ONTARIO TECH UNIVERSITY
FACULTY OF SCIENCE

Group 3 December 8, 2024

# CinemaShots
# Real-Time Image and Video Processing Through Filter Style Transfer

## by

Kershan Arulneswaran Parsa Zahraei Mohammadabady

**Abstract:** Achieving effective filter style transfer is a challenging and intriguing problem in computational photography. To this end, a team of researchers from Samsung's Visual Solution Lab proposed a method to train and apply a CNN model to extract and apply filter parameters from a single input image. This report covers the bases, methodologies, and results from which we trained a version of the CNN model described by the researchers, in addition to constructing a lightweight and portable model inspired by their solution, which is ideal for use in mobile applications. Our locally tained model performed well on filter style transfers on filters contained within its datasets and similar filters but struggled with filters that contained unusual parameters. Our mobile filter was successful in extracting and storing filter parameters for easy application to targets, but we encountered difficulty in fully implementing it in the context of a mobile application. Insights gained from our endeavors are the importance of diversity in constructing a dataset, and the technical challenges faced when implementing a machine learning model in the development of a mobile app.

# 1 Introduction

The inspiration and source of knowledge for this project stems from the paper Filter Style Transfer (FST) between Photos by Yim et Al for Samsung's Visual Solution's Lab. The paper discusses a novel method for performing filter style transfers, in which the style of a filtered image can be extracted and applied to a target image. This innovative technique is achieved through the training of a convolutional neural network (CNN), referred to in the paper as a "defilterization network" [1]. The paper emphasizes the applicability of the technique on mobile devices, as it does not require extensive computing resources. Based on the methodologies described in the paper and the GitHub coded published the researchers, we have successfully trained a defilterization convolutional neural network model, in addition to developing a similar portable model for use in a mobile app where users can extract filters and overlay it on their camera feed for ease of application (although this full implementation of the model in-app ultimately did come to fruition). This report will outline the methodologies used to create our two models, the methodology of the development and implementation of the mobile application, and display the results of the defilterization network and of the mobile application model. The discussion shall cover the efficacy of our models, and explore potential improvements to their implementation.

# 2 Models Methodology

## 2.1 Defilterization Model as described in the paper

The basis of the FST technique is the extraction of a filter from a single reference image $(I)$, and the application of the filter to a target image $(X)$ . This process involves the restoration of the original image $(\hat{I})$ from $(I)$ in a step named defilterization. The two images are subsequently used to derive a filter parameter $w$ in a step called filter style estimation, from which a filter function $f_w$ is created for the application of the filter to the target $(X)$ [1].

This crucial process of defilterization is made possible, efficient, and effective through the use of a convolutional neural network autoencoder, $f_w^{-1}$. The authors describe a training regiment by which images from the popular dataset MSCOCO where combined with 26 instagram filters with publicly available source code through the project CSSgram, in addition to synthetic filters to create a var-

ied dataset upon which the model was trained. The defilterization model was validated against a set of 100 randomly selected images, from which 1800 filtered images where generated through the application of 188 CSSgram filters (8 filters where excluded due to their production of vignetting effects) [1].

## 2.2 Defilterization Model Recreation

The GitHub code associated with this paper contains files that outline the architecture of the CNN, contains functions to perform filter style transfer and various related operations, and a file ”main_inv_function.py” that is used to train a defilterization CNN [2]. The repository does not contain any pre-trained models nor the requisite training data, so it is incumbent on those who wish to construct a model of their own to manufacture the datasets themselves. For our purpose, we closely followed the data generation process outlined in the paper by downloading the 2017 Train images dataset from the MSCOCO website and using a JavaScript file to programmatically apply the filters obtained from CSSgram. Due to the vagueness of the paper pertaining to the number and nature of synthetic filters generated, synthetic filter data was not added to the training mix. Although a specific training set size was not specified in the paper, we selected 1000 images from the MSCOCO dataset - $10\times$ the amount of base validation images - and applied all 26 filters to each for a total training set size of 26,000 filtered images. Also important to training are text files containing lines of the following structure: ”imageName.jpg filterName_imageName.jpg” which associate an unfiltered image file name to its filtered counterpart. These text files were generated using Python scripts.

Once the training and testing data was generated, a PowerShell script file was written to run main_inv_function.py on a Windows 11 System with an NVIDIA RTX 3050 Laptop GPU. The aforementioned python script makes use of NVIDIA specific Pytorch commands, so it is necessary for a user to train the model on a system with a NVIDIA GPU or on a remotely hosted virtual machine. Training took approximately 12 hours, with the process being interrupted once and having to start again from a checkpoint. Training lasted for 9 epochs, of which 7 (including the last iteration) were saved as models.

### 2.3 Mobile App Defilterization Model

Compared to the original model, there are several tweaks in the mobile model. The most important part of this model is that it uses TensorFlow instead of PyTorch, which is then converted into a TensorFlow Lite model. The dataset is split into 4 folders: training_data and testing_data have all of the filtered images, while training_subset and testing_subset have the original unfiltered images.

To prepare the dataset, all the image paths were put into an array and then shuffled with a random function. Then, the number of paths was reduced to 800 for testing and 200 for training to ensure safe data processing in Google CoLab. NumPy [3] zero arrays were later created for both the regular and filtered images after that. With the concurrent.futures [4] library, a results list was created with the executor mapped to the connect_to_ground_truth function, which includes a process_image function. With OpenCV [5], the process function converts into an RGB image, then resizes the image to 64x64. Once the filtered image is processed, it finds the original by splitting the image path so that the filter is gone. The original image then goes through the same process function before both are stored in the ndarrays. Finally, each image is augmented, resized again to 64x64 and flattened to a 1x4096x3 tensor to increase model complexity.

For the model and the loss function that was used in the project, it was mostly similar to the original code, but because Conv2D is considered to be a SELECT_OP for TensorFlow Lite from experience, it instead uses 1D functions for convolution and upsampling. Because of this using 1D, 256x256 will crash CoLab as well, so 64x64 was used here. After the model is trained with 20 epochs, it was converted to TensorFlow Lite to be used for the application.

## 3 Mobile Application Methodology

When it comes to building an interface for the project, a mobile application suits this idea better than a website, since the project is aiming towards greater accessibility. For cross-platform frameworks, there were two potential choices that could be used for the project: React Native and Flutter. React Native is a framework created by Meta in 2015 that uses the popular React framework based on either JavaScript or TypeScript to create native applications for various platforms, like iOS and Android [6]. Meanwhile, Flutter is Google's framework that creates cross-platform applications like React Native, but instead of using JavaScript or

TypeScript, it uses Google's own programming language called Dart [7]. Flutter is also the framework used for the university's own Mobile Devices course, which both team members did not take. Ultimately, because of its familiarity and more mature platform, React Native was selected, with TypeScript as the language of choice, as it combines the flexibility of JavaScript with static types from other programming languages.

When the user installs the application, it would first ask for permissions to enable camera, microphone and the gallery. Once those permissions are granted, the user can interact with the application in various ways. Most of the screen is taken up by the camera viewfinder, with the bottom section being used for controls. At the top of the section, there is a zoom slider that the user can use to increase or decrease magnification of the resulting media. Below that, there are six separate buttons for interaction. The bottom middle button is used to capture the image or photo, while the top middle button toggles between the two modes. The top left button redirects to the user's library, the top right redirects to settings and the bottom right allows the users to change the camera used from the rear to the front and vice versa. The bottom left button is a special button, where one selects a photo, which the app stores as a URI. It then gets passed to a function that extracts the filter via the model that was explained in the previous section into a .yaml file in a separate function consisting of the mean RGB values. Then, the .yaml file is processed into a variable which is then passed to the frame processor. The frame processor in the code creates a colour filter via a 5x4 matrix, with the columns indicating R,G,B,A and Offset. Finally, this filter is applied via React-Native-Skia [8] to the camera's viewfinder and the resulting piece of media.

Some notable npm libraries and tutorials were used to build this application. Despite being familiar with frameworks like React and Next.js through building websites and side projects, this was the first time that both members created a mobile app. To get started, they followed a video tutorial on building a React Native Camera app by YouTuber and Expo contributor Alberto Moedano [9]. Portions of the tutorial code, such as the original Permissions page and the button component, were adapted for use in the project. However, the majority of the codebase was developed independently by the team members.

As for libraries, there are a few that should be discussed, the most notable one being the React-Native-Vision-Camera library [10]. This library is the standard when it comes to open source camera applications and this was used for the camera view and the various functions like taking photos and recording a video,

in addition to various permissions for the camera and the phone's microphone. React-Native-Image-Picker [11] is another library that was used for the application at this allows the user to pick an image. This specific library was used to take a photo and extract the colour image from the filter. Finally, to store the media to the user's gallery, Expo-Media-Library was used [12]. Additional libraries include React-Native-Skia, React-Native-Worklets-Core [13] for frame processing and React-Native-Fast-Tflite [14] for Tensorflow.

# 4 Results

## 4.1 Recreated Defilterization Model Results

To test our trained model, a PowerShell script was written to invoke main_stylizer.py, passing to the python script arguments including the directory containing the stylized images whose filters are to be extracted and the directory containing the images to be stylized. To demonstrate the effectiveness of the model, we will be comparing its results in restoring an unfiltered image from an input image and the results of filter style transfer to a target image using three filters: the 1977 filter from CSSgram, beFunky's Sepia 1 filter, and a synthetic filter created by arbitrarily adjusting color, contrast, and hue controls.

### 4.1.1 1977 Filter Results

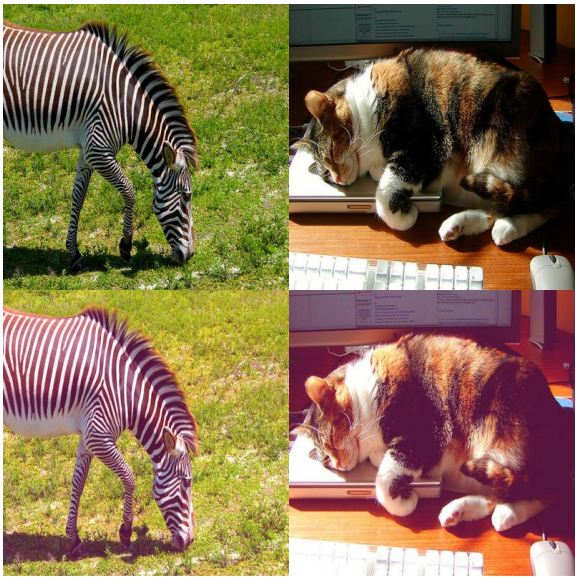

Figure 1: Top from left: original input image, target image, input image, ground truth target image

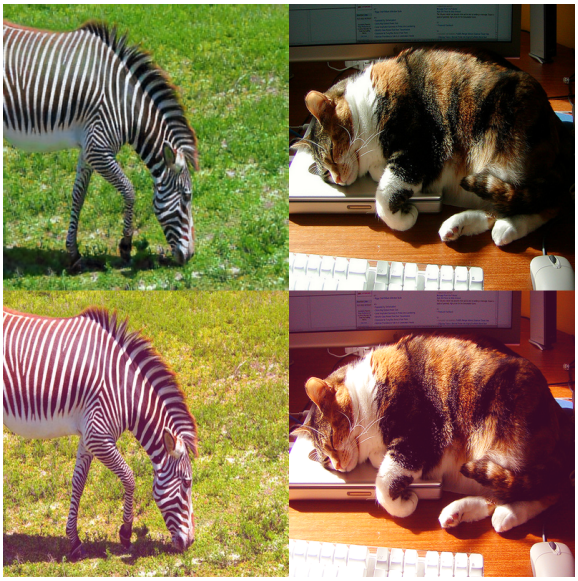

Figure 2: Top from left: Defilterized input image, target image, input image, stylized target image

The 1977 filter comes from the CSSgram repository and was one of the filters used in the construction of the dataset. Figure 2 shows that the model was able to restore the original input image to great effect, and the resulting filter style transfer was highly accurate in capturing the colors and tones of the filter. A few deficiencies appear in the results, however. The restored image is somewhat cooler in tone than the original image, and while the colors of the stylized target image match that of the ground truth, it does not have the same washed-out look.

### 4.1.2 beFunky Sepia Filter Results

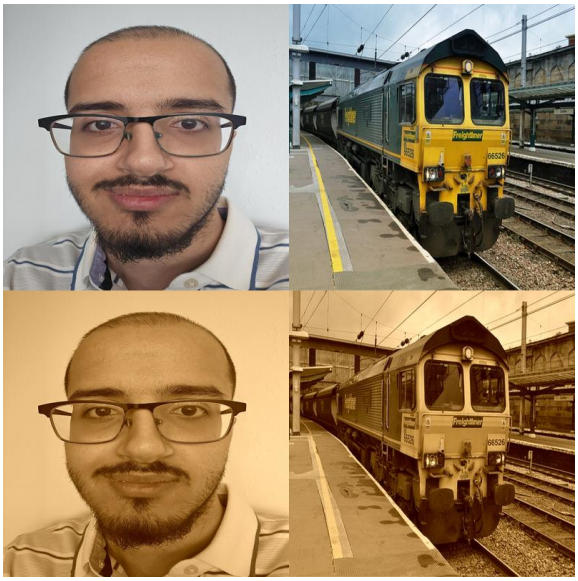

Figure 3: Top from left: original input image, target image, input image, ground truth target image

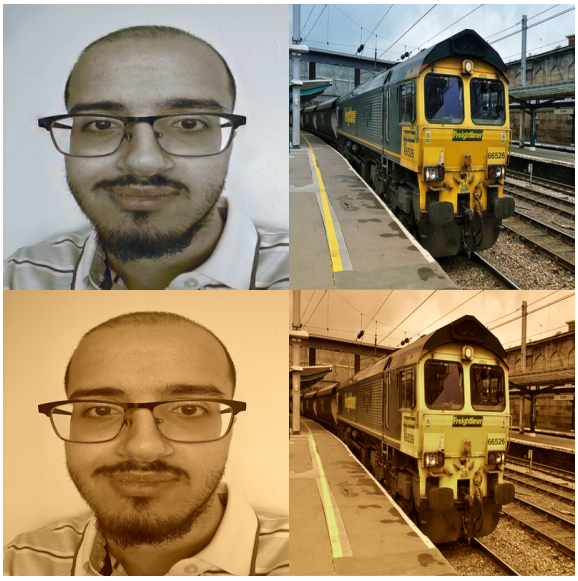

Figure 4: Top from left: Defilterized input image, target image, input image, stylized target image

The filter used here is a sepia filter available in the mobile app beFunky. Although the model was not as accurate in restoring the input image as in the previous example, with the displayed skin tone being an unnatural colour and there being an excess of blue hues, the filter style transfer was largely successful in applying the tones of the extracted filter to the target image. One flaw in the resulting stylized image is that compared to the ground truth image, the yellow parts of the train and platform floor in the input are more saturated than the rest of the train and background. Those flaws aside the overall result is satisfactory.

### 4.1.3 Synthetic Blanched Filter Results

Figure 5: Top from left: original input image, target image, input image, ground truth target image

Figure 6: Top from left: Defilterized input image, target image, input image, stylized target image

The model's performance on this filter is the most lackluster of all the results presented. Similar to results for the Sepia filter, the restored input image is far blue-shifted compared to the original, and its saturation is significantly lower. Unlike the strong transfer of the Sepia filter, the resulting stylized image fails to capture the chilled effect on display for the ground truth and input image, instead only appearing mildly cooler and bluer than the original. The filter's parameters involve the halving of saturation, a slight increase in contrast, and a substantial blue shift. These significant transformations potentially explain the model's difficulty in capturing its effects.

### 4.1.4 Mobile Filter Results

Figure 7: Input image of mobile model

Figure 8: Expected Output image

The mobile model's performance was decent in terms of extracting the image and gives off a similar vibe to the input image. The final .yaml outputs ranged around 0.43, 0.46 and 0.49 for r,g and b. Based on the matrix where the alpha values are r,g,b,1, it looks suboptimal. In the future, alpha values should be optimized to produce more lifelike images.

# 5 Discussion

## 5.1 Recreated Defilterization Model Discussion

In the results section, we showed that our locally trained Defilterization CNN model was for the most part accurately able to facilitate the extraction and application of CSSgram's 1977 filter and beFunky's Speia filter, but it faltered in its performance on the synthetic blanched filter. The model's best performance was on the 1977 filter; this is to be expected, as it was a filter used in the construction of the dataset. Although the reconstruction of the Sepia filter input image was less accurate compared to the restoration of the 1977 filter input, the filter style transfer was still well implemented onto the target image, owing to the optimizations in estimating the filter parameters in the research code. As noted in section 4.1.3, the parameters of the synthetic blanched contained a combination saturation and hue transformations that are perhaps uncommon in filters such as those deployed in app like

Instagram. There is definite room for improvement in this regard for the model, and the most obvious avenue for this improvement is to follow the researcher's method of supplementing CSSgram filter images with filtered images obtained by applying various real and arbitrary synthetic filters [1]. Increasing the diversity of the dataset is likely to increase the robustness and utility of the defilterization model.

Another factor in the model's performance was the magnitude of the training set. With limited computing resources available to us, training for 9+ epochs on a dataset of 26,000 filtered + 1000 original training images and 1800 filtered + 100 original testing images produced a model that delivered good to great results on transferring filter styless from filters in its dataset and other common filters used in mobile camera applications. With the addition of both greater computing resources that can handle training on an expanded dataset and a greater diversity of data, the potential for a model capable of facilitating seamless filter style on almost any filter remains within reach.

# 6 Conclusion

To conclude, our efforts in this project were to create two CNN defilterization models for use in the performance of filter style transfers. The first model was trained according to a recipe outlined in a seminal paper, constructing a large and robust dataset using source code for 26 popular Instagram filters. The second model was a simplified and portable iteration of the model described in the paper, making it well suited for use in real-time filter style transfer in the context of a mobile app. Our locally trained model based on the paper produced strong results in the performance of filter style transfer on filters contained within its datasets and similar filters from other mobile applications but struggled to replicate the effects of synthetic filters. Our mobile application model was able to effectively capture the stylings of common filters, but technical issues encountered in the development of the mobile app prevented its full deployment in that context. Overall, while there exists great room for improvement both in the training of a CNN model as described in the paper and in the implementation of home-grown model, this project served as a fruitful exercise in the construction of CNN models to tackle problems like filter style transfer.

# Works Cited

[1] J. Yim, J. Yoo, W. Do, B. Kim, and J. Choe, “Filter Style Transfer between Photos,” European Computer Vision Association, pp. 1–17, Jul. 2020, Accessed: Oct. 2024. [Online]. Available: https://www.ecva.net/papers/eccv$_2$020/$papers_E CCV/papers/12351010$

[2] jonghwa-yim, ”FilterStyleTransfer”, GitHub, 2020. [Online]. Available: https://github.com/jongh yim/FilterStyleTransfer

[3] NumPy documentation, https://numpy.org/doc/ (accessed Dec. 8, 2024).

[4]“concurrent.futures — Launching parallel tasks — Python 3.9.5 documentation,” docs.python.org. https://docs.python.org/3/library/concurrent.futures.html

[5] OpenCV, “OpenCV: OpenCV modules,” docs.opencv.org. https://docs.opencv.org/4.x/index.ht

[6] React Native, “React Native · A framework for building native apps using React,” reactnative.dev, 2022. https://reactnative.dev/

[7] Flutter, “Flutter - Beautiful native apps in record time,” Flutter.dev, 2024. https://flutter.dev/

[8] “React Native Skia — React Native Skia,” Github.io, 2024. https://shopify.github.io/react-native-skia/ (accessed Dec. 08, 2024).

[9] Code with Beto, “Building an Obscura Pro Camera App with React Native Vision Camera and Expo: A Step-by-Step Tutorial,” YouTube, Aug. 21, 2024. https://www.youtube.com/watch?v=xNaGYGDZ2JU (accessed Dec. 08, 2024).

[10] M. Rousavy, “VisionCamera Documentation,” react-native-vision-camera.com. https://react-native-vision-camera.com/

[11] “react-native-image-picker,” npm. https://www.npmjs.com/package/react-native-image-picker

[12] “MediaLibrary,” Expo Documentation. https://docs.expo.dev/versions/latest/sdk/media-library/

[13] Margelo, “GitHub - margelo/react-native-worklets-core: A library to run JS functions (‘Worklets’) on separate Threads,” GitHub, Nov. 07, 2024. https://github.com/margelo/r native-worklets-core/tree/main (accessed Dec. 08, 2024).

[14] M. Rousavy, “GitHub - mrousavy/react-native-fast-tflite: High-performance TensorFlow Lite library for React Native with GPU acceleration,” GitHub, Jul. 13, 2024. https://github.com/mrousavy/react-native-fast-tflite