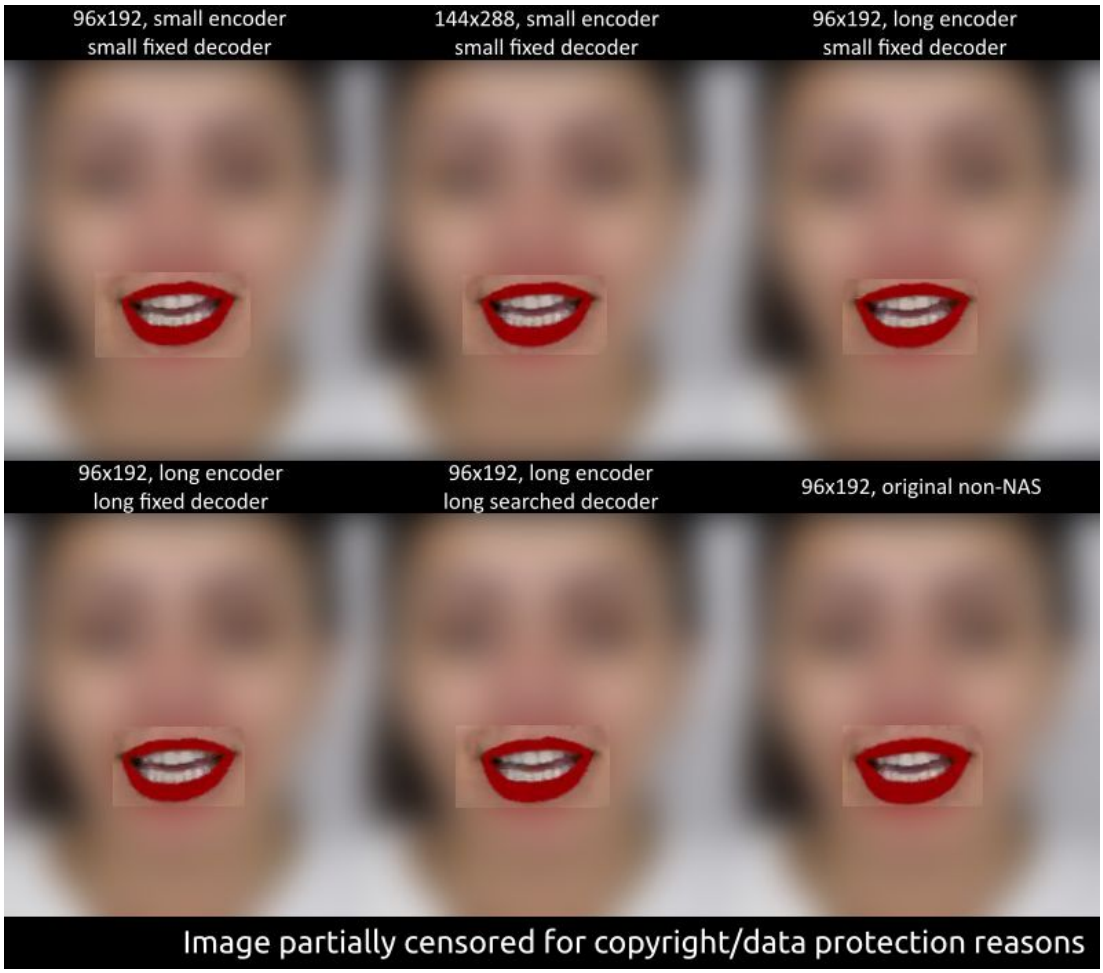

Figure 47: Results on one frame of the *easy* video, showing that all models work well in general. 48×92-input short encoder, short fixed decoder is not shown, due to excessively low resolution output.

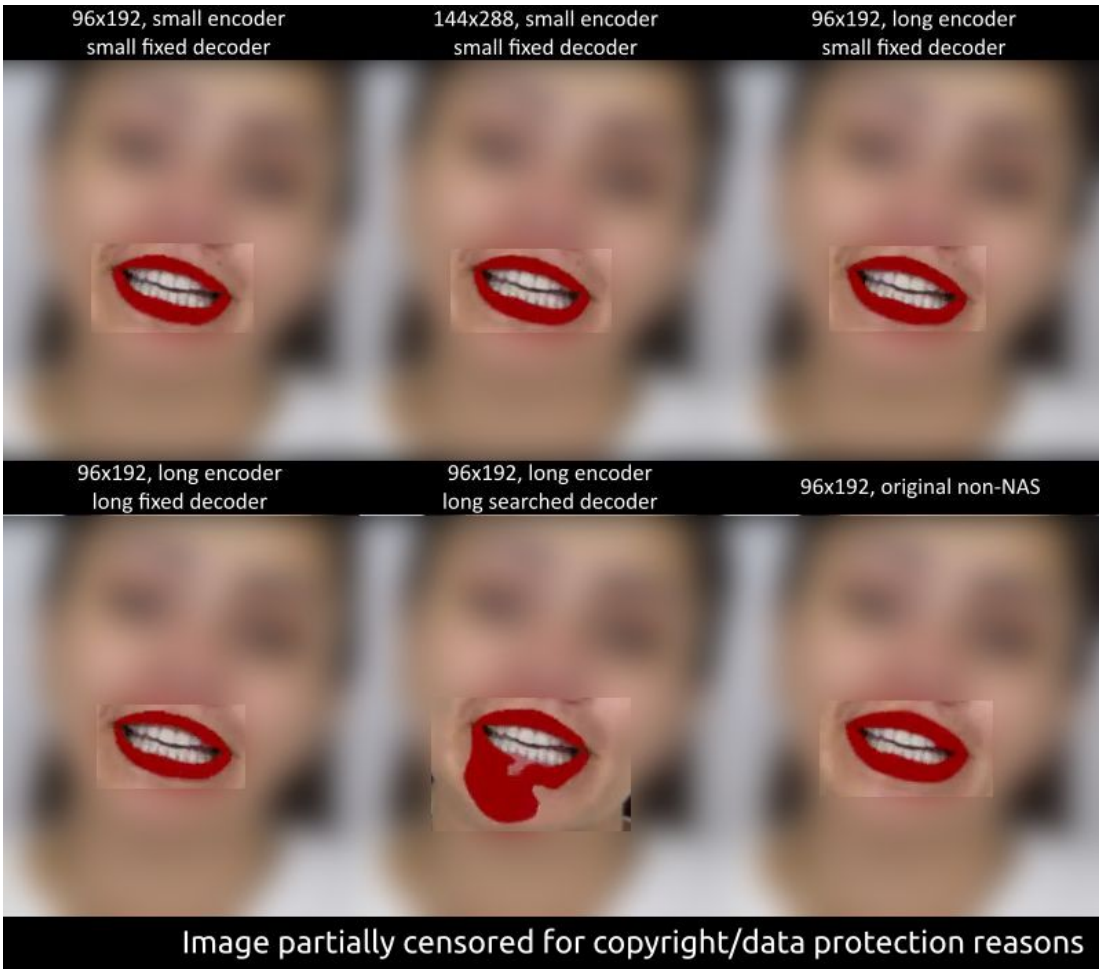

Figure 48: Results on one frame of the *easy* video, where the fully searched model is shown to be the only one to fail, due to the scaling in the preprocessing. Similar failures were also observed in other frames and in the other videos, with the output sometimes being completely blank despite good outputs from every other model.

Figure 49: Results in a case of heavy occlusion in a difficult video. The original non-NAS model has the best output, with the others failing completely.

Figure 50: Results for a normal dark skin image. Two FBNet-based models incorrectly mark the tongue as lips, while the other three miss part of the upper lip. The original non-NAS model has the best output, although it seems to extend beyond the lip boundary in places.

6.2.5 Known and potential causes of issues of the method

Due to a mistake in the removal of the latency loss when training only for accuracy, only the β value in the latency loss calculation (Equation (6.4)) was set to zero, leading to a fixed, block latency-independent, but nonzero loss contribution. While minimizing the overall loss therefore still implies the minimization of the overall loss, the constant contribution likely leads to training instability as the accuracy loss shrinks, in turn leading to suboptimal search results.

Furthermore, the relationship between the sum of latencies of the blocks and the total on-device latency of the model was not verified, mainly due to the early change in focus to only accuracy-aware training. While the original FBNet work verified this relationship on mobile devices, it only tested inference on the CPU with INT8 quantization, using the Caffe2 framework. The correlation is expected to be maintained for GPU inference, but this was not verified for TensorFlow Lite with any GPU delegate, nor for TensorFlowJS. This could be accomplished by simply generating a large enough number of random sampled architectures based on all the supernet variants, and observing the dependency between the sum of block latencies and model latencies for each network. It needs to be noted that without a reasonably good correlation, the latency awareness scheme of FBNet cannot work.

Finally, the cell blocks of the original FBNet were left as is in our FBNet-Seg variants, with the exception of adding the fixed decoder in the variants that did not search it. Since these were selected specifically for classification tasks on smartphones, they may not be the most optimal for semantic segmentation. This is potentially exacerbated by the original target being mobile CPUs, as it may have inspired selection of more CPU-friendly blocks, as opposed to GPU-friendly ones.

6.3 Notes on state-of-the-art

With respect to already existing latency-aware NAS approaches for segmentation, the state of the art as of August 2021 appears to be FasterSeg (Chen et al. 2019). As previously mentioned in Subsection 2.8.3, FasterSeg is a neural architecture search method created specifically for segmentation tasks, using multiple-resolution branches with candidate cells, but also candidate connections. Each possible path from the input to the output of the network represents the equivalent of an FBNet-like supernet architecture, since each cell is also searchable, having several convolution types or a no-op as candidate blocks. This is similar to the approach in Auto-Deeplab, as mentioned in the literature review (Section 2.8).

FasterSeg appears to be a more favorable base of research than FBNet for facial

feature segmentation, given its express focus on segmentation and claim of improved balancing of latency and accuracy in its search. However, we did not discover it until after work on the FBNet adaptation was already underway. Furthermore, given that the available implementation of FasterSeg (Chen et al. 2020) is written in PyTorch, there are the same issues regarding model conversion to TensorFlow Lite as were pointed out in Chapter 3 for the single-shot-pose method. There is also the additional issue of constructing the latency tables by converting and running all the candidate blocks in each cell in TensorFlowJS or TensorFlow Lite, which must be carried out before any other work on FasterSeg, unless restricting the search to only accuracy loss.

To check if the state-of-the-art design is capable of improving on the non-NAS models used, there was an initial desire to adapt the approach in FasterSeg to our dataset and latency tables. However, due to the PyTorch-related issues listed, as well as the code for the approach involving a complicated multi-step process, this could not be done in the time available. It is considered important future work, as proof of the ability of the method to create superior models specifically for segmentation already exists.

6.4 Ethical considerations

As in the face makeup transfer (Chapter 4) and wrinkle and blemish detection projects (Chapter 5), the main ethical concern was output quality fairness with respect to skin tone. While efforts had been made by the placement company to provide additional data to combat underrepresentation in the training datasets, the experiments in this project only use the data from CelebAMask-HQ, which has a significant Caucasian bias (Karkkainen and Joo 2021). This appears to be reflected in the lower output quality on the unseen videos featuring people with darker skin. This could be combatted in future work by including the extra data collected by the placement company.

As for previous projects (Chapters 3, 4, and 5), this research was intended for use in the placement company's apps, which in turn had the purpose of suggesting makeup products for purchase, and creating a beauty recommendation engine. No ethical analysis was made in this regard by the author of this thesis, due to lack of involvement with this aspect of the company's work.

Real-time, smartphone-optimized semantic segmentation with neural architecture search has many applications beyond beauty, for various augmented reality tasks like occlusion of virtual objects and background replacement. Smartphone-oriented semantic segmentation can also be used for background removal, either in real-time (for video calls) or offline (for simple video editing for video sharing apps such as TikTok

(TikTok 2022)).

6.5 Conclusions

While some of the FBNet-Seg variants were of acceptable quality and latency for real-time lip segmentation in the Unity and TensorFlow Lite system, they were slightly worse than the non-NAS model. Future work on FBNet-Seg would include fixing of the latency removal issue, verification of the relationship between model latency and sum of block latencies, and re-comparing with the non-NAS architecture retrained with the same preprocessing scheme. Further FBNet-based supernet architectures could also be created and tested, for both segmentation and other real-time tasks, like facial keypoint estimation.

However, given the explicit focus on segmentation and the improved search space, the testing or adaptation of the FasterSeg approach for real-time smartphone GPU operation is deemed to be a better avenue for future research.

Chapter 7

Conclusions and future work

The rapid rise of machine learning for computer vision has allowed for efficient solutions to problems that were previously not solvable for real-time applications. These advances also enabled the extension to more constrained target running environments, including mobile devices and web browsers. This has lead to growing interest for CVML-powered applications from various industries, including the beauty industry, offering many opportunities for research and implementation. However, the technology still requires improvement in many areas, for better neural network accuracy and latency, new or improved algorithms, and dataset collection and processing.

In this thesis, the main focus was on the application of machine learning algorithms and frameworks for beauty apps on mobile devices. As the area is too large to tackle in its entirety, this work aimed to isolate various elements and solve smaller problems within the general CVML context.

Overall, the work presented theoretical and practical advances in machine learning for computer vision applications, with a focus on beauty applications, as was requested by the placement company. The work resulted in the contributions listed below.

The first was the design of a prototype C++ framework that could run TensorFlow Lite models in Unity on Android device CPUs, and its implementation as a C++ Unity plugin, which aided in the creation of an improved, production-grade framework capable of running models in Unity applications on Android and iOS, both on the CPU and GPU (Chapter 2.4). This became the groundwork for most of the projects of this thesis, as well as other projects done by the company.

The second was a real-time 6DoF hair curler tracking system using two machine learning models, with good accuracy aside from the roll angle (Chapter 3).

Other contributions were an extension of a method for wrinkle detection to allow

for generic blemish detection (like acne and moles), the implementation of both as a Unity C++ plugin, and the description of a potential alternate method using Gabor filters (Chapter 5).

Finally, an incremental improvement was made to a makeup style transfer approach, with fewer output artifacts and lower latency in most execution environments (Chapter 4).

While strong contributions did not result from every project, some conclusions can be drawn from the research undertaken in them. Below, we expand and discuss specific contributions and conclusions for all the projects outlined in this thesis.

7.1 Adaptation of TensorFlow Lite for Unity

In Chapter 2.4, a framework for using the TensorFlow Lite C++ API from Unity was created and tested, in order to make machine learning inference possible in smartphone Unity apps. The system that eventually resulted was built upon to try and add extra GPU functionality, to allow more of the required neural networks to use hardware acceleration, and have lower on-device latency.

The initial prototype code, mentioned in Section 2.4.4, was not, to our knowledge, used in later development or production versions of the TensorFlow Lite plugin, except potentially for reference. It was however a required step, if only as a feasibility study, as the TensorFlow Lite plugin that eventually emerged became the basis of almost all the research carried out in this thesis. It further plays a significant and continuing role in the placement company's machine learning-based apps.

As the addition of new GPU ops to TensorFlow Lite was partially successful but not maintainable (Subsection B.3), future work could be investigation of better implementation approaches. Furthermore, there are other ops that TensorFlow Lite GPU could benefit from, such as instance normalization layers, as mentioned in Subsection 4.8.4.

7.2 Handheld object pose estimation on smartphones

This project, detailed in Chapter 3, resulted in a working curler pose model and tracking system. The initial non-tracking version was implemented in an app delivered to the client company, albeit as a secondary option to a higher-latency but higher-accuracy model, developed by a different placement company employee.

Future work would focus on improving the YOLOv2-style tail of the model, following the YOLOv3 multi-resolution output approach. Also, in order to handle the

difficult roll rotation estimation, the hand joints could be tracked by treating them as extra keypoints in the pose model, which would allow for roll speed to be approximated. The required hand keypoint ground truth could be obtained by using a sufficiently powerful offline hand pose tracking network.

In addition, the tracking scheme could be improved by using a recurrent form of the network taking the previous pose result as an extra input, although this would require the input dataset to have reasonably large sequences of time-adjacent frames annotated (without significant gaps). Alternatively, this time-adjacency could be simulated by interpolating motion between available ground truth poses, to mimic real-life motion and provide synthetic pose data for this recurrent scheme.

As such tracking schemes are not dependent on the base model used, more recent, efficient or accurate models could be built upon instead, as was planned with the coarse segmentation-based approach in Section 3.9.

7.3 Face makeup transfer with generative adversarial networks

As shown in Chapter 4, the improvements to the output quality were too modest to allow for a version usable in any app, especially when using unusual makeup style targets. Furthermore, the latency-improved model was not fully finalized. Still, lessened output artifacts and overall better results were obtained by using depthwise separable convolution residual blocks, when the transpose convolutions were retained in the output branches.

Future work maintaining the general method would involve testing more architecture variants for improved accuracy and latency. Investigation of on-device failure cases (particularly on the GPU on Android) could be beneficial, but given the non-real-time single-frame use case and the overhead of preparing the GPU delegate, the model could probably be kept CPU-bound without a significant degradation of user experience.

With greater modifications, the outputs could probably be improved more significantly. A promising option is to use the UV-unwrapped face textures as to remove head pose variation, or preferably, to use a face mesh fitter and a position-aware form of histogram loss (like the UV loss in Li et al. (2020)). This would likely reduce the blurriness caused by using histogram loss without position awareness to transfer the makeup colors.

Finally, the BeautyGAN scheme could be used to instead generate a makeup overlay instead of a style-transferred image, by calculating the training-time loss using

the overlay output composited onto the non-makeup input. This would allow for real-time use as a virtual try-on, similarly to the preset or manually configured makeup overlays.

7.4 Wrinkle and blemish detection

In Chapter 5, use of the Jerman enhancement filter for wrinkles (Jerman et al. 2016; Elbashir and Yap 2020) was extended to facial blemishes, and both were implemented as a Unity C++ plugin. This gave suitable results, with the wrinkle output quality being superior to that of the the blemish outputs. An alternative method for wrinkle detection was also described, using Gabor filters and a direction map. Initial tests showed mixed results, both when it was used as a postprocessing step for the Jerman filter, as well as when used as a single process.

The more immediate future work would be better tuning of the σ and τ variables of the Jerman filter, particularly for the blemishes, to minimize noise and false positives in the outputs. Alternate normalization schemes could also be attempted, in order to verify whether they are responsible for any part of the noisy outputs.

Any more significant improvements to the Jerman enhancement filter scheme for wrinkles would probably need to use wrinkle tracking, for example as in Batool and Chellappa (2015). These tracing steps do tend to increase the latency of the method, and may therefore become unsuitable for high resolution images.

As a simple improvement in any app implementation, the Unity plugin could verify that the image is not too blurry by checking the variance of the Laplacian.

Given that a machine learning approach is required if classifying the blemishes is also desired alongside their detection, the method could be used for fine-grained segmentation of manually annotated zones of different blemish classes, as mentioned in Section 2.7 and Subsection 5.2.5.

With respect to the Gabor filter approach with direction maps, the most relevant future work would be implementation of the method as a GPU shader, to ensure low latency. With this implementation, more in-depth testing and validation of the method could be carried out.

Subsequent future work could include use of a directionality map, as well as the extension to face poses other than forward-facing, as mentioned in Subsection 5.3.5.

7.5 Hardware-aware neural architecture search for segmentation in mobile and web apps

As shown in Chapter 6, quality parity was not obtained with the non-NAS version of the lip segmentation model, although the apparent best NAS architecture had lower latency than it, and the other ones were also either faster or comparable. It was also observed that latency awareness is not relevant in the initial phases of creating a better NAS model, as the largest factor in end latency was the supernet structure, at least in the FBNet static supernet scheme.

While the resulting models were more lightweight, having better latencies for TensorFlow Lite GPU and likely TensorFlowJS GPU operation, this was not due to latency awareness, and is not deemed a theoretical success, especially given the inferior accuracy.

Future work would involve investigation of the FasterSeg method (Chen et al. 2019), or another method that uses a variable network-level architecture based on optimizing a path through a multiresolution cell grid. Other methods using this tactic are Auto-Deeplab (Liu et al. 2019a) and DCNAS (Zhang et al. 2021). These all target segmentation directly, and are therefore likely a better starting point than FBNet for a sufficiently low-latency and high-accuracy segmentation model.

As was the intention in Chapter 6, given the other types of non-NAS models in use, the FBNet or FasterSeg schemes could be extended to other outputs, like facial keypoints, or even a combined facial segmentation and keypoint model.

7.6 Discussion and future work

While the undertaken thesis projects were considered only with respect to the beauty industry, many have applications in other domains. The Unity and TensorFlow Lite system, for example, can be used for any smartphone or web browser-oriented augmented reality task, like games. Other examples include the object pose detection and neural architecture search for segmentation projects, as these are general tasks, with applications in many domains, such as medical or defense. While makeup transfer does not have applications outside beauty, low-resource generative adversarial networks can and could be used for various other tasks, such as Snapchat face filters and inpainting for object removal.

The low latency and low resource requirements of smartphone augmented reality could likely become more significant with the future emergence of augmented reality smartglasses. This is because smartphones can capture a camera frame, add virtual

elements, and then show the frame on screen. Conversely, many smartglasses, like the Hololens (Microsoft 2021b), are see-through, so any latency in generating the virtual elements is more noticeable.

Given the resource requirements of standalone augmented reality glasses, it is possible that initial future versions may use a connected smartphone as the computing device, as may be the case for the upcoming Apple Glasses (Rado 2021). As such, many aspects of smartphone-oriented approaches for augmented reality, as the ones presented in this thesis, may also apply to these forms of smartglasses.

Despite the emergence of such devices, given the ubiquity of mobile devices, smartphone-oriented augmented reality approaches and the associated CVML techniques will remain important for many years to come.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y. and Zheng, X., 2016. TensorFlow: A System for Large-Scale Machine Learning. *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, USA: USENIX Association, OSDI'16, 265–283.

Active Beans Inc., 2021. PicBeauty - Apps on Google Play [online]. Available from: `https://play.google.com/store/apps/details?id=com.ywqc.picbeauty` [Accessed 08 October 2021].

Aitken, A., Ledig, C., Theis, L., Caballero, J., Wang, Z. and Shi, W., 2017. Checkerboard artifact free sub-pixel convolution: A note on sub-pixel convolution, resize convolution and convolution resize [online]. Available from: `http://arxiv.org/abs/1707.02937`.

Alamdari, N., Tavakolian, K., Alhashim, M. and Fazel-Rezai, R., 2016. Detection and classification of acne lesions in acne patients: A mobile application. *IEEE International Conference on Electro Information Technology*, 2016-Augus, 739–743.

Alarifi, J. S., Goyal, M., Davison, A. K., Dancey, D., Khan, R. and Yap, M. H., 2017. Facial skin classification using convolutional neural networks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10317 LNCS, 479–485.

Aldoma, A., Tombari, F., Prankl, J., Richtsfeld, A., Di Stefano, L. and Vincze, M., 2013. Multimodal cue integration through Hypotheses Verification for RGB-D object recognition and 6DOF pose estimation. *Proceedings - IEEE International Conference on Robotics and Automation*, 2104–2111.

Aleotti, F., Zaccaroni, G., Bartolomei, L., Poggi, M., Tosi, F. and Mattoccia, S., 2021.

Real-time single image depth perception in the wild with handheld devices. *Sensors (Switzerland)*, 21 (1), 1–17. Available from: http://arxiv.org/abs/2006.05724.

Aleph, 2008. File:Angela Merkel (2008).jpg - Wikimedia Commons [online]. Available from: https://commons.wikimedia.org/wiki/File:Angela_Merkel_(2008).jpg [Accessed 26 June 2022].

Alfed, N., Khelifi, F., Bouridane, A. and Seker, H., 2015. Pigment network-based skin cancer detection. *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBS*, Institute of Electrical and Electronics Engineers Inc., volume 2015-Novem, 7214–7217.

Alquran, H., Qasmieh, I. A., Alqudah, A. M., Alhammouri, S., Alawneh, E., Abughazaleh, A. and Hasayen, F., 2017. The melanoma skin cancer detection and classification using support vector machine. *2017 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies, AEECT 2017*, Institute of Electrical and Electronics Engineers Inc., volume 2018-Janua, 1–5.

Alrabiah, A., Alduailij, M. and Crane, M., 2019. Computer-based approach to detect wrinkles and suggest facial fillers. *Int J Adv Comput Sci Appl*, 10 (9).

Apple, 2021. ARKit Overview - Augmented Reality - Apple Developer [online]. Available from: https://developer.apple.com/augmented-reality/arkit/ [Accessed 16 September 2021].

Aznar-Casanova, J., Torro-Alves, N. and Fukusima, S., 2010. How Much Older Do You Get When a Wrinkle Appears on Your Face? Modifying Age Estimates by Number of Wrinkles. *Aging, Neuropsychology, and Cognition*, 17 (4), 406–421. Available from: https://doi.org/10.1080/13825580903420153.

Babenko, B., Yang, M.-H. and Belongie, S., 2009. Visual tracking with online Multiple Instance Learning. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, Institute of Electrical and Electronics Engineers (IEEE), 983–990.

Bahraini, M. S., Rad, A. B. and Bozorg, M., 2019. SLAM in Dynamic Environments: A Deep Learning Approach for Moving Object Tracking Using ML-RANSAC Algorithm. *Sensors 2019, Vol. 19, Page 3699*, 19 (17), 3699. Available from: https://www.mdpi.com/1424-8220/19/17/3699.

Barowski, T., Szczot, M. and Houben, S., 2019. 6DoF Vehicle Pose Estimation Using Segmentation-Based Part Correspondences. *2019 IEEE Intelligent Transportation Systems Conference, ITSC 2019*, 573–580.

Batool, N. and Chellappa, R., 2012a. A Markov Point Process model for wrinkles in human faces. *Proceedings - International Conference on Image Processing, ICIP*, 1809–1812.

Batool, N. and Chellappa, R., 2012b. Modeling and detection of wrinkles in aging human faces using marked point processes. A. Fusiello, V. Murino and R. Cucchiara, eds., *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Berlin, Heidelberg: Springer Berlin Heidelberg, volume 7584 LNCS, 178–188.

Batool, N. and Chellappa, R., 2014. Detection and inpainting of facial wrinkles using texture orientation fields and Markov random field modeling. *IEEE Transactions on Image Processing*, 23 (9), 3773–3788.

Batool, N. and Chellappa, R., 2015. Fast detection of facial wrinkles based on Gabor features using image morphology and geometric constraints. *Pattern Recognition*, 48 (3), 642–658.

Bazarevsky, V., Grishchenko, I., Raveendran, K., Zhu, T., Zhang, F. and Grundmann, M., 2020. BlazePose: On-device Real-time Body Pose tracking. *arXiv preprint arXiv:2006.10204*. Available from: `http://arxiv.org/abs/2006.10204`.

Bazarevsky, V., Kartynnik, Y., Vakunov, A., Raveendran, K. and Grundmann, M., 2019. BlazeFace: Sub-millisecond Neural Face Detection on Mobile GPUs. *arXiv preprint arXiv:1907.05047*. Available from: `http://arxiv.org/abs/1907.05047`.

Belousov, S., 2021. MobileStyleGAN: A Lightweight Convolutional Neural Network for High-Fidelity Image Synthesis. *arXiv preprint arXiv:2104.04767*.

Bhoi, A., 2019. Monocular depth estimation: A survey. *arXiv preprint arXiv:1901.09402*.

Biron, B., 2019. Beauty Becomes a $532 Billion Industry Thanks to These Trends [online]. Available from: `https://www.businessinsider.com/beauty-multibillion-industry-trends-future-2019-7?r=US&IR=T`.

Biswas, J. and Veloso, M., 2012. Depth camera based indoor mobile robot localization and navigation. *Proceedings - IEEE International Conference on Robotics and Automation*, 1697–1702.

Bonfiglio, N., 2018. DeepMind and Unity Will Collaborate on Artificial Intel-

ligence Research [online]. Available from: https://www.dailydot.com/debug/unity-deempind-ai/.

Borenstein, J. and Koren, Y., 1988. Obstacle avoidance with ultrasonic sensors. *IEEE Journal on Robotics and Automation*, 4 (2), 213–218.

Cai, H., Zhu, L. and Han, S., 2018. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. *arXiv*. Available from: http://arxiv.org/abs/1812.00332.

Cao, C., Bradley, D., Zhou, K. and Beeler, T., 2015. Real-time high-fidelity facial performance capture. *ACM Transactions on Graphics*, 34 (4).

Chen, L. C., Papandreou, G., Schroff, F. and Adam, H., 2017a. Rethinking atrous convolution for semantic image segmentation. *arXiv*. Available from: http://arxiv.org/abs/1706.05587.

Chen, W., Gong, X., Liu, X., Zhang, Q., Li, Y. and Wang, Z., 2019. FasterSeg: Searching for Faster Real-time Semantic Segmentation. *Iclr*, 1–14. Available from: http://arxiv.org/abs/1912.10917.

Chen, W., Gong, X., Liu, X., Zhang, Q., Li, Y. and Wang, Z., 2020. VITA-Group/FasterSeg: [ICLR 2020] "FasterSeg: Searching for Faster Real-time Semantic Segmentation" by Wuyang Chen, Xinyu Gong, Xianming Liu, Qian Zhang, Yuan Li, Zhangyang Wang [online]. Available from: https://github.com/VITA-Group/FasterSeg [Accessed 16 November 2021].

Chen, X., Ma, H., Wan, J., Li, B. and Xia, T., 2017b. Multi-view 3d object detection network for autonomous driving. *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 1907–1915.

Choi, C. and Christensen, H. I., 2012a. 3D pose estimation of daily objects using an RGB-D camera. *IEEE International Conference on Intelligent Robots and Systems*, 3342–3349.

Choi, C. and Christensen, H. I., 2012b. 3D textureless object detection and tracking: An edge-based approach. *IEEE International Conference on Intelligent Robots and Systems*, 3877–3884. Available from: http://ieeexplore.ieee.org/document/6386065/.

Chollet, F., 2017. Xception: Deep learning with depthwise separable convolutions. *Pro-*

ceedings of the IEEE conference on computer vision and pattern recognition, 1251–1258.

Choukroun, Y., Kravchik, E., Yang, F. and Kisilev, P., 2019. Low-bit Quantization of Neural Networks for Efficient Inference. *ICCV Workshops*, 3009–3018.

Chu, X., Zhang, B. and Xu, R., 2020. Multi-objective Reinforced Evolution in Mobile Neural Architecture Search. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12538 LNCS, 99–113. Available from: `http://arxiv.org/abs/1901.01074`.

Çiçek, Ö., Abdulkadir, A., Lienkamp, S. S., Brox, T. and Ronneberger, O., 2016. 3D U-net: Learning dense volumetric segmentation from sparse annotation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer, volume 9901 LNCS, 424–432.

Cliff Dermatology, 2021. Acne — cliff dermatology [online]. Available from: `https://www.cliff-dermatologist.co.uk/acne` [Accessed 19 November 2021].

Collet, A., Berenson, D., Srinivasa, S. S. and Ferguson, D., 2009. Object recognition and full pose registration from a single image for robotic manipulation. *2009 IEEE International Conference on Robotics and Automation*, 48–55.

Cula, G. O., Bargo, P. R., Nkengne, A. and Kollias, N., 2013. Assessing Facial Wrinkles: Automatic Detection and Quantification. *Skin Research and Technology*, 19 (1).

DCGM and Nvidia, 2020. DCGM/ffhq-features-dataset: Gender, Age, and Emotion for Flickr-Faces-HQ Dataset (FFHQ) [online]. Available from: `https://github.com/DCGM/ffhq-features-dataset` [Accessed 10 September 2021].

Deng, X., Mousavian, A., Xiang, Y., Xia, F., Bretl, T. and Fox, D., 2021. PoseRBPF: A rao-blackwellized particle filter for 6-D object pose tracking. *IEEE Transactions on Robotics*, 37 (5), 1328–1342. Available from: `http://arxiv.org/abs/1905.09304`.

Dice, L. R., 1945. Measures of the Amount of Ecologic Association Between Species. *Ecology*, 26 (3), 297–302. Available from: `http://www.jstor.org/stable/1932409`.

Dickmann, J., Klappstein, J., Hahn, M., Appenrodt, N., Bloecher, H.-L., Werber, K. and Sailer, A., 2016. "Automotive radar the key technology for autonomous driving: From detection and ranging to environmental understanding". *2016 IEEE Radar Conference (RadarConf)*, 1–6.

Durrant-Whyte, H. and Bailey, T., 2006. Simultaneous localization and mapping: Part I. *IEEE Robotics and Automation Magazine*, 13 (2), 99–108.

Elbashir, R. M. and Yap, M. H., 2020. Evaluation of automatic facial wrinkle detection algorithms. *Journal of Imaging*, 6 (4), 17. Available from: `https://www.mdpi.com/2313-433X/6/4/17`.

Everingham, M., Van Gool, L., Williams, C. K., Winn, J. and Zisserman, A., 2010. The pascal visual object classes (VOC) challenge. *International Journal of Computer Vision*, 88 (2), 303–338. Available from: `http://link.springer.com/10.1007/s11263-009-0275-4`.

Facebook, 2018. Caffe2 and PyTorch join forces to create a Research + Production platform PyTorch 1.0 — Caffe2 [online]. Available from: `https://caffe2.ai/blog/2018/05/02/Caffe2_PyTorch_1_0.html` [Accessed 15 October 2021].

Facebook, 2020a. pytorch/caffe2/mobile/contrib/libopencl-stub at master · pytorch/pytorch · GitHub [online]. Available from: `https://github.com/pytorch/pytorch/tree/master/caffe2/mobile/contrib/libopencl-stub` [Accessed 15 October 2021].

Facebook, 2020b. pytorch/caffe2/mobile/contrib/libvulkan-stub at master · pytorch/pytorch · GitHub [online]. Available from: `https://github.com/pytorch/pytorch/tree/master/caffe2/mobile/contrib/libvulkan-stub` [Accessed 15 October 2021].

Facebook, 2021a. Home — PyTorch [online]. Available from: `https://pytorch.org/mobile/home/` [Accessed 15 October 2021].

Facebook, 2021b. Home — PyTorch [online]. Available from: `https://pytorch.org/mobile/home/#key-features` [Accessed 15 October 2021].

Facebook, 2021c. Integrating Caffe2 on iOS/Android — Caffe2 [online]. Available from: `https://caffe2.ai/docs/mobile-integration.html` [Accessed 15 October 2021].

Fan, R., Wang, L., Bocus, M. J. and Pitas, I., 2020. Computer Stereo Vision for Autonomous Driving. *arXiv preprint arXiv:2012.03194*. Available from: `https://arxiv.org/abs/2012.03194v2`.

Feng, Y., Wu, F., Shao, X., Wang, Y. and Zhou, X., 2018a. Joint 3D Face Reconstruction and Dense Alignment with Position Map Regression Network. *Proceedings of the European Conference on Computer Vision (ECCV)*, 534–551.

Feng, Y., Wu, F., Shao, X., Wang, Y. and Zhou, X., 2018b. YadiraF/PRNet: Joint 3D Face Reconstruction and Dense Alignment with Position Map Regression Network (ECCV 2018) [online]. Available from: `https://github.com/YadiraF/PRNet` [Accessed 05 November 2021].

Flynn, P. J. and Jain, A. K., 1991. CAD-based computer vision: from CAD models to relational graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13 (2), 114–132.

Fogel, I. and Sagi, D., 1989. Gabor filters as texture discriminator. *Biological Cybernetics 1989 61:2*, 61 (2), 103–113. Available from: `https://link.springer.com/article/10.1007/BF00204594`.

Frangi, A. F., 2001. *Three-dimensional model-based analysis of vascular and cardiac images*. Ph.D. thesis, Utrecht University.

Fu, M. and Zhou, W., 2019. DeepHMap++: Combined projection grouping and correspondence learning for full DoF pose estimation. *Sensors (Switzerland)*, 19 (5).

Garnier, 2021. Virtual Hair Colour Try On — Try On Hair Colour — Garnier [online]. Available from: `https://www.garnier.co.uk/virtual-try-on` [Accessed 20 April 2021].

Garon, M., Boulet, P. O., Doironz, J. P., Beaulieu, L. and Lalonde, J. F., 2017. Real-Time High Resolution 3D Data on the HoloLens. *Adjunct Proceedings of the 2016 IEEE International Symposium on Mixed and Augmented Reality, ISMAR-Adjunct 2016*, Institute of Electrical and Electronics Engineers Inc., 189–191.

Gary, B., 2008. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 1 (2236121).

Gatys, L. A., Ecker, A. S. and Bethge, M., 2015. A Neural Algorithm of Artistic Style. *Journal of Vision*, 16 (12), 326. Available from: `https://arxiv.org/abs/1508.06576v2`.

Geiger, A., Lenz, P. and Urtasun, R., 2012. Are we ready for autonomous driving? the KITTI vision benchmark suite. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, IEEE, 3354–3361. Available from: `http://ieeexplore.ieee.org/document/6248074/`.

Godard, C., Mac Aodha, O. and Brostow, G. J., 2017. Unsupervised monocular depth

estimation with left-right consistency. *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017-Janua, 6602–6611.

Godard, C., Mac Aodha, O., Firman, M. and Brostow, G. J., 2019. Digging into self-supervised monocular depth estimation. *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 3828–3838. Available from: `http://arxiv.org/abs/1806.01260`.

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y., 2014. Generative Adversarial Networks. *Advances in neural information processing systems*, 2672–2680. Available from: `http://arxiv.org/abs/1406.2661`.

Google, 2018. Real-time Human Pose Estimation in the Browser with TensorFlow.js — The TensorFlow Blog [online]. Available from: `https://blog.tensorflow.org/2018/05/real-time-human-pose-estimation-in.html` [Accessed 24 September 2021].

Google, 2019. tensorflow/tensorflow/examples/ios at r2.3 · tensorflow/tensorflow · GitHub [online]. Available from: `https://github.com/tensorflow/tensorflow/tree/r2.3/tensorflow/examples/ios` [Accessed 15 October 2021].

Google, 2020a. Google Developers Blog: A new wave of AR Realism with the ARCore Depth API [online]. Available from: `https://developers.googleblog.com/2020/06/a-new-wave-of-ar-realism-with-arcore-depth-api.html` [Accessed 19 August 2020].

Google, 2020b. google/mediapipe: MediaPipe is a cross-platform framework for building multimodal applied machine learning pipelines [online]. Available from: `https://github.com/google/mediapipe` [Accessed 05 March 2020].

Google, 2020c. tensorflow/tensorflow/examples/android at r2.3 · tensorflow/tensorflow · GitHub [online]. Available from: `https://github.com/tensorflow/tensorflow/tree/r2.3/tensorflow/examples/android` [Accessed 15 October 2021].

Google, 2021a. Detect faces — Cloud Vision API — Google Cloud [online]. Available from: `https://cloud.google.com/vision/docs/detecting-faces` [Accessed 12 November 2021].

Google, 2021b. Firebase Test Lab — Firebase Documentation [online]. Available from: `https://firebase.google.com/docs/test-lab` [Accessed 15 October 2021].

Google, 2021c. FlatBuffers: FlatBuffers [online]. Available from: https://google.github.io/flatbuffers/ [Accessed 06 October 2021].

Google, 2021d. GitHub - google/XNNPACK: High-efficiency floating-point neural network inference operators for mobile, server, and Web [online]. Available from: https://github.com/google/XNNPACK [Accessed 13 October 2021].

Google, 2021e. Hosted models — TensorFlow Lite [online]. Available from: https://www.tensorflow.org/lite/guide/hosted_models#quantized_models [Accessed 05 October 2021].

Google, 2021f. Hosted models — TensorFlow Lite [online]. Available from: https://www.tensorflow.org/lite/guide/hosted_models#floating_point_models [Accessed 15 October 2021].

Google, 2021g. Hosted models — TensorFlow Lite [online]. Available from: https://www.tensorflow.org/lite/guide/hosted_models [Accessed 25 August 2021].

Google, 2021h. TensorFlow Lite [online]. Available from: https://www.tensorflow.org/lite/guide#key_features [Accessed 06 October 2021].

Google, 2021i. TensorFlow Lite [online]. Available from: https://www.tensorflow.org/lite/guide#1_generate_a_tensorflow_lite_model [Accessed 07 October 2021].

Google, 2021j. TensorFlow Lite 8-bit quantization specification [online]. Available from: https://www.tensorflow.org/lite/performance/quantization_spec [Accessed 02 April 2021].

Google, 2021k. TensorFlow Lite Roadmap [online]. Available from: https://www.tensorflow.org/lite/guide/roadmap [Accessed 16 October 2021].

Google, 2021l. tensorflow/tensorflow/lite/tools/benchmark at master · tensorflow/tensorflow · GitHub [online]. Available from: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/tools/benchmark [Accessed 15 March 2021].

Google, 2021m. tfjs/e2e/benchmarks/local-benchmark at master · tensorflow/tfjs [online]. Available from: https://github.com/tensorflow/tfjs/tree/master/e2e/benchmarks/local-benchmark [Accessed 15 November 2021].

Gualtieri, M., Pas, A. T., Saenko, K. and Platt, R., 2016. High precision grasp pose

detection in dense clutter. *IEEE International Conference on Intelligent Robots and Systems*, 2016-Novem, 598–605.

Guha, S., 2015. *Computer Graphics Through OpenGL: From Theory to Experiments*. 2nd edition. Boca Raton: CRC Press. Available from: http://books.google.com/books?hl=en&lr=&id=7bCiFepXle0C&oi=fnd&pg=PP1&dq=Computer+Graphics+Through+OpenGL:+From+Theory+to+Experiments&ots=PK49-WKcsv&sig=Lpxw8-ZnoDXiHksD8uR86-ZTBwo.

Guo, D. and Sim, T., 2009. Digital face makeup by example. *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2009*, IEEE, volume 2009 IEEE, 73–79. Available from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5206833.

Haines, A., 2021. From 'Instagram Face' To 'Snapchat Dysmorphia': How Beauty Filters Are Changing The Way We See Ourselves. *Forbes*. Available from: https://www.forbes.com/sites/annahaines/2021/04/27/from-instagram-face-to-snapchat-dysmorphia-how-beauty-filters-are-changing-the-way-we-see-ourselves/.

Harbison, C., 2019. Snapchat's New Gender Swap Filter Will Make You Question Your Identity: How to Get the Male to Female Filter. *Newsweek*. Available from: https://www.newsweek.com/snapchat-gender-swap-filter-how-get-girl-boy-change-male-female-how-use-not-1425014.

Hare, S., Saffari, A. and Torr, P. H., 2011. Struck: Structured output tracking with kernels. *Proceedings of the IEEE International Conference on Computer Vision*, 263–270.

Harisankar, V., Sajith, V. V. and Soman, K. P., 2020. Unsupervised Depth Estimation from Monocular Images for Autonomous Vehicles. *Proceedings of the 4th International Conference on Computing Methodologies and Communication, ICCMC 2020*, 904–909.

He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778.

He, Y., Sun, W., Huang, H., Liu, J., Fan, H. and Sun, J., 2020. Pvn3d: A deep point-wise 3d keypoints voting network for 6dof pose estimation. *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 11632–11641.

He, Z., Zuo, W., Kan, M., Shan, S., Chen, X., Shan, S. and Chen, X., 2019. AttGAN: Facial Attribute Editing by Only Changing What You Want. *IEEE Transactions on Image Processing*, 28 (11), 1–1. Available from: https://ieeexplore.ieee.org/document/8718508/.

Held, D., Thrun, S. and Savarese, S., 2016. Learning to track at 100 FPS with deep regression networks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer Verlag, volume 9905 LNCS, 749–765. Available from: http://arxiv.org/abs/1604.01802.

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B. and Hochreiter, S., 2017. GANs trained by a two time-scale update rule converge to a local Nash equilibrium. I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan and R. Garnett, eds., *Advances in Neural Information Processing Systems*, Curran Associates, Inc., volume 2017-Decem, 6627–6638. Available from: https://proceedings.neurips.cc/paper/2017/file/8a1d694707eb0fefe65871369074926d-Paper.pdf.

Hodaň, T., Haluza, P., Obdrzalek, Š., Matas, J., Lourakis, M. and Zabulis, X., 2017. T-LESS: An RGB-D dataset for 6D pose estimation of texture-less objects. *Proceedings - 2017 IEEE Winter Conference on Applications of Computer Vision, WACV 2017*, 880–888.

Holynski, A. and Kopf, J., 2018. Fast Depth Densification for Occlusion-Aware Augmented Reality. *ACM Trans. Graph.*, 37 (6). Available from: https://doi.org/10.1145/3272127.3275083.

Hopfield, J. J., 1982. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79 (8), 2554–2558. Available from: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC346238/.

Hoshyar, A. N., Al-Jumaily, A. and Hoshyar, A. N., 2014. The beneficial techniques in preprocessing step of skin cancer detection system comparing. *Procedia Computer Science*, Elsevier, volume 42, 25–31.

Howard, A., Sandler, M., Chen, B., Wang, W., Chen, L. C., Tan, M., Chu, G., Vasudevan, V., Zhu, Y., Pang, R., Le, Q. and Adam, H., 2019. Searching for mobileNetV3. *Proceedings of the IEEE International Conference on Computer Vision*, 2019-Octob, 1314–1324. Available from: http://arxiv.org/abs/1905.02244.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M. and Adam, H., 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv*. Available from: http://arxiv.org/abs/1704.04861.

Hu, Y., Hugonot, J., Fua, P. and Salzmann, M., 2019. Segmentation-driven 6D object pose estimation. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2019-June, 3380–3389. Available from: http://arxiv.org/abs/1812.02541.

Huang, H., Chai, J., Tong, X. and Wu, H.-T., 2011. Leveraging Motion Capture and 3D Scanning for High-Fidelity Facial Performance Acquisition. *ACM SIGGRAPH 2011 Papers*, New York, NY, USA: Association for Computing Machinery, SIGGRAPH '11, 1–10. Available from: https://doi.org/10.1145/1964921.1964969.

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J. and Keutzer, K., 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv preprint arXiv:1602.07360*. Available from: http://arxiv.org/abs/1602.07360.

IJoysoft, 2021. Beauty Camera - Selfie Camera – Apps on Google Play [online]. Available from: https://play.google.com/store/apps/details?id=photo.beauty.sticker.ar.camera&hl=en_GB&gl=US [Accessed 08 October 2021].

Ikeuchi, K., 1987. Generating an interpretation tree from a CAD model for 3D-object recognition in bin-picking tasks. *International Journal of Computer Vision*, 1 (2), 145–165. Available from: https://doi.org/10.1007/BF00123163.

Ioffe, S. and Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *32nd International Conference on Machine Learning, ICML 2015*, PMLR, volume 1, 448–456. Available from: http://proceedings.mlr.press/v37/ioffe15.html.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H. and Kalenichenko, D., 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2704–2713.

Jain, S., Jagtap, V. and Pise, N., 2015. Computer aided melanoma skin cancer detection using image processing. *Procedia Computer Science*, Elsevier, volume 48, 735–740.

Jamiruddin, R., Sari, A. O., Shabbir, J. and Anwer, T., 2018. RGB-Depth SLAM

Review. *arXiv preprint arXiv:1805.07696*, 14 (8), 1–18. Available from: http://arxiv.org/abs/1805.07696.

Jang, E., 2019. Eric Jang: Fun with Snapchat's Gender Swapping Filter [online]. Available from: https://blog.evjang.com/2019/05/fun-with-snapchats-gender-swapping.html [Accessed 07 September 2021].

Jerman, T., Pernuš, F., Likar, B., Špiclin, Ž., Pernus, F., Likar, B. and Spiclin, Z., 2016. Enhancement of Vascular Structures in 3D and 2D Angiographic Images. *IEEE Transactions on Medical Imaging*, 35 (9), 2107–2118.

Jerman, T., Pernuš, F., Likar, B., Špiclin, Ž., Pernus, F., Likar, B. and Spiclin, Z., 2020. timjerman/JermanEnhancementFilter: Jerman's tubular (vessel) and spherical (blob) enhancement filters [online]. Available from: https://github.com/timjerman/JermanEnhancementFilter [Accessed 26 November 2020].

Jh88, 2019. jh88/fbnet: A Keras (TensorFlow 2.0) implementation of FBNet [online]. Available from: https://github.com/jh88/fbnet [Accessed 15 November 2021].

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S. and Darrell, T., 2014. Caffe: Convolutional architecture for fast feature embedding. *Proceedings of the 22nd ACM international conference on Multimedia*, 675–678.

Jiang, W., 2021. GitHub - wtjiang98/BeautyGAN_pytorch: Official PyTorch implementation of BeautyGAN (ACM MM 2018) [online]. Available from: https://github.com/wtjiang98/BeautyGAN_pytorch [Accessed 15 March 2021].

Johannes Kepler University Linz Institute of Bioinformatics, 2017. bioinf-jku/TTUR: Two time-scale update rule for training GANs [online]. Available from: https://github.com/bioinf-jku/TTUR [Accessed 04 November 2021].

Johnson, J., Alahi, A. and Fei-Fei, L., 2016. Perceptual losses for real-time style transfer and super-resolution. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer Verlag, volume 9906 LNCS, 694–711. Available from: http://arxiv.org/abs/1603.08155.

Kalal, Z., Mikolajczyk, K. and Matas, J., 2011. Tracking-learning-detection. *IEEE transactions on pattern analysis and machine intelligence*, 34 (7), 1409–1422.

Karkkainen, K. and Joo, J., 2021. FairFace: Face Attribute Dataset for Balanced

Race, Gender, and Age for Bias Measurement and Mitigation. *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 1548–1558.

Karras, T., Laine, S. and Aila, T., 2019. A style-based generator architecture for generative adversarial networks. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2019-June, 4396–4405. Available from: `http://arxiv.org/abs/1812.04948`.

Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J. and Aila, T., 2020. Analyzing and improving the image quality of stylegan. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 8107–8116. Available from: `http://arxiv.org/abs/1912.04958`.

Kim, J., 2021. taki0112/SPADE-Tensorflow: Simple Tensorflow implementation of "Semantic Image Synthesis with Spatially-Adaptive Normalization" a.k.a. GauGAN, SPADE (CVPR 2019 Oral) [online]. Available from: `https://github.com/taki0112/SPADE-Tensorflow` [Accessed 26 July 2021].

Kim, J.-H., Starr, J. W. and Lattimer, B. Y., 2015. Firefighting Robot Stereo Infrared Vision and Radar Sensor Fusion for Imaging through Smoke. *Fire Technology*, 51 (4), 823–845. Available from: `https://doi.org/10.1007/s10694-014-0413-6`.

Klein, G. and Murray, D., 2009. Parallel Tracking and Mapping on a camera phone. *2009 8th IEEE International Symposium on Mixed and Augmented Reality*, 83–86.

Kuznietsov, Y., Stückler, J. and Leibe, B., 2017. Semi-supervised deep learning for monocular depth map prediction. *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017-Janua, 2215–2223.

Laga, H., Jospin, L. V., Boussaid, F. and Bennamoun, M., 2020. A survey on deep learning techniques for stereo-based depth estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Lai, K., Bo, L. and Fox, D., 2014. Unsupervised feature learning for 3D scene labeling. *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 3050–3057.

Lai, K., Bo, L., Ren, X. and Fox, D., 2011. A large-scale hierarchical multi-view RGB-D object dataset. *2011 IEEE International Conference on Robotics and Automation*, 1817–1824.

Lee, C. H., Liu, Z., Wu, L. and Luo, P., 2020. MaskGAN: Towards Diverse and Inter-

active Facial Image Manipulation. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 5548–5557.

Lee, J., Chirkov, N., Ignasheva, E., Pisarchyk, Y., Shieh, M., Riccardi, F., Sarokin, R., Kulik, A. and Grundmann, M., 2019. On-Device Neural Net Inference with Mobile GPUs. *arXiv preprint arXiv:1907.01989*. Available from: `http://arxiv.org/abs/1907.01989`.

Lessel, D., 2021. dominiklessel/opencv-gabor-filter [online]. Available from: `https://github.com/dominiklessel/opencv-gabor-filter` [Accessed 21 June 2021].

Li, J., Xu, W., Cheng, Z., Xu, K. and Klein, R., 2015. Lightweight wrinkle synthesis for 3D facial modeling and animation. *Computer-Aided Design*, 58, 117–122. Available from: `https://www.sciencedirect.com/science/article/pii/S0010448514001857`.

Li, P., Qin, T. and Others, 2018a. Stereo vision-based semantic 3d object and ego-motion tracking for autonomous driving. *Proceedings of the European Conference on Computer Vision (ECCV)*, 646–661.

Li, T., Liu, S., Qian, R., Yan, Q., Lin, L., Dong, C. and Zhu, W., 2018b. Beautygan: Instance-level facial makeup transfer with deep generative adversarial network. *MM 2018 - Proceedings of the 2018 ACM Multimedia Conference*, 645–653. Available from: `http://dl.acm.org/citation.cfm?doid=3240508.3240618`.

Li, Y., Huang, H., Cao, J., He, R. and Tan, T., 2020. Disentangled Representation Learning of Makeup Portraits in the Wild. *International Journal of Computer Vision*, 128 (8-9), 2166–2184.

Lin Hong, Yifei Wan and Jain, A., 1998. Fingerprint image enhancement: algorithm and performance evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20 (8), 777–789.

Liu, C., Chen, L. C., Schroff, F., Adam, H., Hua, W., Yuille, A. L. and Fei-Fei, L., 2019a. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2019-June, 82–92.

Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L. J., Fei-Fei, L., Yuille, A., Huang, J. and Murphy, K., 2018. Progressive Neural Architecture Search. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11205 LNCS, 19–35.

Liu, H., Simonyan, K. and Yang, Y., 2019b. DARTS: Differentiable architecture search. *7th International Conference on Learning Representations, ICLR 2019*. Available from: https://github.com/quark0/darts.

Liu, M., Ding, Y., Xia, M., Liu, X., Ding, E., Zuo, W. and Wen, S., 2019c. STGAN: A unified selective transfer network for arbitrary image attribute editing. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2019-June, 3668–3677. Available from: http://arxiv.org/abs/1904.09709.

Liu, Q., Liu, B., Wu, Y., Li, W. and Yu, N., 2019d. Real-Time Online Multi-Object Tracking in Compressed Domain. *IEEE Access*, 7, 76489–76499.

Liu, S., Ou, X., Qian, R., Wei, W. and Cao, X., 2016a. Makeup like a superstar: Deep localized makeup transfer network. S. Kambhampati, ed., *IJCAI International Joint Conference on Artificial Intelligence*, IJCAI/AAAI Press, volume 2016-Janua, 2568–2575. Available from: http://www.ijcai.org/Abstract/16/365.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. Y. and Berg, A. C., 2016b. SSD: Single shot multibox detector. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9905 LNCS, 21–37. Available from: http://arxiv.org/abs/1512.02325.

Liu, Y., Jiang, J., Sun, J., Bai, L. and Wang, Q., 2020. A Survey of Depth Estimation Based on Computer Vision. *2020 IEEE Fifth International Conference on Data Science in Cyberspace (DSC)*, IEEE, 135–141.

L'Oréal, 2021. Virtual Try On — L'Oréal Paris [online]. Available from: https://www.loreal-paris.co.uk/virtual-try-on [Accessed 20 April 2021].

Lu, J., Manton, J. H., Kazmierczak, E. and Sinclair, R., 2010. Erythema detection in digital skin images. *Proceedings - International Conference on Image Processing, ICIP*, 2545–2548.

Lu, Z., Whalen, I., Boddeti, V., Dhebar, Y., Deb, K., Goodman, E. and Banzhaf, W., 2019. Nsga-net: neural architecture search using multi-objective genetic algorithm. *Proceedings of the Genetic and Evolutionary Computation Conference*, 419–427.

Lynch, G. and Peckham, J., 2022. Apple Glasses: here's everything we know so far — TechRadar. *TechRadar*. Available from: https://www.techradar.com/news/apple-glasses#section-apple-ar-glasses-features-and-design.

Ma, N., Zhang, X., Zheng, H. T. and Sun, J., 2018. Shufflenet V2: Practical guidelines for efficient cnn architecture design. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11218 LNCS, 122–138.

Magic Leap, 2021. Magic Leap 1 — Magic Leap [online]. Available from: `https://www.magicleap.com/en-us/magic-leap-1` [Accessed 13 March 2022].

Marder-Eppstein, E., 2016. Project Tango. *ACM SIGGRAPH 2016 Real-Time Live!*, New York, NY, USA: Association for Computing Machinery, SIGGRAPH '16, 25–25. Available from: `https://doi.org/10.1145/2933540.2933550`.

Microsoft, 2021a. Facial Recognition — Microsoft Azure [online]. Available from: `https://azure.microsoft.com/en-gb/services/cognitive-services/face/#demo` [Accessed 12 November 2021].

Microsoft, 2021b. HoloLens 2—Overview, Features, and Specs — Microsoft HoloLens [online]. Available from: `https://www.microsoft.com/en-us/hololens/hardware` [Accessed 24 November 2021].

Milletari, F., Navab, N. and Ahmadi, S.-A., 2016. V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation. *2016 Fourth International Conference on 3D Vision (3DV)*, 565–571.

Mitchell, T., 1997. *Machine learning*. New York, New York, USA: McGraw Hill.

Mittal, R., Pathak, V., Gandhi, G. C., Mithal, A. and Lakhwani, K., 2021. Application of machine learning in SLAM algorithms. *Machine Learning for Sustainable Development*, 9, 147.

Muhimmah, I., Muchlis, N. F. and Kurniawardhani, A., 2021. Automatic Facial Redness Detection on Face Skin Image. *IIUM Engineering Journal*, 22 (1), 68–77. Available from: `https://journals.iium.edu.my/ejournal/index.php/iiumej/article/view/1495`.

Munguía, R. and Grau, A., 2012. Monocular SLAM for visual odometry: A full approach to the delayed inverse-depth feature initialization method. *Mathematical Problems in Engineering*, 2012.

Munkelt, O., 1994. Feature based aspects-trees. Generation and interpretation. *Proceedings of 1994 IEEE 2nd CAD-Based Vision Workshop*, 192–201.

Neven, D., De Brabandere, B., Georgoulis, S., Proesmans, M. and Van Gool,

L., 2017. Fast Scene Understanding for Autonomous Driving. *arXiv preprint arXiv:1708.02550*. Available from: http://arxiv.org/abs/1708.02550.

Newell, A., Yang, K. and Deng, J., 2016. Stacked Hourglass Networks for Human Pose Estimation. B. Leibe, J. Matas, N. Sebe and M. Welling, eds., *Computer Vision – ECCV 2016*, Cham: Springer International Publishing, 483–499.

Ng, C. C., Yap, M. H., Costen, N. and Li, B., 2015a. Automatic wrinkle detection using hybrid Hessian filter. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9005, 609–622.

Ng, C. C., Yap, M. H., Costen, N. and Li, B., 2015b. Wrinkle detection using hessian line tracking. *IEEE Access*, 3 (Xx), 1079–1088.

Ng, C. C., Yap, M. H., Costen, N. and Li, B., 2016. Will Wrinkle Estimate the Face Age? *Proceedings - 2015 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2015*, 2418–2423.

Ning, G., Zhang, Z., Huang, C., Ren, X., Wang, H., Cai, C. and He, Z., 2017. Spatially supervised recurrent convolutional neural networks for visual object tracking. *Proceedings - IEEE International Symposium on Circuits and Systems*, Institute of Electrical and Electronics Engineers Inc., 1, 9–12.

Noh, H., Hong, S. and Han, B., 2015. Learning deconvolution network for semantic segmentation. *Proceedings of the IEEE international conference on computer vision*, 1520–1528.

Nowacki, P. and Woda, M., 2020. Capabilities of ARCore and ARKit Platforms for AR/VR Applications. *Advances in Intelligent Systems and Computing*, Springer, volume 987, 358–370.

Nreal, 2021. Nreal Light [online]. Available from: https://www.nreal.ai/light/ [Accessed 13 March 2022].

Nvidia, 2021. NVlabs/ffhq-dataset: Flickr-Faces-HQ Dataset (FFHQ) [online]. Available from: https://github.com/NVlabs/ffhq-dataset [Accessed 10 April 2021].

Odena, A., Dumoulin, V. and Olah, C., 2016. Deconvolution and Checkerboard Artifacts. *Distill*. Available from: http://distill.pub/2016/deconv-checkerboard.

Oh, S., Kim, H. J. S., Lee, J. and Kim, J., 2020. RRNet: Repetition-Reduction Network

for Energy Efficient Depth Estimation. *IEEE Access*, 8, 106097–106108. Available from: http://arxiv.org/abs/1907.09707.

Pandey, R., Pidlypenskyi, P., Yang, S. and Kaeser-Chen, C., 2018. Egocentric 6-DoF Tracking of Small Handheld Objects. *arXiv preprint arXiv:1804.05870*. Available from: http://arxiv.org/abs/1804.05870.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L. and Others, 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 8026–8037.

Pauwels, K. and Kragic, D., 2015. SimTrack: A simulation-based framework for scalable real-time object pose detection and tracking. *IEEE International Conference on Intelligent Robots and Systems*, 2015-Decem, 1300–1307.

Peluso, V., Cipolletta, A., Calimera, A., Poggi, M., Tosi, F. and Mattoccia, S., 2019. Enabling Energy-Efficient Unsupervised Monocular Depth Estimation on ARMv7-Based Platforms. *Proceedings of the 2019 Design, Automation and Test in Europe Conference and Exhibition, DATE 2019*, 1703–1708. Available from: https://ieeexplore.ieee.org/document/8714893/.

Peng, S., Liu, Y., Huang, Q., Bao, H. and Zhou, X., 2018. PVNet: Pixel-wise Voting Network for 6DoF Pose Estimation. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 4561–4570. Available from: http://arxiv.org/abs/1812.11788.

Pérez, P., Gangnet, M., Blake, A., Pérez, P., Gangnet, M. and Blake, A., 2003. Poisson image editing. *ACM SIGGRAPH 2003 Papers, SIGGRAPH '03*, 5 (3), 313–318. Available from: http://portal.acm.org/citation.cfm?doid=1201775.882269.

Perfect365, 2021. Perfect365 is the #1 Augmented Reality Beauty Platform [online]. Available from: https://www.perfect365.com/# [Accessed 14 July 2021].

PerfectCorp, 2021. YouCam Makeup — Best Selfie Editor, Makeover & Retouch App [online]. Available from: https://www.perfectcorp.com/consumer/apps/ymk [Accessed 31 August 2021].

Plantinga, W. H. and Dyer, C., 1986. An algorithm for constructing the aspect graph. *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, 123–131.

Poggi, M., Aleotti, F., Tosi, F. and Mattoccia, S., 2018. Towards Real-Time Unsu-

pervised Monocular Depth Estimation on CPU. *IEEE International Conference on Intelligent Robots and Systems*, 5848–5854.

Rad, M. and Lepetit, V., 2017. BB8: A Scalable, Accurate, Robust to Partial Occlusion Method for Predicting the 3D Poses of Challenging Objects without Using Depth. *Proceedings of the IEEE International Conference on Computer Vision*, IEEE, volume 2017-Octob, 3848–3856. Available from: `http://ieeexplore.ieee.org/document/8237675/`.

Radford, A., Metz, L. and Chintala, S., 2016. Unsupervised representation learning with deep convolutional generative adversarial networks. *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*. Available from: `http://arxiv.org/abs/1511.06434`.

Rado, M., 2021. AR is the future of smartphones, starting with Apple's AR glasses - PhoneArena. *PhoneArena.com*. Available from: `https://www.phonearena.com/news/apple-augmented-reality-glasses-the-future-of-smartphones_id135935`.

Ramakrishnan, A. G., Kumar Raja, S. and Raghu Ram, H. V., 2002. Neural network-based segmentation of textures using Gabor features. *Neural Networks for Signal Processing - Proceedings of the IEEE Workshop*, 2002-Janua, 365–374.

Real, E., Aggarwal, A., Huang, Y. and Le, Q. V., 2019. Regularized evolution for image classifier architecture search. *33rd AAAI Conference on Artificial Intelligence, AAAI 2019, 31st Innovative Applications of Artificial Intelligence Conference, IAAI 2019 and the 9th AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019*, volume 33, 4780–4789.

Redmon, J., Divvala, S., Girshick, R. and Farhadi, A., 2016. You only look once: Unified, real-time object detection. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2016-Decem, 779–788. Available from: `http://arxiv.org/abs/1506.02640`.

Redmon, J. and Farhadi, A., 2017. YOLO9000: Better, faster, stronger. *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, volume 2017-Janua, 6517–6525. Available from: `http://arxiv.org/abs/1612.08242`.

Redmon, J. and Farhadi, A., 2018. YOLOv3: An Incremental Improvement. *arXiv preprint arXiv:1804.02767*. Available from: `http://arxiv.org/abs/1804.02767`.

Roeder, L., 2021. lutzroeder/netron: Visualizer for neural network, deep learning, and machine learning models [online]. Available from: https://github.com/lutzroeder/netron [Accessed 16 July 2021].

Ronneberger, O., Fischer, P. and Brox, T., 2015. U-net: Convolutional networks for biomedical image segmentation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer, Cham, volume 9351, 234–241. Available from: https://link.springer.com/chapter/10.1007/978-3-319-24574-4_28.

Rothe, R., Timofte, R. and Van Gool, L., 2015. DEX: Deep EXpectation of Apparent Age from a Single Image. *Proceedings of the IEEE International Conference on Computer Vision*, 2015-Febru, 252–257.

Rothe, R., Timofte, R. and Van Gool, L., 2018. Deep Expectation of Real and Apparent Age from a Single Image Without Facial Landmarks. *International Journal of Computer Vision*, 126 (2-4), 144–157. Available from: https://link.springer.com/article/10.1007/s11263-016-0940-3.

Roy, K., Chaudhuri, S. S., Ghosh, S., Dutta, S. K., Chakraborty, P. and Sarkar, R., 2019. Skin disease detection based on different segmentation techniques. *2019 International Conference on Opto-Electronics and Applied Optics, Optronix 2019*, Institute of Electrical and Electronics Engineers Inc.

Royo, S. and Ballesta-Garcia, M., 2019. An Overview of Lidar Imaging Systems for Autonomous Vehicles. *Applied Sciences*, 9 (19). Available from: https://www.mdpi.com/2076-3417/9/19/4093.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. and Chen, L. C., 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 4510–4520. Available from: http://arxiv.org/abs/1801.04381.

Scharstein, D., Hirschmüller, H., Kitajima, Y., Krathwohl, G., Nešić, N., Wang, X. and Westling, P., 2014. High-resolution stereo datasets with subpixel-accurate ground truth. *German conference on pattern recognition*, Springer, 31–42.

Shaw, A., Hunter, D., Landola, F. and Sidhu, S., 2019. SqueezeNAS: Fast neural architecture search for faster semantic segmentation. *Proceedings - 2019 International Conference on Computer Vision Workshop, ICCVW 2019*, 2014–2024.

SideFX, 2021. Houdini - 3D modeling, animation, VFX, look development, lighting and

rendering — SideFX [online]. Available from: https://www.sidefx.com/ [Accessed 22 November 2021].

Sifre, L. and Mallat, P. S., 2014. *Rigid-Motion Scattering For Image Classification*. Ph.D. thesis, École Polytechnique.

Silberman, N. and Fergus, R., 2011. Indoor scene segmentation using a structured light sensor. *Proceedings of the IEEE International Conference on Computer Vision*, 601–608.

Silberman, N., Hoiem, D., Kohli, P. and Fergus, R., 2012. Indoor Segmentation and Support Inference from RGBD Images. A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato and C. Schmid, eds., *Computer Vision – ECCV 2012*, Berlin, Heidelberg: Springer Berlin Heidelberg, 746–760.

Simon, J., 2021. Artbreeder [online]. Available from: https://www.artbreeder.com/about [Accessed 02 April 2022].

Smilkov, D., Thorat, N., Assogba, Y., Yuan, A., Kreeger, N., Yu, P., Zhang, K., Cai, S., Nielsen, E., Soergel, D., Bileschi, S., Terry, M., Nicholson, C., Gupta, S. N., Sirajuddin, S., Sculley, D., Monga, R., Corrado, G., Viégas, F. B. and Wattenberg, M., 2019. TensorFlow.js: Machine Learning for the Web and Beyond. *arXiv preprint arXiv:1901.05350*. Available from: http://arxiv.org/abs/1901.05350.

Snap Inc., 2020. Snapchat [online]. Available from: https://www.snapchat.com.

Snap Inc., 2021. Anime Style by Snapchat [online]. Available from: https://lens.snapchat.com/b8c89687c5194c3fb5db63d33eb04617 [Accessed 07 September 2021].

Song, S., Lichtenberg, S. P. and Xiao, J., 2015. Sun rgb-d: A rgb-d scene understanding benchmark suite. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 567–576.

Sorensen, T. A., 1948. A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons. *Biol. Skar.*, 5, 1–34.

Statista Research Department, 2018. • Number of smartphone users worldwide 2014-2020 — Statista [online]. Available from: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ [Accessed 15 September 2019].

Suay, H. B. and Chernova, S., 2011. Humanoid Robot Control Using Depth Camera.

Proceedings of the 6th International Conference on Human-Robot Interaction, New York, NY, USA: Association for Computing Machinery, HRI '11, 401–402. Available from: https://doi.org/10.1145/1957656.1957802.

Suwajanakorn, S., Hernandez, C. and Seitz, S. M., 2015. Depth from focus with your mobile phone. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, IEEE, volume 07-12-June, 3497–3506. Available from: http://ieeexplore.ieee.org/document/7298972/.

Suwajanakorn, S., Snavely, N., Tompson, J. and Norouzi, M., 2018. Discovery of latent 3D keypoints via end-to-end geometric reasoning. *Advances in Neural Information Processing Systems*, 2018-Decem, 2059–2070. Available from: http://arxiv.org/abs/1807.03146.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. and Wojna, Z., 2016. Rethinking the Inception Architecture for Computer Vision. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Los Alamitos, CA, USA: IEEE Computer Society, volume 2016-Decem, 2818–2826. Available from: https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.308.

Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A. and Le, Q. V., 2019. Mnasnet: Platform-aware neural architecture search for mobile. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2019-June, 2815–2823. Available from: https://github.com/tensorflow/tpu/.

Tan, M. and Le, Q. V., 2019. EfficientNet: Rethinking model scaling for convolutional neural networks. *36th International Conference on Machine Learning, ICML 2019*, 2019-June, 10691–10700. Available from: http://arxiv.org/abs/1905.11946.

Taraba, M., Adamec, J., Danko, M. and Drgona, P., 2018. Utilization of modern sensors in autonomous vehicles. *2018 ELEKTRO*, 1–5.

Tekin, B., Bogo, F. and Pollefeys, M., 2019. H+O: Unified Egocentric Recognition of 3D Hand-Object Poses and Interactions. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 4511–4520. Available from: http://arxiv.org/abs/1904.05349.

Tekin, B., Sinha, S. N. and Fua, P., 2018. Real-Time Seamless Single Shot 6D Object Pose Prediction. *Proceedings of the IEEE Computer Society Conference on Computer*

Vision and Pattern Recognition, IEEE, 292–301. Available from: http://arxiv.org/abs/1711.08848.

Tesla, 2021. Autopilot — Tesla UK [online]. Available from: https://www.tesla.com/en_GB/autopilot [Accessed 20 September 2021].

The Linux Foundation, 2021. ONNX — Home [online]. Available from: https://onnx.ai/ [Accessed 30 October 2021].

TikTok, 2022. TikTok - Apps on Google Play [online]. Available from: https://play.google.com/store/apps/details?id=com.zhiliaoapp.musically&hl=en&gl=US [Accessed 02 April 2022].

Tong, W.-S., Tang, C.-K., Brown, M. S. and Xu, Y.-Q., 2007. Example-Based Cosmetic Transfer. *15th Pacific Conference on Computer Graphics and Applications (PG'07)*, 211–218.

Travelwayoflife, 2012. Nepali Lady at Ghyaru — www.travelwayoflife.com — Flickr [online]. Available from: https://www.flickr.com/photos/travelwayoflife/8052522076/ [Accessed 21 November 2021].

Ujiie, T., Hiromoto, M. and Sato, T., 2018. Interpolation-Based Object Detection Using Motion Vectors for Embedded Real-time Tracking Systems. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 729–7298.

Ulyanov, D., Vedaldi, A. and Lempitsky, V., 2016. Instance Normalization: The Missing Ingredient for Fast Stylization. *arXiv preprint arXiv:1607.08022*. Available from: http://arxiv.org/abs/1607.08022.

Unity Technologies, 2021a. GitHub - Unity-Technologies/barracuda-release [online]. Available from: https://github.com/Unity-Technologies/barracuda-release [Accessed 13 October 2021].

Unity Technologies, 2021b. GitHub - Unity-Technologies/ml-agents: Unity Machine Learning Agents Toolkit [online]. Available from: https://github.com/Unity-Technologies/ml-agents [Accessed 15 October 2021].

Unity Technologies, 2022a. Mobile Game Engine for Mobile Game Developers — Unity [online]. Available from: https://unity.com/features/mobile [Accessed 20 March 2022].

Unity Technologies, 2022b. Unity Real-Time Development Platform — 3D, 2D VR & AR Engine [online]. Available from: https://unity.com/ [Accessed 20 March 2022].

Valueva, M. V., Nagornov, N. N., Lyakhov, P. A., Valuev, G. V. and Chervyakov, N. I., 2020. Application of the residue number system to reduce hardware costs of the convolutional neural network implementation. *Mathematics and Computers in Simulation*, 177, 232–243.

Veit, A., Wilber, M. and Belongie, S., 2016. Residual Networks Behave like Ensembles of Relatively Shallow Networks. *Proceedings of the 30th International Conference on Neural Information Processing Systems*, Red Hook, NY, USA: Curran Associates Inc., NIPS'16, 550–558.

Wadhwa, N., Garg, R., Jacobs, D. E., Feldman, B. E., Kanazawa, N., Carroll, R., Movshovitz-Attias, Y., Barron, J. T., Pritch, Y. and Levoy, M., 2018. Synthetic depth-of-field with a single-camera mobile phone. *ACM Transactions on Graphics*, 37 (4), 1–13. Available from: http://dl.acm.org/citation.cfm?doid=3197517.3201329.

Wagner, D., Reitmayr, G., Mulloni, A., Drummond, T. and Schmalstieg, D., 2010. Real-Time Detection and Tracking for Augmented Reality on Mobile Phones. *IEEE Transactions on Visualization and Computer Graphics*, 16 (3), 355–368.

Wan, A., Dai, X., Zhang, P., He, Z., Tian, Y., Xie, S., Wu, B., Yu, M., Xu, T., Chen, K., Vajda, P. and Gonzalez, J. E., 2020. FBNetV2: Differentiable Neural Architecture Search for Spatial and Channel Dimensions. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 12962–12971.

Wang, C., Xu, D., Zhu, Y., Martín-Martín, R., Lu, C., Fei-Fei, L. and Savarese, S., 2019a. Densefusion: 6d object pose estimation by iterative dense fusion. *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 3343–3352.

Wang, N., Shi, J., Yeung, D.-Y. and Jia, J., 2015. Understanding and diagnosing visual tracking systems. *Proceedings of the IEEE international conference on computer vision*, 3101–3109.

Wang, R., Cheng, M., Chen, X., Tang, X. and Hsieh, C.-J., 2021a. Rethinking Architecture Selection in Differentiable NAS. *International Conference on Learning Representations (ICLR)*. Available from: http://arxiv.org/abs/2108.04392.

Wang, R., Pizer, S. M. and Frahm, J.-M., 2019b. Recurrent Neural Network for (Un-)supervised Learning of Monocular VideoVisual Odometry and Depth. *Proceedings*

of the IEEE Conference on Computer Vision and Pattern Recognition, 5555–5564. Available from: http://arxiv.org/abs/1904.07087.

Wang, S., Lu, H. and Deng, Z., 2019c. Fast object detection in compressed video. *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 7104–7113.

Wang, X., Yu, K., Wu, S., Gu, J., Liu, Y., Dong, C., Qiao, Y. and Loy, C. C., 2019d. ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks. L. Leal-Taixé and S. Roth, eds., *Computer Vision – ECCV 2018 Workshops*, Cham: Springer International Publishing, 63–79.

Wang, X., Yu, K., Wu, S., Gu, J., Liu, Y., Dong, C., Qiao, Y. and Loy, C. C., 2021b. xinntao/ESRGAN: ECCV18 Workshops - Enhanced SRGAN. Champion PIRM Challenge on Perceptual Super-Resolution. The training codes are in BasicSR. [online]. Available from: https://github.com/xinntao/ESRGAN [Accessed 31 August 2021].

Weng, Y., Zhou, T., Li, Y. and Qiu, X., 2019. NAS-Unet: Neural Architecture Search for Medical Image Segmentation. *IEEE Access*, 7, 44247–44257.

Wofk, D., Ma, F., Yang, T. J., Karaman, S. and Sze, V., 2019. FastDepth: Fast monocular depth estimation on embedded systems. *Proceedings - IEEE International Conference on Robotics and Automation*, Institute of Electrical and Electronics Engineers Inc., volume 2019-May, 6101–6108.

Wofk, D., Ma, F., Yang, T. J., Karaman, S. and Sze, V., 2020. FastDepth [online]. Available from: http://fastdepth.mit.edu/ [Accessed 19 August 2020].

Wu, B., Dai, X., Zhang, P., Wang, Y., Sun, F., Wu, Y., Tian, Y., Vajda, P., Jia, Y. and Keutzer, K., 2019. FBNET: Hardware-aware efficient convnet design via differentiable neural architecture search. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2019-June, 10726–10734.

Xiang, Y., Schmidt, T., Narayanan, V. and Fox, D., 2018. PoseCNN: A Convolutional Neural Network for 6D Object Pose Estimation in Cluttered Scenes. *Robotics: Science and Systems XIV*, Robotics: Science and Systems Foundation. Available from: http://arxiv.org/abs/1711.00199http://www.roboticsproceedings.org/rss14/p19.pdf.

Xiao, J., Owens, A. and Torralba, A., 2013. Sun3d: A database of big spaces recon-

structed using sfm and object labels. *Proceedings of the IEEE international conference on computer vision*, 1625–1632.

Yakobchuk, O., 2021. Serine Senior Lady with Wrinkles on Face Stock Photo - Image of modern, elderly: 90249112 [online]. Available from: `https://www.dreamstime.com/stock-photo-serine-senior-lady-wrinkles-face-portrait-calm-wrinkled-old-woman-looking-camera-friendly-smile-isolated-copy-image90249112` [Accessed 09 November 2021].

Yu, C., Gao, C., Wang, J., Yu, G., Shen, C. and Sang, N., 2021. BiSeNet V2: Bilateral Network with Guided Aggregation for Real-Time Semantic Segmentation. *International Journal of Computer Vision*. Available from: `https://doi.org/10.1007/s11263-021-01515-2`.

Zaeemzadeh, A., Rahnavard, N. and Shah, M., 2021. Norm-Preservation: Why Residual Networks Can Become Extremely Deep? *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43 (11), 3980–3990.

Zhang, H., Chen, W., He, H. and Jin, Y., 2019. Disentangled Makeup Transfer with Generative Adversarial Network. *arXiv preprint arXiv:1907.01144*. Available from: `http://arxiv.org/abs/1907.01144`.

Zhang, L., Wei, L., Shen, P., Wei, W., Zhu, G. and Song, J., 2018. Semantic SLAM based on object detection and improved octomap. *IEEE Access*, 6, 75545–75559.

Zhang, X., Xu, H., Mo, H., Tan, J., Yang, C., Wang, L. and Ren, W., 2021. Dcnas: Densely connected neural architecture search for semantic image segmentation. *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 13951–13962.

Zhao, Z., Peng, G., Wang, H., Fang, H.-S., Li, C. and Lu, C., 2018. Estimating 6D Pose From Localizing Designated Surface Keypoints. *arXiv preprint arXiv:1812.01387*. Available from: `http://arxiv.org/abs/1812.01387`.

Zhou, T., Brown, M., Snavely, N. and Lowe, D. G., 2017. Unsupervised learning of depth and ego-motion from video. *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017-Janua, 6612–6621.

Zhu, J. Y., Park, T., Isola, P. and Efros, A. A., 2017. Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks. *Proceedings of the IEEE International Conference on Computer Vision*, volume 2017-Octob, 2242–2251. Available from: `http://arxiv.org/abs/1703.10593`.

Zhu, M. and Liu, M., 2018. Mobile Video Object Detection with Temporally-Aware Feature Maps. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 5686–5695.

Zhu, Z., Liu, C., Yang, D., Yuille, A. and Xu, D., 2019. V-NAS: Neural Architecture Search for Volumetric Medical Image Segmentation. *Proceedings - 2019 International Conference on 3D Vision, 3DV 2019*, IEEE, 240–248.

Zoph, B. and Le, Q. V., 2017. Neural architecture search with reinforcement learning. *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, International Conference on Learning Representations, ICLR. Available from: `http://arxiv.org/abs/1611.01578`.

Zoph, B., Vasudevan, V., Shlens, J. and Le, Q. V., 2018. Learning transferable architectures for scalable image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 8697–8710.

Appendix A

Publications

Valentin Miu and Oleg Fryazinov. 2020. Real-Time Monocular 6DoF Pose Tracking for Handheld Objects on Smartphone GPUs. ECCV 2020 - Demos.

Valentin Miu and Oleg Fryazinov. 2020. Real-Time Monocular 6DoF Object Pose Tracking on Smartphone GPUs. CVMP 2020 - Short Papers. https://www.cvmp-conference.org/files/2020/short/17.pdf

Valentin Miu and Oleg Fryazinov. 2019. Accelerating Handheld Object Pose Estimation on Smartphones in Unity with TensorFlow Lite. CVMP 2019 - Short Papers. https://www.cvmp-conference.org/files/2019/short/56.pdf

Valentin Miu and Oleg Fryazinov 2018, Real-time 3D Smoke Simulation with Convolutional Neural Network-based Projection Method, CVMP 2018 - Short Papers

Appendix B

Extending TensorFlow Lite with new GPU operations

As mentioned previously in Subsubsection 2.4.3, in TensorFlow Lite inference, layers (formed from one or more tensor operations, or *ops*) can either be run on the CPU, or on a hardware accelerator (like the GPU, or some other DSP of the device), by means of a delegate. The latter requires that all the ops be supported by the delegate. While most TensorFlow operations can be replicated in TensorFlow Lite by means of one or more TensorFlow Lite CPU operations, the equivalent delegate op is often not available, and needs to be implemented to fully run the associated model on the GPU.

In general, when attempting delegate execution of a model with one or more ops unsupported by that delegate, TensorFlow Lite will run part of the model on the CPU (including the unsupported ops), and part on the GPU or DSP. This is not always the case - for example if the output tensor size of an op in the model varies independently of the model input size, leading to a so-called *dynamic* output tensor, this causes the entire delegate execution to be aborted. The model must then be run on the CPU in its entirety, or broken up into multiple models, with only the supported partial models being run on the GPU. In either case, both the total or partial CPU inference lead to unacceptable latency increases in most cases, especially if the CPU-executed portion is not INT8-quantized for lower latency.

During investigation of alternate neural network architectures for higher precision and better latency, a number of relevant works were looked into that used more exotic convolutions, which were not available at the time as TensorFlow Lite GPU ops, and often not even CPU ops either. To allow such works to be tested and potentially used, there was a desire to extend the available TensorFlow Lite GPU operations.

There was also a more immediate reason for this, given by the inability of Ten-

sorFlow Lite to do GPU inference for one of the machine learning models in use at the time for the virtual try-on app. These unusual op requirements were due to the model requiring dynamic cropping of some regions of interest internally.

In the GPU operations of TensorFlow Lite, the tensor sizes are generally fixed, and cannot be changed except on a model-wide basis, by changing the model input and output sizes, and triggering a recalculation of the new intermediate tensor sizes. In the case of cropping the region of interest, the crop output size was variable, creating a dynamically-sized tensor, even if this was then resized to a fixed size. This tensor size was determined by the previous layers of the model, so the model-wide resizing was not useful in this case. To resolve this, the cropping and resizing operations were combined into a single op, which removed the intermediate dynamic tensor from between them. This was done by taking advantage of an operation combination step, referred to as *op fusion*, performed by the GPU delegates in TensorFlow Lite for optimization purposes, as described in Subsection B.1. This method also allowed the combined op to not require a corresponding TensorFlow Lite CPU or TensorFlow layer, as will be shown for the CROP_RESIZE operation.

In our efforts to add ops, we address only the OpenGL, OpenCL and Metal delegates, as they cover most of our GPU use cases. This implies three new shaders per added op. Since our target delegates have all or almost all the same ops already implemented, to ensure feature parity between DSPs and platforms, the starting points for our efforts are largely the same for each GPU delegate.

B.1 Graph optimizations for op fusion or removal

In TensorFlow Lite, as in TensorFlow, the network is represented as a directed acyclic graph, where the nodes are tensor operations, and the directed edges indicate the intermediate tensors and the data flow between operations.

Before the graph is run by the TensorFlow Lite GPU delegates, it is modified by *fusing* certain recognized sequences of operations into single operations. This is done either for optimization purposes, or to simplify delegate implementation. Sometimes, some of the unfused ops have no corresponding GPU ops, and cannot be run without the fusing step.

Some existing optimizations and fusions involved the merging of padding operations with convolutions and other ops with built-in padding support, the merging of convolutions with additions and multiplications (in the absence of nonlinearities between them), and the removal of identity operations like single-input additions and concatenations.

Aside from helping implement the CROP_RESIZE operation, fusing ops can also help with casting problems in TensorFlow Lite GPU. These arise from TensorFlow Lite GPU only supporting float tensors, which causes issues for implementing new operations with integer-output operations, like ARGMAX (which returns tensor indices). By placing a casting operation from int to float afterward, and then fusing the two into a single op, a float-output ARGMAX can be implemented in TensorFlow Lite GPU instead. This technique will also be used for the NOT_EQUAL and GREATER_THAN ops, which normally have unsupported boolean outputs.

B.2 Ops added

The ops added are listed below. With the exception of CROP_RESIZE, these ops either had a TensorFlow Lite CPU version with no GPU equivalent implemented, or had a more restricted GPU implementation unsuitable for the model that needed to be run.

FLOOR_DIV

This is the floor of the elementwise division between two tensors, meaning that in the general case, it follows Equation (B.1).

$$\begin{aligned} R(b+\delta b, x+\delta x, y+\delta y, c+\delta c) = OP(&A_1(b+\delta b_1, x+\delta x_1, y+\delta y_1, c+\delta c_1),\\ &A_2(b+\delta b_2, x+\delta x_2, y+\delta y_2, c+\delta c_2),\\ &\ldots\\ &A_N(b+\delta b_n, x+\delta x_n, y+\delta y_n, c+\delta c_n)) \end{aligned} \tag{B.1}$$

Since elementwise operations are treated as a special case in the code, due to the similarities between them, they are generally relatively easy to implement. However, some care must be taken to implement both the FLOAT32 and FLOAT16 (half float) correctly.

NOT_EQUAL, GREATER_THAN

While these are elementwise operations as well, in the form required by their equivalent TensorFlow Lite CPU ops, they have a boolean type output tensor. Since the GPU ops only support float or half float tensors for any of the relevant delegates, they cannot be implemented directly.

The solution used here is to apply a casting operation from boolean (or integer, in some of the other ops implemented) to float, implement the NOT_EQUAL and

GREATER_THAN operations with float and half float outputs, and remove the casting node through a graph optimization operation. Since only float and half float nodes are supported in TensorFlow Lite, there can be no valid casting operation in a TensorFlow Lite graph, outside a pointless identity operation. Therefore, the casting operation can be removed from the TensorFlow Lite graph in all cases, without the need for additional checks.

REDUCE_SUM

The REDUCE_SUM operation sums the elements of a tensor along a list of axes, so that the resulting dimensions are 1 element along those axes, and unchanged for the other axes. For a single axis, this op follows one of Equation (B.2), depending on the axis.

$$\begin{aligned} R_b(0,x,y,c) &= \sum_b I(b,x,y,c) \\ R_x(b,0,y,c) &= \sum_x I(b,x,y,c) \\ R_y(b,x,0,c) &= \sum_y I(b,x,y,c) \\ R_c(b,x,y,0) &= \sum_c I(b,x,y,c) \end{aligned} \tag{B.2}$$

For multiple axes, REDUCE_SUM is equivalent to the respective single-axis versions, applied in any order. As such, it was decided that only single-axis versions would be implemented, and only for the height, width and channel axes, given the requirements for the model that needed to be supported.

Special consideration was required to implement the channel-wise version of this op, due to the sampled values being 4-floats ($vec4$) for speed.

The output of the both the TensorFlow Lite and supported TensorFlow op are both float-like (float or half-float), so there are no casting considerations to be made.

REDUCE_MAX

This op returns the maximum values of a tensor along a given axis, as given by Equation (B.3).

$$
\begin{aligned}
R_y(0, x, y, c) &= max_b(I(b, x, y, c)) \\
R_y(b, 0, y, c) &= max_x(I(b, x, y, c)) \\
R_y(b, x, 0, c) &= max_y(I(b, x, y, c)) \\
R_c(b, x, y, 0) &= max_c(I(b, x, y, c))
\end{aligned}
\tag{B.3}
$$

The shader implementation is similar to REDUCE_SUM, and the output is again float-like.

ARGMAX

This is similar to REDUCE_MAX, except the indices of the maximum values are returned, instead of the values themselves. Unlike for REDUCE_MAX, the multiple-axis version cannot be easily reproduced using multiple single-axis versions. However, the single-axis versions were enough for the purposes of the model, so they were the only versions implemented, again with the exception of the single batch axis.

As it returns indices, the argmax operation in TensorFlow has an integer output. This is again handled with a casting operation on the TensorFlow side, which is removed on the TensorFlow Lite side.

When testing the half float version of this op with random inputs, occasional differences were noticed between the float (full float32) implementation, and by extension the float32 TensorFlow layer. This was due to differences between axis maximums and the runner-up values being too small to be resolved at half float precision, thereby appearing equal, while being resolvable at float32 precision. Since this is not technically incorrect and anyway cannot be "fixed" at half precision, it was left unchanged, and the testing method was updated to account for this by comparing the values at the indices, as opposed to the indices themselves.

It should be noted that the TensorFlow argmax layer returns the smallest index in the case of equality of the two or more largest values, and this behaviour was retained in this TensorFlow Lite GPU implementation, so the differences are unrelated to this.

CONCAT_BATCH

Oftentimes, while a TensorFlow Lite CPU op may have a GPU equivalent, said equivalent is more limited in its operation. In the case of the CONCAT (concatenation) op, the GPU version cannot concatenate along the batch dimension. Support for batch concatenation was added to allow for concatenating single-dimension tensors, which appeared as a result of calculating the cropping parameters through tensor operations.

CROP_RESIZE

This op involves an XY-crop of the input tensor, followed by a bilinear XY resizing to a fixed output shape. As mentioned previously, this ensures the absence of the dynamic tensor by combining the crop and resize.

While there is an equivalent TensorFlow op ($tf.image.crop\_and\_resize$), there is no equivalent TensorFlow Lite op for the CPU. However, dynamic tensors are supported in TensorFlow Lite CPU inference, so it can be implemented as separate cropping and resizing operations, through STRIDED_SLICE and RESIZE_BILINEAR, respectively.

In this situation, there are two options for the GPU implementation. The first is to implement $tf.image.crop\_and\_resize$ both as CPU and GPU ops. The second is to implement the crop and resize as separate cropping and resizing ops on the TensorFlow side, so that it works on the CPU in TensorFlow Lite without any additional changes. The CROP_RESIZE op is then implemented a single GPU op, and the crop and resize operations are fused into this single op, using a graph-modifying optimization step before the model is run on the GPU.

The second method was chosen, as the optimization step appeared to be less complicated than implementing the single CPU op.

It is possible to have a STRIDED_SLICE followed by a RESIZE op that works on the GPU, if the slice parameters (the indices determining what part of the input tensor to keep) are fixed, thereby causing the intermediate tensor to not be dynamic. While this is not our use case, to avoid other model inferences possibly breaking, the optimization step needed a check on the number of inputs to the STRIDED_SLICE (cropping) op, as multiple inputs indicate a variable output tensor size.

B.3 Cost analysis

Since the TensorFlow Lite library is used from within Unity, certain GPU operations can be implemented on the Unity side as HLSL shaders. Alternatively, they can be replicated on the CPU with C++ libraries using OpenCV.

Maintenance of the new GPU ops proved too difficult, as merging in a newer official commit of TensorFlow broke most of them. Most of all, the CROP_RESIZE op, as it required the most and most fundamental changes, was too difficult to maintain. In retrospect, the first CROP_RESIZE implementation option, involving the implementation of the TensorFlow Lite CPU op, may have been more stable.

In the model considered, all the new ops are in only one specific section, and address the aforementioned region-of-interest cropping. As such, even a single op requiring to be run outside the GPU delegate would significantly lower the speed of the

model by requiring a GPU-CPU tensor copy, running the op in question on the CPU, and another CPU-GPU copy to run the rest of the model. If the op can be written as a Unity shader, which requires the model split into multiple submodels, then the expensive CPU-GPU copies can be avoided, as TensorFlow Lite does allow accessing externally written GPU memory. Furthermore, having a Unity-side system handling a sequence of multiple models simplifies the swapping and testing of each model. This alone was deemed to be enough of a benefit to opt for the Unity-side running, even without knowing if a Unity shader can be used instead of a CPU implementation with OpenCV.

Abbreviations

6DoF Six Degrees of Freedom.
AdaIN Adaptive Instance Normaslization.
API Application Programming Interface.
AR Augmented Reality.
CGI Computer Generated Imagery.
CPU Central Processing Unit.
CVML Computer Vision with Machine Learning.
D2C Direct-to-Consumer.
DSP Digital Signal Processor.
FID Frechet Inception Distance.
FLOP Floating point operation.
GAN Generative Adversarial Network.
GPU Graphics Processing Unit.
HLSL High-Level Shading Language.
HSV Hue, Saturation, Value.
LeakyReLU Leaky Rectified Linear Unit.
LIDAR Light Detection and Ranging.
MAdd Multiply-add.
mIOU Mean Intersection Over Union.
ML Machine Learning.
NAS Neural Architecture Search.
NNAPI Neural Network Application Programming Interface.
PnP Perspective and Point.
ReLU Rectified Linear Unit.
SIFT Scale-Invariant Feature Transform.
SLAM Simultaneous Localization And Mapping.
SOTA State of the Art.
SSD Single Shot Detector.

TFLT TensorFlow-Lite-Tester.

VR Virtual Reality.

VRAM Video Random Access Memory.