CS329S Final Report: Developing A Real-Time Mobile Video Style Transfer App

Jihun Hong
Jaewoo Jang
Collin Kwon

Problem Definition

The goal of our project is to build a lean, real-time style transfer application for videos that runs on mobile devices using machine learning. Recently, we have seen the rise of new avenues of self-expression among Millennials and Generation Z, who have become mainstream users of the internet. As we can observe from the popularity of Instagram's photo filters, the young generation prefer uniqueness and style when they express themselves through pictures of videos. Also, we noticed that videos have become the major source of self-expression with the rise of social media platforms like TikTok, YouTube, and Instagram Reels, which use videos as the main form of shared content. While texts and photos have been popular for the last decade in platforms such as Twitter, Facebook, and Instagram, we believe that video is the fastest growing form of media on the internet in terms of popularity, especially along with technical developments that allow more efficient streaming and storage of video content. In addition, we believe that people spend a lot of time on mobile devices nowadays, especially when they are using social media. Also, users rely mostly on mobile devices for taking pictures and videos, so it made more sense that we create an application that runs on mobile devices for style transfer. Therefore, the challenge we are trying to solve is to create a cross-platform mobile application that could be run on both iOS and Android devices; we aim to use a small but efficient model that could be run on an edge device with low latency, because video style transfer requires faster inference compared to image style transfer.

System Design

In order to implement the application, we used Flutter to develop a cross-platform mobile application. The style transfer application can be easily built for mobile devices running on both iOS and Android. This approach, we believe, reduces the need to develop separate UI components for both mobile platforms, thereby reducing redundant workload. In order to access the camera module, we had to use native API. For this project, we focused on getting a working version fully working on an Android device. We created a mobile application that runs TensorFlow Lite inference on the edge. The reasons for adopting edge computation was to achieve low latency and enable the application to run even in offline mode. Although we could have designed the application to run inference online on a separate server, we decided to use a smaller ML model that could run on the edge device. Therefore, we searched and developed a less expressive model that can run fast enough on Android phones. Especially since we were creating an app that applies style transfer over video frames, fast inference was an important feature consideration when designing the system. Also, users may want to take videos and apply style transfer in situations where they don't have a strong internet connection, so we designed the entire pipeline to run on the device for enhanced user experience.

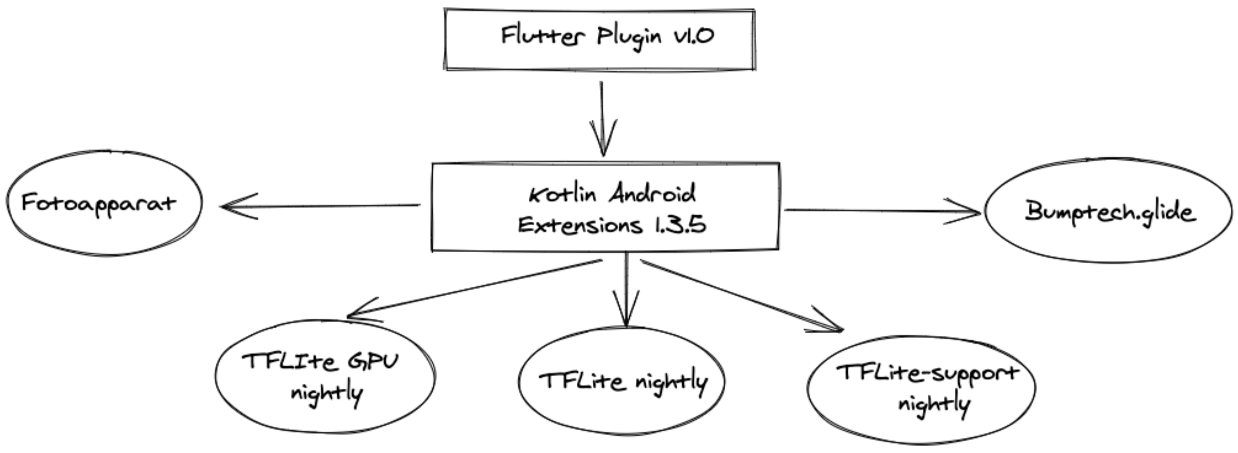


Figure 1: We used Flutter plugin for iOS and Android interoperability. Fotoapparat provided a more streamlined pipeline to use the Camera API 2. Bumptech.glide library was an easy-to-use UI library and we imported modules for TFLite integration.

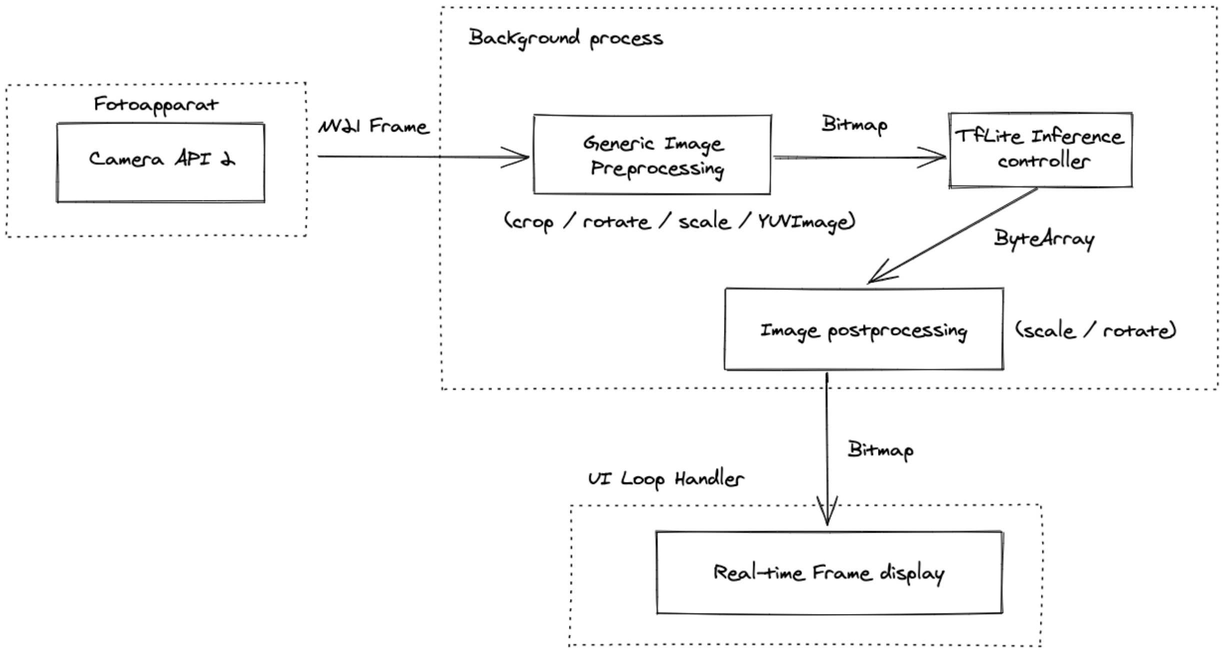


Figure 2: The bulk of our work was in optimizing our background process, optimizing frame pre-processing and Image output post-processing.

Therefore, we used TensorFlow Lite to create a small machine learning model and port the model over to Android mobile application. The model was trained offline on GCP, then the parameter values were saved inside the application to be used during inference. The model parameters are saved as tflite records in the asset directory of the application; when the application wants to run inference, we split the video into frames and run style inference and style transfer on a frame-by-frame basis. The mobile device's GPU is used whenever possible. When we're able to utilize the device's GPU, a 16-bit int model is used; otherwise, a 8-bit int model is used for faster performance. We decided to use 16-bit or 8-bit mixed precision models during inference because it significantly boosts performance without hurting much of the visual performance. Also, we successfully sped up the model by a factor of ~2 after using the device's GPU instead of its CPU. The TFlite's API was very helpful in implementing the model inference pipeline in Kotlin because it allowed us to automatically import the interpreter for the model, as well as model parameters, directly from the tflite record file. It enabled a seamless integration of the offline training pipeline into the deployment of the tflite model on the device, without worrying about rewriting the model architecture inside the mobile application.

Machine Learning Component

The machine learning model used for video style transfer consists of two major parts. The first part is a model for style prediction, which takes in a single style reference image and outputs an embedding vector that represents the style of that image. The second part is a model for style transfer, which takes in a content image and the style vector computed from the first model. During inference, the style reference image is passed on to the style prediction model, which outputs a 100 dimensional vector. This style vector is passed on to the style transfer model along with the content image, which is a single frame in the video, which then outputs a final image that has the style of the reference image and contains the content of the video frame.

The model was trained on the ImageNet dataset, which includes more than 1.3 million images with 155GB in total size. During training, three images are passed into the VGG network: style reference image, content image, and final output image. The VGG network is a neural network architecture based on Convolutional Neural Networks (CNN), which has achieved above 92% accuracy for the image classification for the ImageNet dataset. After the images are passed into the VGG network, two types of losses are calculated. The first loss is the style loss, which is the style similarity between the style reference image and the final output image. The second loss is the content loss, which is the Euclidean distance between the content image and the final output image. Our training loss is set to be the sum of the style loss and the content loss; the training objective is to minimize both the style and content loss calculated by the VGG network. Note that the VGG network is only used during the training phase for calculating the loss, but not during inference.

The first model, which is used for style prediction, is based on the MobileNet V2 network architecture. The MobileNet network is a deep learning model consisting of CNN layers and activation layers, with skip connections between the layers. The second model, which is used for style transfer, is a feed forward network that applies the normalized style embedding vector to the input content image. Both of these models have a relatively small number of parameters, with the style prediction model size being 4.7Mb for float16 and 2.8Mb for int8; the style transfer model size is 0.4Mb for float16 and 0.2Mb for int8.

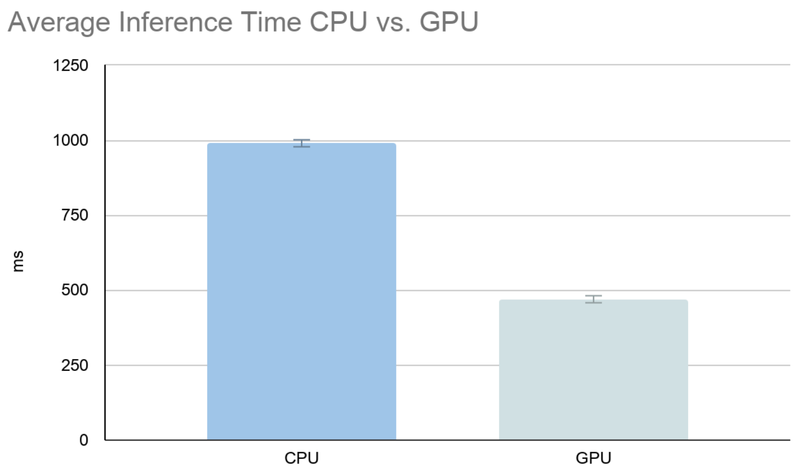
System Evaluation

The most important factors in designing the system of the application were two-fold: having low latency and using less memory. Achieving low latency for running inference on the machine learning model was important in order to ensure that the style transferred videos could be displayed in real time to the user, without the user noticing visual lags between the original video and the style transferred video. The second aspect was important to ensure that the model could be stored within the application and run fast on the edge, because we made a decision to create a mobile application for video style transfer. The following includes a detailed description of how we tried to evaluate the performance of the application's system with respect to the two objectives, as well as how we incorporated these evaluations into the design decisions we made for the application's system architecture.

First, the size of the machine learning model used during inference was very important for ensuring that the application can run predictions entirely on the device. We had to choose a model architecture that shows good performance in terms of image style transfer (since we applied style transfer to the video on a frame-by-frame basis) and had reasonable model size that could fit into a mobile device. Mobile devices typically have limited memory compared to personal computers and the number of model parameters in order to run inference on the CPUs or GPUs of the mobile device. Since the model size had to be relatively small, we picked a MobileNet architecture for the style prediction model and a simple feed forward network for the style transfer model. In addition to selecting a model architecture that is small in size, we used mixed precision models (float16 and int8) in order to fit the model into the application. The table below shows the size of the tflite record containing the model for each of the quantization levels.

	int8	float16
Style Prediction	2.8Mb	4.7Mb
Style Transfer	0.2Mb	0.4Mb

Second, in order to speed up the performance of the model during inference, we decided that the application should utilize the mobile device's GPU whenever possible. As mentioned above, the application uses the float16 model when GPU is available or otherwise uses the int8 model when inference is run using the device's CPU only. After experimenting with utilizing the GPU, we noticed more than 2x drop in the average time it takes for model inference to run on the device. The average inference time of the model when using the CPU was around 990ms. The average inference time of the model when using the GPU was around 470ms.



Lastly, we tried to speed up the inference time by precalculating all of the style vector embeddings from the reference images beforehand. This would allow us to only run inference on the style transfer model during prediction, without the need to redo the calculations for the style prediction model in real time. However, we decided to abandon this approach because of two reasons. First, precalculating the style embedding vectors didn't yield much benefits in terms of inference time. Compared to the previous approach, there was only a ~8ms drop in inference time during prediction, which was very small compared to the total inference time. Second, there was additional overhead for storing the style embedding vectors, which wasn't justified by the insignificant drop in inference time during prediction.

Application Demonstration

The application has three pages. When the user turns on the application, the first page comes up, which serves as the navigation activity for the video style transfer app. A single button appears on the navigation page: "Apply Style Transfer on Video". If the user clicks on the first button, then the user is redirected to the second page of the application, which serves as the main activity for video style transfer.

In the activity, there are six major components. First, the top view displays the style transferred video frames. Since the phone's camera sends the video at a frame rate of 24 frames per second and the model's inference time for each frame is approximately ~470ms, we see a little lag in the style transferred video; nevertheless, the user can see the video being stylized in real time. The second component is the camera view, which shows the original video being taken from the phone's camera in real time. The frames that are shown in this original camera view is used as the input image for the style transferred video in the above view. The third component is the style selection button at the bottom right, which the user can use to select different style reference images. These images are retrieved from the phone's local gallery. The fourth component is a button that lets the user choose the device (cpu or gpu) on which the inference should run. Lastly, at the center left of the phone screen, we display three information: inference time (ms), model name, and device which the model is running on (i.e. GPU or CPU).

Reflection

What worked well for our team during the quarter was using the scrum method for agile development during the entire cycle of application development. Since we weren't able to meet in person due to COVID-19, coordination of the development process was essential in order to ensure that all team members were on the same page and didn't do redundant work that was unnecessary. Even though the team size was relatively small (3 people), the project was challenging because we didn't have a separate project manager and we had to figure out the scope and technical details of the project by ourselves. In order to set the appropriate scope of the project and to decide the objectives, we met once or twice a week to create the sprint backlog, which included all the independent tasks that our team wanted to complete within the next week or so. Then, we divided up the tasks and assigned each task to one or two members, being flexible enough to make sure that those of us who had a busy schedule that week took a little number of tasks. Setting clear goals and deadlines for the MVP and the final product helped us reach the objectives we set at the beginning of the project, as well as ensuring that the functionalities of our mobile application were implemented before each milestone.

What we could have done differently was setting the right scope for the project. At first, when we were deciding the MVP of the project, we initially set a too narrow scope. We thought about creating a mobile application that runs style transfer on an image for the MVP milestone. But we later figured out that creating an application for image style transfer was a bit insignificant as the MVP of the final product, which would be a video style transfer app. After talking to the TAs, we modified our plans to at least create an app that works from end to end for video style transfer as the MVP milestone. Even though the MVP only used a pre-trained version of the style transfer ML model and had high latency, we were able to test out the concept of the final product and to figure out the bottlenecks for system performance.

If given the opportunity to use unlimited resources and time for the project, it would be interesting to add video upload and save features to the application. The application could allow users to upload already existing videos into the application, then let them download the style transferred videos back to the photo album. Also, we thought it would be interesting to integrate the video style transfer feature with a social media app like TikTok or Instagram Reels. This would add a video filter to the social media application that lets users automatically complete a video style transfer on the content of their choice. Doing so would increase the usability of our application, especially for young users who we targeted the application in the first place.

Broader Impacts

We have developed a style transfer application that runs real-time, on a frame-by-frame basis. Through our work, we expect that people would be able to come up with more novel ways of self-expression. We believe our project has its strength in that users can choose their own style image, and apply the style to not only static images but also dynamic videos.

However, we were concerned that our application remains defenseless against copyrights issues on images that are to be used as styles. We have observed that once frames are style-transferred, it is extremely difficult to distinguish which style image has been used for style transformation. Thus, we believe that it would be very easy to create style-transferred videos with copyrights issues unattended or unnoticed.

To tackle this problem, we have thought of two approaches. The first approach was to limit the choice of style images that we provide, and the second was to add a feature that classified if a style image was free to use or not. With limited time, we went on with the first option, but we hope to update our app with a novel way to filter out style images that may be problematic to use.

References

1. https://web.stanford.edu/class/cs331b/2016/projects/tam.pdf
2. https://arxiv.org/abs/1603.08155
3. https://github.com/jcjohnson/fast-neural-style
4. https://arxiv.org/pdf/1508.06576.pdf
5. https://blog.tensorflow.org/2020/04/optimizing-style-transfer-to-run-on-mobile-with-tflite.html
6. https://www.tensorflow.org/lite/examples/style_transfer/overview
7. https://www.tensorflow.org/tutorials/generative/style_transfer
8.
https://medium.com/google-cloud/on-device-machine-learning-train-and-run-tensorflow-lite-models-in-your-flutter-apps-15ea796e5ad4
9.
https://farmaker47.medium.com/android-implementation-of-video-style-transfer-with-tensorflow-lite-models-9338a6d2a3ea
10.
https://medium.com/@starling.jonah/artistic-style-transfer-with-tensorflow-lite-on-android-943af9ca28d8
11. https://github.com/googlecodelabs/tensorflow-style-transfer-android
12. https://github.com/magenta/magenta
13. https://web.eecs.umich.edu/~honglak/bmvc2017_style_transfer.pdf
14. https://arxiv.org/pdf/1705.06830v2.pdf

Contributions

Jihun Hong: Contributed to development of flutter application, assisted in model training, improved style transfer latency, debugged multiple Bitmap related issues, found and fixed multiple edge cases, improved UI of application
Jaewoo Jang: Main contributor in developing the android application with flutter, overall project organizer and team leader, got the application to work on the edge device, developed overall pipeline, debugged multiple Bitmap processing related issues
Collin Kwon: Prepared and preprocessed data, trained multiple style transfer models for better latency