Poznan University of Technology
Faculty of Computing
Institute of Computing

BACHELOR THESIS

VIDEO PROCESSING USING STYLE TRANSFER
WITH CONVOLUTIONAL NEURAL NETWORK

by
Kamil Burdziński, Filip Bończyk,
Bartosz Sobkowiak, Joanna Świda

supervisor
dr hab. inż. Wojciech Kotłowski

Poznan, January 2020

Abstract

In this thesis, we are interested in the use of convolution neural networks in video processing. We analyzed the neural style transfer concept and base our project on the work of Li et al. [24]. Our goal is to enable fast real-time video processing using artistic style transfer by speeding up the existing convolutional neural network and ensure easy access by mobile application and lightweight server. We investigate different approaches and finally use network pruning and TensorRT library in order to improve the speed of the algorithm. Focusing on availability, we design the stream server in WebRTC standard and cross-platform mobile application written in Flutter. The first chapters describe the technologies and implementation of these three components. Then, we outline the experiments with various parameters on different architectures. Our main contribution to the development of initial network architecture [24] is network and model pruning. Network pruning was proved to be very efficient for classification task. Since transforming picture to picture can be more demanding and is often ill-defined, it was not clear, whether the same compression algorithms would work just as well. We demonstrate even simple pruning methods can be used with little to none degradation of output's quality. The results, after the final adjustments, shows that our network can achieve similar results to the initial one, but with greater speed and higher image resolution. We reach the point where the application can smoothly stream transformed video, so we are pleased with results. The possible delays depend on the network connection between a mobile device and the server and might be multiplied by poor GPUs architecture. There is still a field for further quality improvements and speeding, since we focus only on some architectural aspects of network and did not optimize core server functions.

Keywords— Artistic Style Transfer, Convolutional Neural Network, Real-Time Video Processing

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The recent innovations in the field of neural networks created new possibilities in the domain of video and image processing. The neural style transfer (NST) introduced by [2] enables rendering image with its own content but a style of an another one. Various papers present improvements of this concept - enhancement of: artistic stylization effects, computational speed or content-style distribution. However, the NST approach is still fresh and further improvements could be done. We find that there still is a lack in the real-time video processing area, as well as in easy access for non-skilled users. Even the latest works we base on, provide algorithms that are not fast enough for high-resolution processing, neither for processing on average machine architectures. The deficit of availability of free and easy-to-use applications or platforms is also noticeable.

1.2 Project objective

The overall objective of our thesis is to enable fast real-time video processing using artistic style transfer by speeding up the existing convolutional neural network and ensure easy access by mobile application and lightweight server.

We aim to maintain good video quality (1024x576) while reducing the computational time by use of network pruning and TensorRT. The speedup must go hand in hand with video stability and color preservation although we do not want to depend on the sophisticated architecture. Moreover, we attempted to achieve a movie-like framerate (~24 FPS) without limiting the number of possible styles.

Access to our program should be provided for anyone interested - especially for those without programming skills necessary to launch neural network model. Therefore designing a user-friendly and cross-platform mobile application was the second goal. And since we focus on the video quality, the server should seamlessly connect application and algorithm - from any place without delays - what was the third objective.

1.3 Work structure

Since our project contains parts initially unrelated to each other, we decided to split the work into four main categories. Every team member individually focused on an owned part and then we join them together.
The work division can be summarized as follows:

1. Mobile Application - Filip Bończyk
    - Designing mobile application in Flutter
    - Client-server stream testing
    - Creating docker containers
    - Final merged project testing
2. Server - Bartosz Sobkowiak
    - Development of server in WebRTC standard
    - Preparing mechanism of frame catching and passing to the CNN
    - Server deploying
    - Client-server stream testing
3. Nerual Network - Kamil Burdziński
    - Related works research
    - Project structure design
    - Optimization of image processing algorithm
    - Network retraining
    - Network testing and accelerating
    - Binding server and algorithm together
4. Nerual Network - Joanna Świda
    - Style scaling
    - Color preservation
    - Network testing support

1.4 Repositories

Each part of the project can be found on GitHub:

- Style Transfer Algorithm and Neural Network

  https://github.com/kamieen03/style-transfer-net

  https://github.com/kamieen03/style-transfer-server

- Mobile Application

  https://github.com/bonczol/style-app

- Server

  https://github.com/bbbrtk/aiortc

- Docker script

  https://github.com/bonczol/style-docker

- Thesis

  https://github.com/bbbrtk/bachelor-thesis

2 Background

This chapter describes the background necessary for our work. At the beginning, we outline the core concepts of machine learning and neural networks. The key elements of image style transfer are briefly explained afterwards.

2.1 Machine Learning

Machine Learning (ML) is the 'field of study that gives computers the capability to learn without being explicitly programmed' [39]. Technically, machine learning is an approach to data science and data analysis that involves adapting and building special models, which allow computer programs to learn through experience. It is based on many other fields such as: linear algebra, calculus, statistics, probability or graph theory. ML models are constructed and improved by algorithms which goal is to make the most accurate possible prediction.

There are various ML concepts and architectures, but one can highlight three parts included in most common designs:

- Model - the part which makes predictions
- Parameters - the factors used by the **model** to make decisions
- Learner - the part that adjusts the **parameters**

Machine Learning works by gathering the data (where the quality of data determines the future quality of model), processing them to ensure consistence and relevance, followed by splitting them to different sets (usually: training and test sets). Next, the chosen algorithm and techniques try to build the model which is evaluated and upgraded several times to reach the appropriate level of prediction accuracy [43]. ML model needs well-prepared data to work properly. A prepared batch of data usually includes label describing its type. It enables algorithm to distinguish the batches and learn about their specific features.

One way that we can classify the tasks that machine learning algorithms solve is by the amount of labeled data and the generated feedback. In the first type, the significant amount of labeled data is provided and the model generates some output. This paradigm is called **supervised learning**. In another paradigm, no labeled data is provided, no feedback generated - it is **unsupervised learning**. Lastly, the **reinforcement learning**, rather different from the previous ones, take suitable action to maximize reward in a particular situation. The three main paradigms of ML are depicted in Figure 2.1.

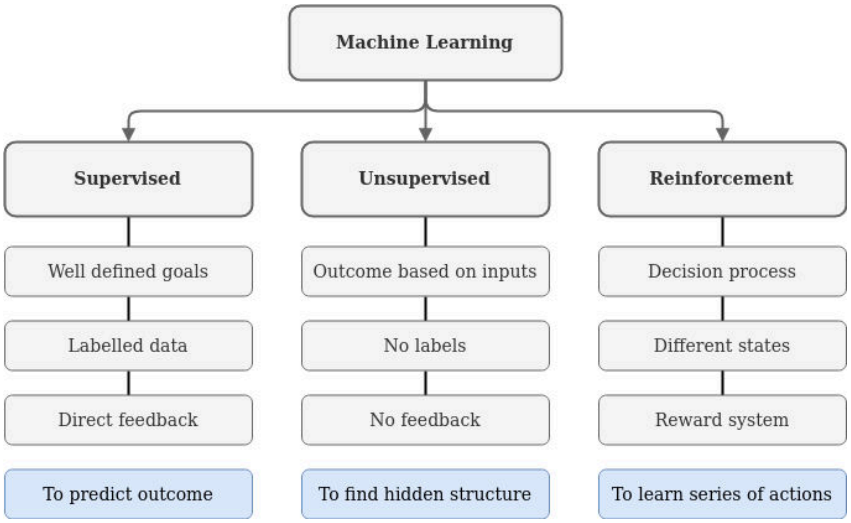

Figure 2.1: Machine Learning paradigms

There are a whole bunch of different applications to which machine learning methods can be implemented. The most popular ones [6] incl.: Natural Language Processing, Classification Problems, Learning Associations, Pattern Recognition and Image Processing which is the one we focus on in this paper.

2.2 Neural Networks

2.2.1 Overview

The neural network is a concept and technique for building a computer program that learns from data and which is based very loosely on how the human brain works. More precisely, it is a collection of artificial **neurons** which are connected and allowed to send messages to each other. When this collection is asked to solve a problem, it attempts to do so over and over, each time strengthening the connections that lead to successful solution and reduce those leading to the wrong one [9].

Technically, neural networks are **weighted graphs**. They consist of an ordered set of layers, where every layer is a group of neurons (nodes). The first layer of the neural network is called the **input layer**, and the last one is called the **output layer**. The layers in between are named **hidden layers**. Nodes belonging to one layer are connected to the nodes in the following and/or the previous layers. These connections are weighted edges.

Given an input, each node produces an output (a number) which is calculated by the **activation**

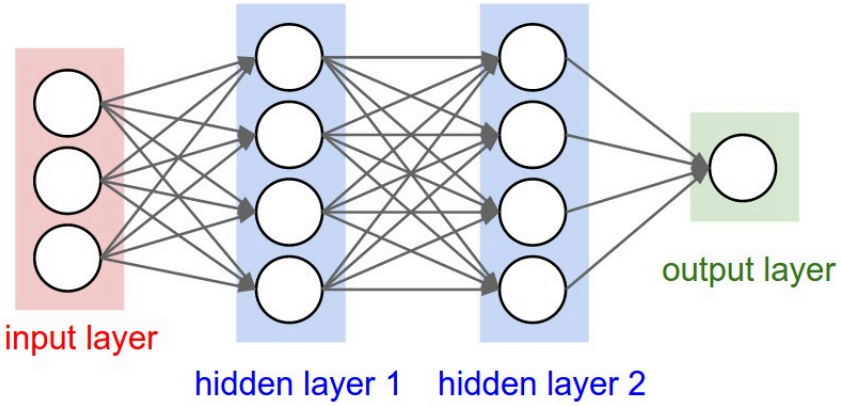

Figure 2.2: Neural network layers [42]

function. This output is then passed to the subsequently connected nodes, which calculate the next outputs. By calculating layer outputs consecutively, we calculate the output of the final layer, which is the one we need.

2.2.2 Deep Neural Networks

Deep neural network is simply a feedforward network with many hidden layers. There is no definite answer to the question of how many layers does a network need to be qualified as deep, but usually having two or more hidden layers counts as deep. In contrast, a network with only a single hidden layer is conventionally called "shallow" [5].

2.2.3 Relationships

The variety of terms in the field of AI/ML may be confusing. Especially, the relations between major concepts described in this paper. Diagram presented on Figure 2.3 tries to clarify it.

2.3 Convolutional Neural Networks

2.3.1 Definition

Convolution Neural Networks (later: CNN or Conv) are specialized types of networks. The **convolution** itself is a specific kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers [5]. CNN is well adapted for the processing of images, but the same concept also fits other fields, like audio or video. Compared to the 'standard' feed-forward neural network with a similarly-sized layer, CNN could have fewer connections and parameters. Therefore it is easier to train, while its theoretically-best performance is going to be only slightly worse [21].

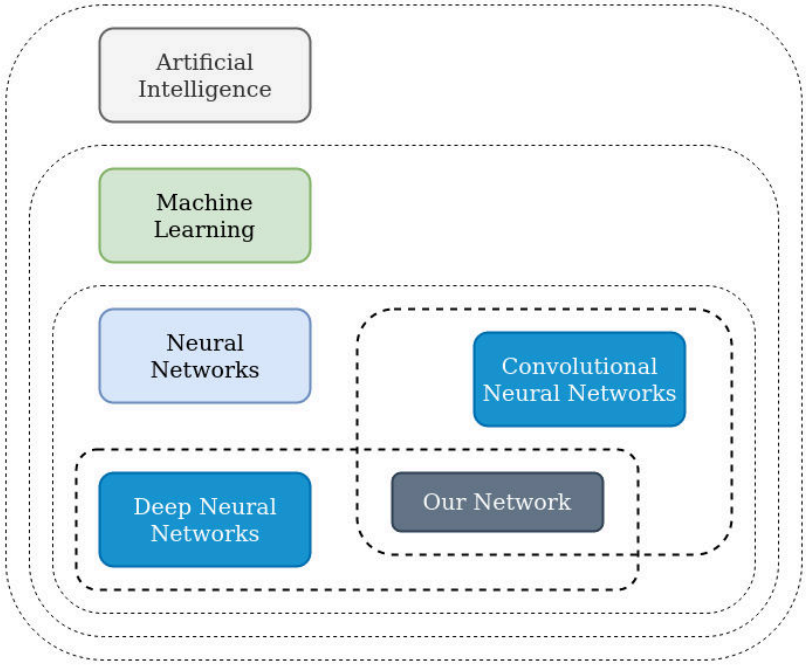

Figure 2.3: The relationship between main fields

2.3.2 CNN Layers

The architecture of a CNN is designed to take advantage of the two-dimensional structure of an input image. Convolutional network is comprised of one or more convolutional layers followed often by subsampling step (pooling) and then by one or more fully connected layers - just as in a standard multi-layer network [42]. The most common CNN layers listed below are normally placed where the hidden layers are depicted on Figure 2.2.

- The **convolution layer** has to extract features from the input. It computes a dot product between neuron weights and a small region it is connected to in the input volume.

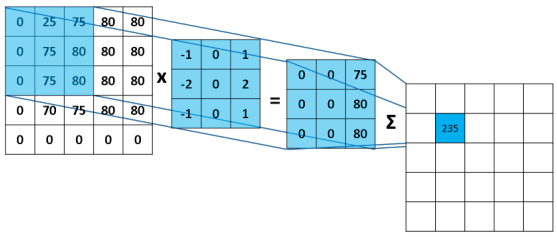

Figure 2.4: Convolution with 3x3 filter [42]

- The **pooling layer** reduces the spatial size of these feature maps. Sometimes, the input image is a big one (and therefore time-consuming, especially if you have a large input set) or there are sparse data. In these cases, the objective of the Pooling Layers is to reduce

the spatial dimension of the input matrix. A pooling layer is applied after one or multiple convolution layers.

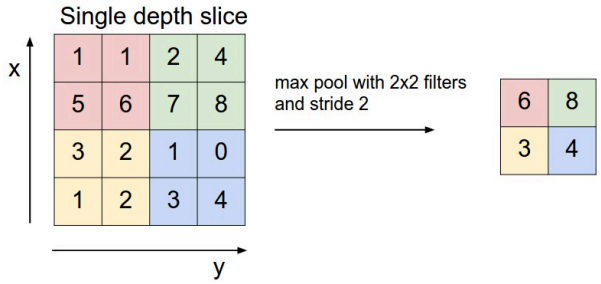

Figure 2.5: Process of max-pooling [42]

- The last one - **fully connected layer** - performs the high-level reasoning and produces the final output. Neurons in a fully connected layer have connections to **all** neurons in the previous layer.

2.3.3 Activation

The activation function is attached to each neuron in the network, and determines whether it should be activated or not. Based on whether each neuron input is relevant for the model prediction. Activation functions also helps to normalize the output of each neuron to some range (generally between 1 and 0 or 1 and -1) [28].

Since activation function is calculated across thousands or even millions of neurons for each data sample, it must be computationally efficient. One of the the most used one (also in our network) is **Rectified Linear Unit (ReLU)**. It applies an element-wise function, such as the max(0,x) - thresholding at zero - to either activate or deactivate each single neuron.

2.3.4 Loss

To get know if the result of operations performed in a neural network is correct (or not), we need a function that allows solutions to be ranked and compared. This is known as the **loss function**. It reduces all the various good and bad aspects of a possibly complex system down to a single number (or vector of numbers), a scalar value, which evaluates the candidate solution [37].

2.3.5 Backpropagation

Backpropagation is an algorithm commonly used to train neural networks. When the neural network is initialized, weights are set randomly (or para-randomly). The input is loaded and then passed through the network, and it provides an output for each one neuron, given the

initial weights. Backpropagation helps to adjust the weights of the neurons so that the result comes closer and closer to the known true result. It is worth to mention that backprop for a convolution operation (for both the data and the weights) is also a convolution. The description of this mechanism in our network can be found in chapter 3.

2.4 Computer Vision

Computer vision is a field of computer science that tries to understand the images and videos - how they are stored, how can we manipulate and how effectively retrieve data from them. It seeks to replicate the human vision system and enable computers to *think and see* the same way we do. Computer Vision plays a major role in photo correction apps and art generation but also in self-driving vehicles or robotics [27].

Thanks to the growing popularity of AI and recent innovations in the field of Deep Learning (and Neural Networks as well), Computer Vision is developing rapidly and is able to achieve better rates in almost all areas. In many tasks, it even can surpass humans (e.g. in object detection and labeling). Since we generate and collect more data used to train and make Computer Vision better, the growth of this field will be ensured.

2.5 Style transfer

2.5.1 Overview

Style transfer is a technique of extraction visual aspects from an image and blending it together with an another image content. It aims to manipulate digital pictures (or videos) and adopt the appearance or visual style. This is usually implemented by optimizing the output image to match the content statistics of the content image and the style statistics of the style reference image. These statistics are extracted from the images using a convolutional network.

The crucial part of style transfer is process of isolating the style and the content from an image. Researchers developed many approaches which enable computers to do it with high precision. We briefly introduce some techniques in next subsections.

2.5.2 Example

One of the most common use of style transfer is creating an artificial artwork from photo. Especially the ones produced from some famous paintings. Attached examples give an idea of this approach capabilities.

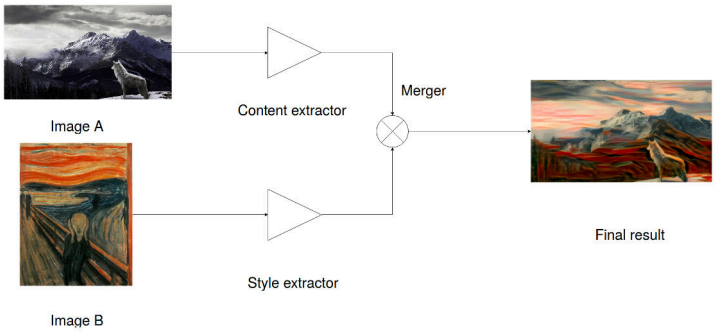

Figure 2.6: Style and content flow diagram from [44]

Figure 2.7: Picture transformed to painting-like image by [2]

2.5.3 State of the art

Neural Style Transfer (NST) was first published in the paper "A Neural Algorithm of Artistic Style" by Gatys et al. [2] and is developing rapidly gaining more and more attention. The core branches of Neural Style Transfer are presented on the Figure 2.8. Since presenting every concept of NST is not the goal of this paper, we recommend to visit this actively updated repository, where one can find information about each sub-field of the Neural Style Transfer.

2.5.4 Loss function in style transfer

In general **loss function** in NST contains two sub-losses: the content reconstruction loss and the style reconstruction loss. The idea is pretty straightforward. The first loss ensures that the activations of the higher layers are similar between the content image and the generated image. The second one makes sure that the correlation of activations in all the layers are similar between the style image and the generated image. So the differences between source and output image

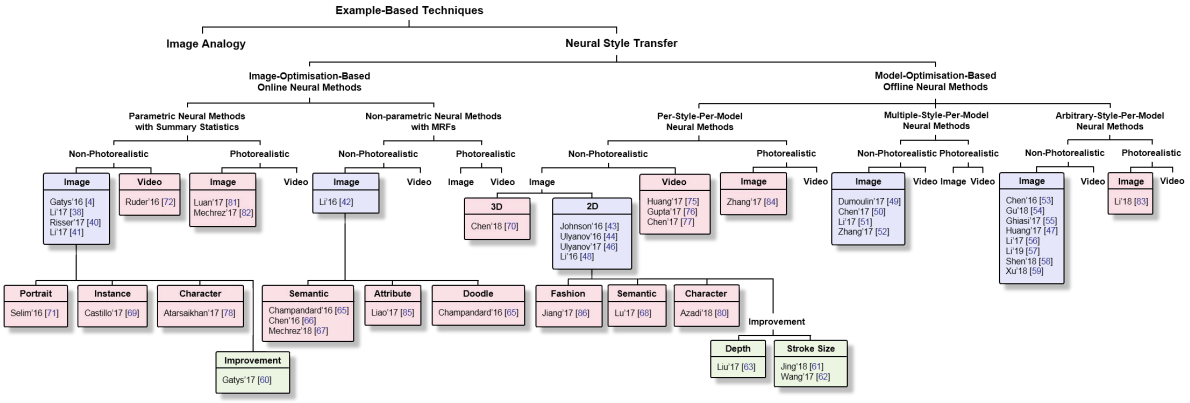

Figure 2.8: State of the art by [19]

are the basis of the loss computing. The concept can be denoted as follows:

$$L = \alpha L_{content} + \beta L_{style}$$

where α and β are hyperparameters. By controlling them one can decide how much style of source image will be preserved (and similarly - the amount of the source image content).

[2] proved that CNN captures the content info in higher levels of the network. Whereas [46] states that if two images have the same content, the activations in their higher levels are similar. It is the core idea of many content loss calculation methods. Style reconstruction loss is usually more advanced topic, but plenty of approaches base on correlation between feature maps. It is also common to compute style losses at multiple levels (while content loss is computed only once).

2.5.5 AdaIn

Among many approaches to the (Artistic) Neural Style Transfer we would like to highlight the AdaIN method [17]. The authors of the Adaptive Instance Normalization aimed to speed up the computationally efficient style transform process. They pointed out that even the best methods cost a lot - either time or computing resources. Therefore are not designed for commercial use, particularly in real-time processing programs. Moreover, their method is free from restriction of pre-defined set of styles. It seemed to achieve all objectives of our project. In a nutshell, the approach uses the **instance normalization (IN)** - the alternative to the batch normalization, which computes the mean and standard deviation, and then normalize across each channel in each training example separately. The authors designed extension to IN which 'adjusts the mean and variance of the content input to match those of the style input'. It enables fast processing and full user control of the image styling. We briefly described this method in section 3.1.3.

Despite the numerous advantages, AdaIN has some cons which are also outlined in section 3.1.3 of this paper. Nevertheless, we find AdaIN idea interesting and out-of-the-box. It might be that future work and experiments will enhance the possibilities and meet our expectations.

2.5.6 Commercial use

Some research projects became a base of mobile applications or web-services which can be used by everyone for any purpose:

- DeepArt - free website where one can turn photo into artistic image
- Neural Styler - NeuralStyler Artificial Intelligence converting videos into art works by using styles of famous artists
- Ostagram - web-service for algorithm combining the content of one image with the style of another image using CNN
- Prisma - photo-editing application able to create artwork from image

2.5.7 Image Analogy - the obsolete approach

The alternative CNN approach which used to be popular beforehand is called **image analogy**. In brief, the algorithm wants to find an analogous image B' that relates to B in the same way as A' relates to A [13]. To transfer the style of image we need a dataset of training pairs of photo and an artwork from processed photo. The main weakness of this method is that mentioned pair barely exist in practice. An artwork is seldom based on a particular photo.

2.6 Neural network compression

Latest neural networks usually have between 2 million and 50 million parameters. Older architectures are even heavier - full VGG16 model has as much as 138 million parameters. In consequence models are often too big for deployment on memory bounded mobile and embedded devices. Number of parameters strongly correlates with inference time, which in turn prevents real-time inference not only on embedded and mobile devices but even on middle-class GPUs. Overcoming these limitations is very active area of research.

Specialized architectures such as MobileNet [16, 40, 15] and ShuffleNet [47, 26], replace full convolutions with bottlenecks of lighweight pointwise, depthwise and group convolutions in order to reduce number of arithmetic operations and parameters. MobileNetV3 [15] uses Neural Architecture Search to optimize the network for mobile phone CPU inference, while discussion

in [26] shows what aspects should be considered, when designing mobile architecture manually. **Network pruning** aims to reduce already trained model's size by pruning away weights with the least impact on network's quality. [49] show that for some models even up to 87.5% weights can be removed with only marginally reduced performance. Because networks are initialized randomly, the least important parameters are usually spread across whole network. Naive pruning then results in sparse networks. Their storage is significantly reduced, however due to the fact that available linear algebra libraries are optimized for dense structures, their inference time does not scale as well. To overcome this, more structured methods of pruning were developed, among them filter pruning [23, 29] and channel pruning [12]. Weight or weights set importance can be measured by various heuristics. [23] prune away the filters with the smallest ℓ_1 norm. This approach easily generalizes to ℓ_p norm. [35] choose to prune away the filters with the smallest activation statistics. More direct methods formulate pruning try to preserve performance metrics (e.g. low loss, high accuracy) on training set. [29] formulate pruning as optimization problem where the goal is preserving original training set loss value, that is minimizing $|L_{new} - L_{orig}|$. Curiously changing the goal to $L_{new} - L_{orig}$, that is letting the loss drop, degrades the resulting network's performance.

Smaller networks can also be trained with aid of larger ones through process called **knowledge distillation** [14]. The key idea is that activations of certain layers contain knowledge which is not present in dataset labels. For example in a network trained for classification, the final softmax layer outputs are usually interpreted as probabilities of respective classes. These probabilities might indicate that for given input x, C_1 is the proper class and class C_2 is twice as probable as class C_3. If the network is well trained this means x is more similar to objects of class C_2, rather than C_3. By training a new small network to match its both softmax outputs with the larger trained network softmax outputs and proper labels, we effectively gain additional training data and ease the training. Almost every network inference time can also be increased by **quantization**. In many deep learning frameworks, computation is by default performed on 32-bit floating point numbers. Such precision is often required during training to ensure gradient numerical stability. However given that network weights have relatively small magnitude, in many applications it is unnecessary during inference. Modern GPUs enable inference in FP16 and INT8 modes. There has also been worked on even lower precision inference, performed on CPUs.

3 Network Architecture

3.1 Neural Network

In this chapter we describe the neural network used for style transfer. First we describe it's architecture and specific layers. Then we detail pruning procedure used to obtain the final model.

3.1.1 Overall network architecture

We follow architecture described in [24], changing its components details in order to enable real time inference. The network consists of two main components - encoder-decoder module and transformation module pictured in Figure 3.1. To perform style transfer, the encoder is fed with the content image and the style image, producing feature maps cF and sF respectively. Next, transformation module transfers sF statistics onto cF. Resulting feature map is passed through decoder, which creates final image. Thanks to this modular architecture, each style image needs to be encoded only once. For both encoder-decoder and transformation module we use pretrained models publicly available at https://github.com/sunshineatnoon/LinearStyleTransfer as a starting point for further development. Because of implementation details of pruning framework we use (each layer can only be used once during feed-forward phase), one encoder was not enough to encode both content and style image. As a consequence we had to include not one, but two copies of encoder in our model - one for content images and one for style images. Both encoders start of with entirely identical set of weights, but progressively diverge from each other during pruning procedure.

3.1.2 Encoder-decoder

Encoders and decoder are based on first layers of VGG-like [41] network. VGG is a family of fully convolutional neural networks developed specifically for computer vision task. Full VGG network consists of stacked Conv layers with ReLU activation function, interleaved with pooling layers performing downsampling, ending with couple of fully connected layers and Softmax layer. Such architecture allows effiecent image classification. To adapt it for other computer vision tasks like segmentation, object detection, tracking, domain adaptation or style transfer, the fully-connected part of network needs to be removed. Depending on task complexity, desired latency and available hardware, VGG-like networks of varying depth can be constructed by simply removing consequent top layers of the network. The architecture used by [24] and our pruned version are outlined in table 3.1.

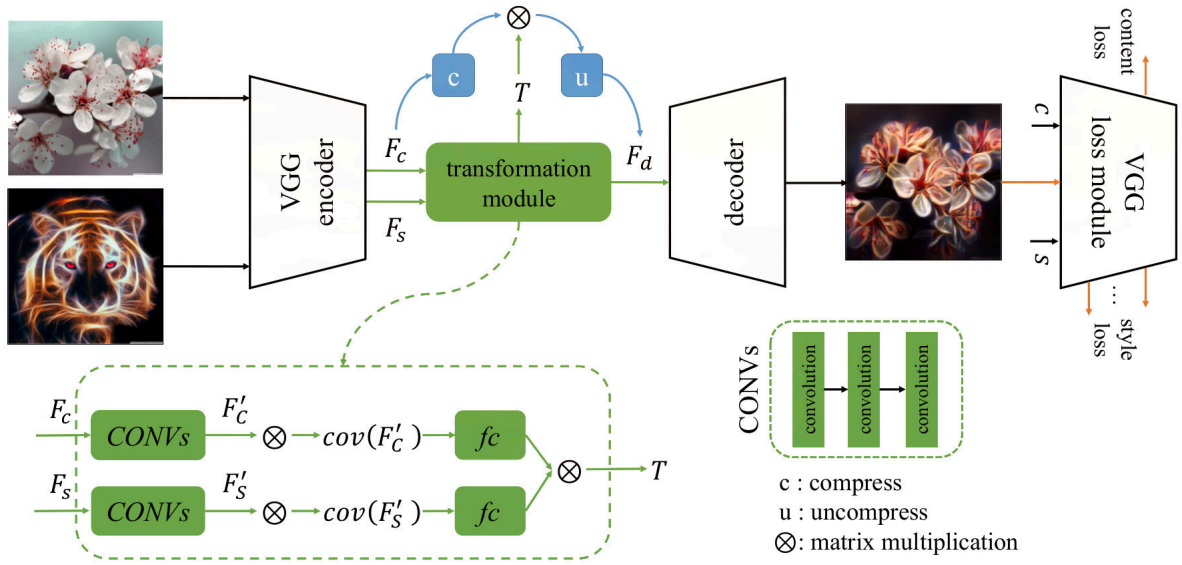

Figure 3.1: Overview of the network architecture [24]

They differ in three aspects:

- **Padding** - [24] use reflection padding. In many models used for style transfer, use of zero padding degrades network performance around the input edges [20], exhibiting as vanishing of style or artifacts (see Figure 3.2). In such situation adding reflection padding fixes the issue. We replace reflection padding with zero padding, because the former one is not implemented within TensorRT used for additional inference speedup (see chapter 4.1.3). Fortunately described problem is not nearly as visible in our model.

Figure 3.2: Result of zero padding (left) and reflection padding (right) in [20]. Style visibly diminishes around the edges in the left image.

- **Width** of Conv layers. In order to reduce inference time, we prune encoder, decoder and part of transformation module to only contain 25% of feature maps from the original architecture. This provides fivefold speedup on NVIDIA GTX 960 GPU the inference server is ran on (see table 6.1 on page 41 for details).

Li et al. [24]	Ours
Pointwise Conv(3, 3)	
CReLU(3, 64)	CReLU(3, 16)
CReLU(64, 64)	CReLU(16, 16)
Max Pool ×2	
CReLU(64, 128)	CReLU(16, 32)
CReLU(128, 128)	CReLU(32, 32)
Max Pool ×2	
CReLU(128, 256)	CReLU(32, 64)
CReLU(256, 128)	CReLU(64, 32)
Upsample Nearest ×2	
CReLU(128, 128)	CReLU(32, 32)
CReLU(128, 64)	CReLU(32, 16)
Upsample Nearest ×2	
CReLU(64, 64)	CReLU(16, 16)
CReLU(64,3)	CReLU(16, 3)

Table 3.1: Encoder-decoder architecture before and after filter pruning. The first Conv layer is pointwise convolution with 1×1 kernel, unitary stride, without padding. It has 3 input and 3 output channels. *CReLU(a,b)* is Conv layer with a input channels and b output channels followed by ReLU activation. All of *CReLU*s have 3×3 kernels, unitary stride and and single pixel of padding on each of 4 sides. Downsampling and upsampling is done with 2×2 kernels and stride 2.

- **Encoders**. As described before, our network consists of two encoders, specialized for content and style images respectively. In section 6.5 we conduct experiments to observe effects of using style encoder for content images and vice-versa.

3.1.3 Transformation module

Transformation module is the component where style is actually transfered by matching feature maps' statistics. [17] introduced statistics matching as style transfer method in form of Adaptive Instance Normalization (AdaIN). Their model also consists of encoder, transformation module and decoder. Transformation module performs AdaIn on feature maps produced by encoder by explicitly matching their means and standard deviations:

$$AdaIN(F_C, F_S) = \sigma(F_S) \left(\frac{F_C - \mu(F_C)}{\sigma(F_C)} \right) + \mu(F_S)$$

where F_C is nth content feature map and F_S is nth style feature map. Transformed feature maps are then decoded to final image. This simple transformation module performs reasonably well for smooth styles (e.g. impressionist paintings), but often fails if content or style image contains some sharp edges (e.g. cubist paintings). [24] observe AdaIN is in fact an affine transformation of feature map vector F_C. This suggests it can be generalized to higher dimensional transformation

using matrix multiplication and bias vector. It's then assumed style transfer can be performed by an affine transformation conditioned on parameters of style and content image. Using not only mean and variance, but also covariances between individual feature maps enables transferring more sophisticated features. Exact architecture of transformation module proposed by [24] is depicted in Figure 3.3.

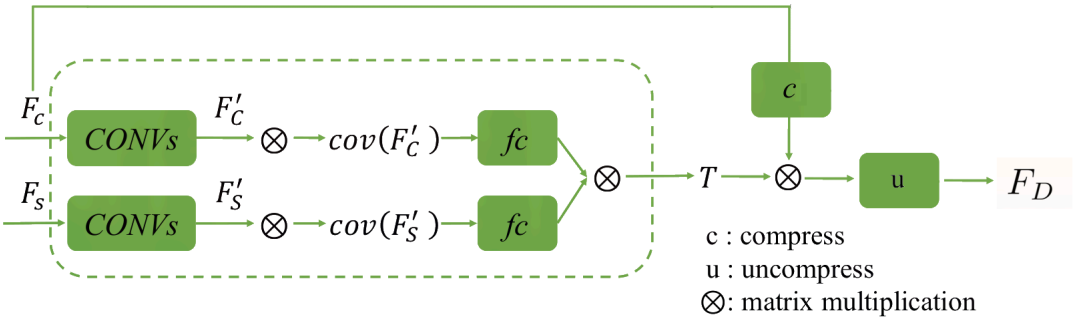

Figure 3.3: Architecture of transformation module. F_C, F_S and F_D are content's, style's and transformed image's feature maps respectively. f_c is fully connected layer and T is transformation matrix. The submodule in dashed green region is supposed to compute transformation matrix conditioned on style and content images. Figure adapted from [24].

First both F_C and F_S are centered around 0 channel-wise, that is their respective means for each channel are subtracted. They are then passed through short series of separate convolutional layers with ReLU non-linearity. It is important that they are separate, since we care about different features in content and style image. Next, they are matrix-multiplied with their respective transposes which results in their uncentered covariance matrices. Each of these matrices is then fed through fully connected layer, which is supposed to select which covariances are the most important. Results of F_C layers are then matrix-multiplied with each other, which gives us the final style transformation matrix T. On parallel path F_C vector is compressed to the size of T. Compressed F_C is matrix-multiplied with T (this the moment style is transferred) and uncompressed to the original size of F_C. Lastly mean vector of original F_S is added. This way means of F_D channels and F_S channels are aligned exactly.

3.2 Pruning

In this chapter we describe the pruning algorithm and the pruning procedure used to obtain our final network.

3.2.1 Filter pruning

Filter pruning is a type of structured pruning in which the smallest unit considered by an algorithm is convolutional filter. Convolutional layer transforms feature maps $x_i \in \mathbb{R}^{n_i \times h_i \times w_i}$ into feature maps $x_{i+1} \in \mathbb{R}^{n_{i+1} h_{i+1} \times w_{i+1}}$ by applying n_{i+1} 3D $n_i \times k \times k$ filters $F_{i,j}$. Based on some criterion, filter pruning algorithm removes selected filters to reduce required storage and inference time. If filter $F_{i,j}$ is pruned, then so is jth feature map of ith layer. Filters of the next layer which were convoluted with this feature map are no longer needed and can also be removed. The criterion by which our algorithm selects filters is ℓ_1 norm, that is $\sum_{i,j,k} |F_{i,j,k}|$ where summation is done over all of filter's values, suggested by [23]. Intuition behind this and similar criteria (e.g. using ℓ_p norm) is that the smaller filter's norm, the smaller it's influence on overall network output. Despite it's simplicity this criterion performs well in practice and was adopted by many recent publications [1, 48].

Our algorithm takes as inputs layers which should be pruned and sparsity levels respective layers should be pruned to. Pruning changes the network and it's activations' distribution very suddenly, so after every removal of filters, network has to be retrained to at least partially regain it's original quality. Pruning can be done at once (so called one shot pruning), where desired sparsity level is achieved just after one epoch or be spread across multiple epochs. Retraining is easier in latter scenario, as network isn't altered suddenly so significantly. Specifically we use AGP (Automated Gradual Pruner) sparsity schedule proposed by [49]. It prunes filters rapidly at the beginning, when redundant weights are abundant and slows down gradually. % of filters remaining (called density) after epoch k is given by

$$D_k = f_f + (100\% - f_f) * (1 - \tfrac{k}{K})^3$$

where f_f is final density and K is total number of epochs pruning is performed for. In our case $f_f = 25\%$ and $K = 10$. Exact densities in our schedule our shown in Figure 3.4.

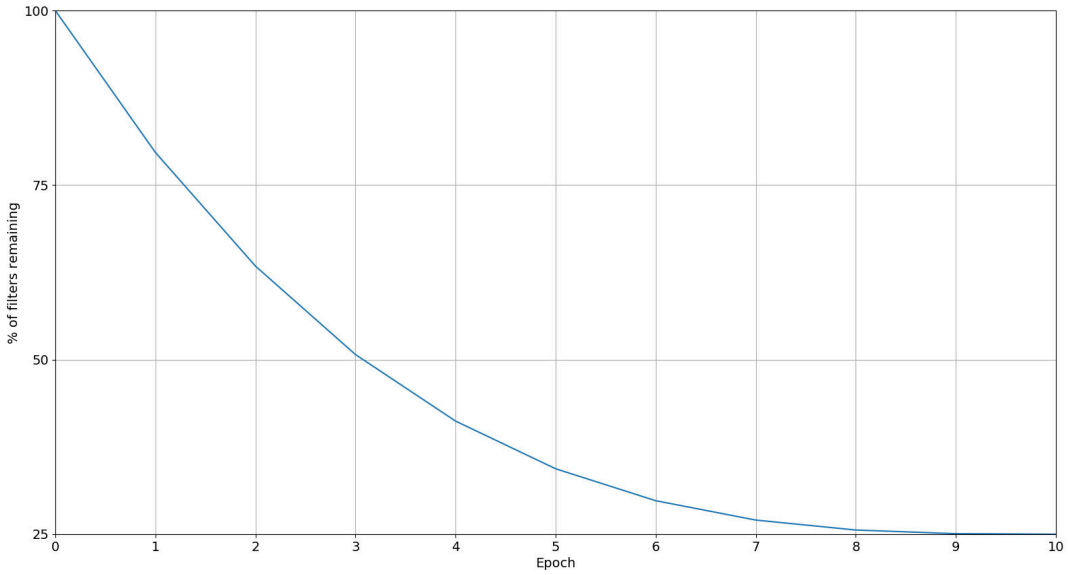

Figure 3.4: Density of pruned layers after each epoch.

3.2.2 Loss function

As can be seen in Figure 3.1, losses are computed by special VGG19 loss module. The network was pretrained on large dataset for classification task and is able to extract useful features for any image. VGG19 is a deep network and it's consecutive layers encode different characteristics of image, for example first layers might encode colors and texture, while latter layers are able to encode more abstract features like shapes. To compute losses, transformed image, content image and style image are passed through loss module, which encodes them into $F_{t,i}$, $F_{c,i}$ and $F_{s,i}$ feature maps respectively, where $F_{x,n}$ is feature map computed by nth ReLU activation layer. [24] find computing style loss combining many feature maps $F_{s,i}$ from various depths of network yields the best visual effects - they pick $i = 2, 4, 6, 10$. For content loss one feature map is enough - $i = 10$. Content loss is then computed as MSE between $F_{c,10}$ and $F_{t,10}$. Style loss is given by

$$\sum_{i \in I} MSE(mean(F_{t,i}), mean(F_{s,i})) + MSE(F_{t,i}F_{t,i}^T, F_{s,i}F_{s,i}^T)$$

where I is index set $\{2, 4, 6, 10\}$ and MSE is mean squared error. $F_{x,n}$ are $2D$ matrices of shape $(C, H \times W)$ reshaped from $3D$ feature maps. Mean is then computed with respect to second dimension, resulting in C-dimensional vector. AA^T is called Gram matrix of matrix A. It serves as A's uncentered covariance. Style loss' purpose is then matching statistics of style images' distribution and transformed images' distribution. Total loss is weighted sum of content loss and

style loss:

$$L = L_{content} + \lambda L_{style}$$

[24] use $\lambda = 0.02$ and we also find it results in the most appealing visual effects. During pruning and fine-tuning we use SGD optimizer to apply gradients. We set learning rate to 10^{-4} and momentum to 0.9. We use SGD mainly because other optimizers aren't implemented in pruning framework we use. However we also find adaptive optimizers result in instable training of transformation module and gradient explosion.

3.2.3 Pruning schedule, datasets

Following [24] we adopt MS-COCO [25] as content dataset and Wikiart [30] as style dataset. Both of these datasets contain roughly 80000 images. We divide each of them into 73000 training images and 7000 validation images. Implementing pruning on your own requires deep knowledge of used deep learning framework. Luckily there are model minimization libraries for both PyTorch and Tensorflow. While Tensorflow has a builtin library aiding pruning and quantization, PyTorch doesn't have one yet - quantization is in experimental phase and pruning is not available at all. The best software for pruning PyTorch models we found is Distiller [50]. It is an open source library which implements many structured and unstructured pruning algorithms, enables testing quality of quantized models and performing knowledge distillation. It's still far from perfect (current version is 0.4.1), but thanks to an open source code, it's easily extendable. Described above ℓ_1 AGP filter pruning is already implemented in Distiller. After specifying our network's architecture and datasets, and fixing couple minor bugs we used it to prune and then fine-tune the original model shared by [24] available at https://github.com/sunshineatnoon/LinearStyleTransfer. Pruning phase takes 10 epochs. At the beginning of each epoch scheduled part of filters with the lowest ℓ_1 norm is masked with zeros and frozen for the rest of procedure (computation and application of gradients for them is turned off). For the rest of epoch usual training is performed - both content and style datasets are shuffled, and network is optimized with consequent mini-batches. During this training, network should at least partially recover from performance drop caused by sudden change of weights. After 10 epochs, specified earlier levels of density are achieved and frozen weights are entirely removed from the model - at this moment architecture of the network is altered and physical storage required to save the model is reduced. Then comes the fine-tuning phase - network is trained for another 10 epochs, which should be enough for model to converge to some satisfying state.

Throughout both phases validation loss is measured after each training epoch. As final model

we pick the one with the lowest validation loss during fine-tuning, which happens to be the one after epoch 18. Training and validation losses are pictured in Figure 3.5. Because image reconstruction is fairly easy, content losses remain almost constant during whole process. We see style loss is much more heavily influenced by pruning. Right at the start, mean style loss on training set is right below 2.5, while mean validation loss is over 11. This means model is able to quickly recover and fit to the data even after very aggresive pruning (in AGP scheme first epoch removes the most filters), but regaining generalization requires more time. While training style loss drops almost uniformly starting from the 5th epoch, when pruning becomes lighter, the validation set style loss only stabilizes after 10th epoch - when pruning ends. It's important to note that even though validation style loss doesn't change much after 11th epoch, model is still learning and after 18th epoch network performs much better than after 11th. This is because the style loss computed by loss module is only a proxy for style difference, which doesn't have any clear mathematical definition. In result value of style loss only partially reflects difference in style between two pictures.

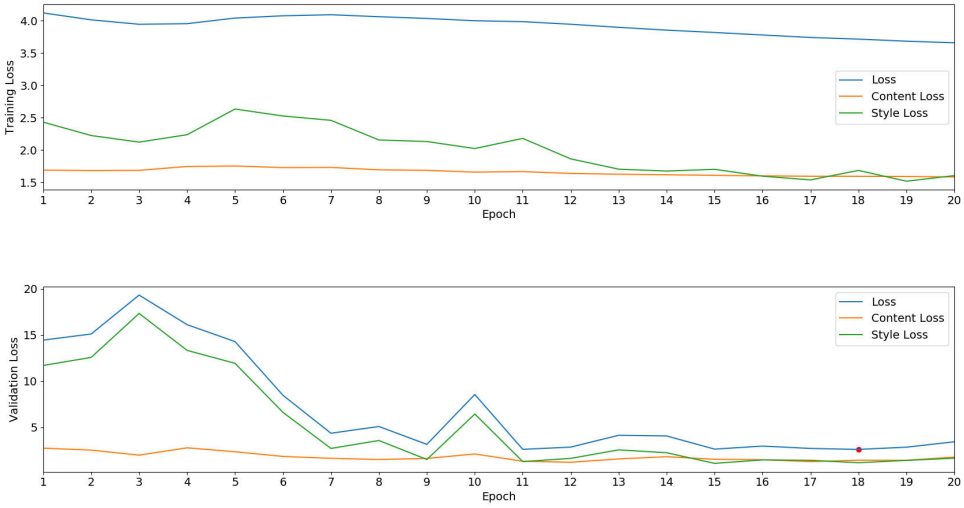

Figure 3.5: Mean training and validation losses throughout pruning and fine-tuning procedure. Optimal model with minimal validation loss in fine-tuning phase is marked with red dot.

4 Technologies

Today we have a great variety of free open-source tools for NN purposes and its number is increasing with the growth of AI and ML. Development of neural networks, mobile application or server is full of nuances - an inexperienced programmer may encounter many troubles since not everything can be easily tracked. However, most problems can be resolved by good Internet research. Therefore, it is highly significant to choose the up-to-date tools with well-prepared documentation and active communities. Technologies we have chosen allowed us to focus on core concepts instead of struggling with implementation or configuration.

4.1 Neural Network

4.1.1 PyTorch

PyTorch is an open-source machine learning library created by Facebook and broadly used in the field of computer vision. Released in 2016 [36] rapidly gained popularity among researchers and developers. PyTorch is not an another Python binding into a C++ framework. It uses core Python concepts like structures, classes or conditions, so integrates easily with the rest of Python ecosystem. Library provides all necessary tools for developing and training neural networks based on deep learning models.

Two main features of this package are:

- Strong **GPU acceleration** during computation of tensors
- Neural networks built on a tape-based **autograd system**

We decided to pick out PyTorch instead of TensorFlow (the main and, actually, the only alternative) due to several reasons. Firstly, TensorFlow was becoming more and more complex for machine learning beginners, where the PyTorch is based on basic and familiar programming paradigms. The new version of TF (2.0) solves this problem but was released before we had started working on the project. Moreover, the documentation is also pretty straightforward and helpful for newcomers. And finally, plenty of style transfer approaches are built on PyTorch. Including the network we adopt.

4.1.2 CUDA

CUDA is a parallel computing platform and programming model created by NVIDIA [31]. It enables training models on GPU with a shorter period of time and allows high floating-point

computing performance. CUDA allows using equivalent CPU functions for maximum use of GPUs and accelerated programs. Many frameworks like TensorFlow or PyTorch relay on CUDA for their GPU support. CUDA supports all NVIDIA graphic cards. The NVIDIA company also created CUDA Toolkit, which includes libraries, debugging and optimization tools. There are three main libraries intended for deep learning: CuDNN, which is the most popular for deep neutral network computations and mostly used in open-source frameworks; TensorRT - the high-performance inference optimizer and runtime; DeepStream, a library to video inference. One can use also use signal processing and math libraries and in case, when there is no needed library - can write own low-level CUDA program in C or C++.

4.1.3 TensorRT

TensorRT is an SDK for deep learning interface **created by NVIDIA and designed for their GPUs**. It is built on CUDA and provides C++ and Python APIs, which allows loading pre-trained deep learning models, (trained with libraries like PyTorch) and run them on NVIDIA GPU. The main advantages of TensorRT are high-performance deep learning interface optimizer and light-weight runtime engine, that provide low latency and minimize streaming video delays [32]. The engine is generated only once for each GPU and serialized as a plan file for the next uses. It is also possible to run an application with lower precision (if hardware allows doing so) to reduce memory requirements and reduce inference time.

The workflow of the build phase network on TensorRT follows those steps:

- Eliminate layers with unused outputs
- Eliminate operation equivalent to no-operation
- Fusion of convolution
- Aggregate operations with similar parameters on the same source tensor
- Merge of concatenation layers
- Modify the precision of weights
- Calibrate and appropriate scaling factors
- Run layers on dummy data - to chose the fastest kernel catalog

TensorRT allows focusing on creating new applications with AI, autonomous machines and high-performance computing instead of calibration for inference development. optimization TensorRT chooses specific CUDA kernels for the platform to maximize performance on GPUs. It

uses Docker for integration with DevOps architectures.

TensorRT doesn't support the direct building of engines from PyTorch models, so we use ONNX format as an intermediate step in the build process. Specifically, after pruning we convert encoders, transformation modules, and decoder to ONNX. ONNX tends to produce excessively complex computational graphs, which sometimes prevents from conversion to TensorRT, so next, we use open-source ONNX simplifier [3] to make models more compact. Finally, using open-source onnx2trt parser [4] we build TensorRT engines based on simplified ONNX models.

4.1.4 OpenCV

Open Source Computer Vision Library is an open-source library, originally developed by Intel, for computer vision and machine learning software [33]. It includes dozens of computer vision algorithms adjusted for easy use for detect and recognize faces, identify objects and many others designed for image processing. Although written in C++, OpenCV has Python bindings and provides support for Machine Learning libraries like PyTorch. By using it, one can process images and videos to detect objects, apply filters, analyze structure or perform reconstruction. OpenCV is designed to handle real-time vision applications and is currently developing full-featured CUDA interfaces.

4.2 Distiller

Distiller[50] is an open-source Python package for neural network compression. It provides a PyTorch environment for prototyping and analyzing compression algorithms, such as sparsity-inducing methods and low precision arithmetic. Distiller is still in experimental phase and is supposed to provide examples and tools to build your own pruning pipelines, rather than complex library of functions that would work of the box. As such it contains:

- A framework for integrating pruning, regularization and quantization algorithms
- A set of tools for analyzing and evaluating compression performance
- Example implementations of state-of-the-art compression algorithms
- Experimental implementations of knowledge distillation and conditional computation
- Convenient YAML format for describing compression schedule

We were particularly interested in available pruning algorithms. Out of many implemented methods only couple of them worked (that is threw easily fixable bugs). Considering results in

[23] and comparisons in other papers we picked L1 structured pruning algorithm as a simple and reliable method.

4.3 Server

Client-server communication is divided into two parts: video streaming and transferring filter specification data. The first part is maintained by a server in WebRTC standard, whereas the second one is micro-server based on Flask. Both segments are written in Python and based on projects developed by open-source communities.

4.3.1 WebRTC

WebRTC is an open framework that enables **Real Time Communication (RTC)** in web browsers and mobile apps via simple API [10]. It provides tools to create peer-to-peer communication without any external plugins or programs. Standardized on API level by W3C and protocol level by IETF is widely used and supported by Google, Apple, Microsoft, and main browser vendors. Given that our project and mobile application is cross-platfoom, we consider it as a key factor when choosing server technology.

WebRTC architecture presented on Figure 4.1 contains two main layers:

- C++ API - for browser makers implementing the Web API proposal
- Javascript Web API - for web developers building audio-video applications

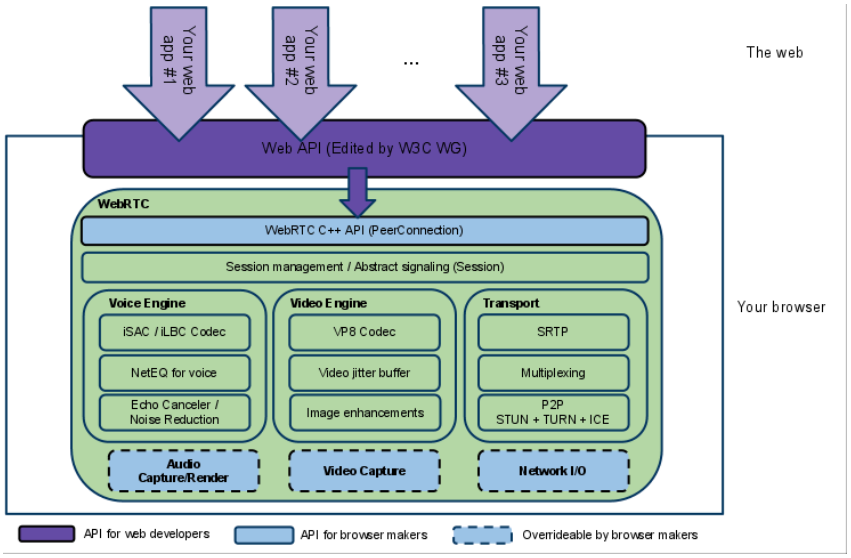

Figure 4.1: Overview of WebRTC architecture from official documentation [10]

Comprehensive documentation and community creating many additional libraries make using this framework rather convenient. One of the most popular open-source libraries is **aiortc**.

4.3.2 Aiortc

Aiortc was developed as a 'WebRTC for Python' [22]. Complexity (especially for beginners) and lack of Python bindings motivated authors of aiortc to design library and API which follow its Javascript counterpart. They pointed out that 'WebRTC (...) is tightly coupled to a media stack, making it hard to plug in audio or video processing algorithms' [22]. Our goal is to literally use the video processing algorithm, so we found this library and its examples extremely useful. We based our server on the aiortc example which performs video, audio and data channel establishing with a browser. It also allows pocessing streaming frames using OpenCV.

4.3.3 Flask

Flask is a so-called **micro-framework** with little dependencies to external libraries, which provides tools and technologies useful with building a small web application [34]. It could be helpful for ones who want to create web pages, blogs or a simple REST server. Technically, Flask is based on:

- Werkzeug WSGI (Web Server Gateway Interface is a popular specification for a universal interface between the server and the web app) - toolkit which implements requests-response objects, and some service functions
- Jinja2 template engine - template system helpful with rendering dynamic web pages

Flask itself does not support database handling, validation or extensive visual templates. Being lightweight is the factor we found crucial, since our server does not perform any complicated tasks. Running constantly in the background, it only has to:

- Receive POST request from mobile application containing filter specification
- Encode filter image and parameters
- Assign it to the IP of sender and store in particular folder
- Send confirmation response to the mobile app

We also decided to write simple Flask client in order to test connection to the server.

4.3.4 Server flow

The aforementioned servers work separately and do not directly interfere with each other and could perform tasks asynchronously. In that way, many clients can upload filter images and parameters as well as stream video without delays. Every client can change filter params and start a new stream in various order, but the usual client-server flow is depicted in Figure 4.2

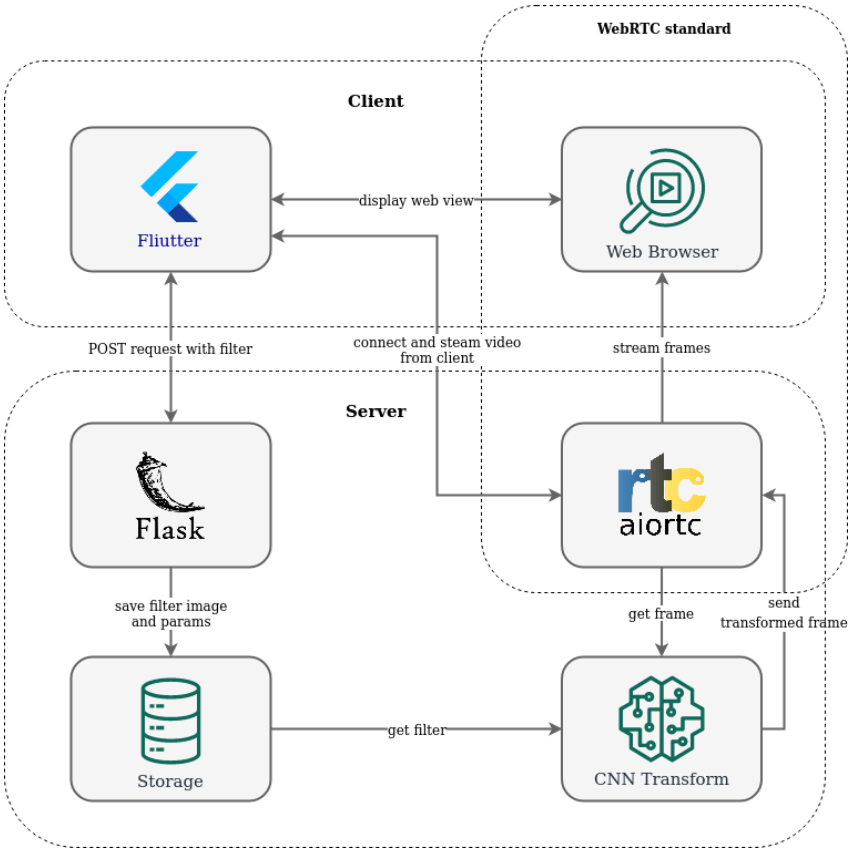

Figure 4.2: Client-server flow

In a nutshell, when user selects filter style the following actions happen:

1. Mobile app send POST request to the Flask server running on port 5000 including JSON with three fields: filter image encoded in base64 format, filter benchmark value in float, boolean information about color preservation during process
2. Flask server performs tasks described in the section 5.2.3
3. Mobile app ping server running on port 8080
4. Server establishes RTC-peer connection with browser running inside app
5. The stream is started - aiortc-based server catch frame and transform it using filter style

image uploaded in advance from storage

6. Transformed frame is added to RTC Media Stream Track and streamed back to the mobile browser

7. Browser displays video using WebView component

When streaming finishes or breaks (due to network error, application shutdown or any other error), server close connection gracefully but still stores last filter style parameters. If they are not specified during the next stream or the POST request to the Flask server fails, the server uses saved ones.

4.3.5 Alternative technologies

We had been trying several options before we chose to use aiortc. Our prime attempt was to adopt WebRTC Plugin for Flutter. It is a small project developed by the Flutter community containing Javascript server and Flutter client. That seemed to be a perfect solution that could accelerate our work. However, we encounter some obstacles in this undertaking. First of all, the mentioned plugin is poorly documented and error-prone what is a rather annoying combination. Secondly, it works efficiently only over the LAN network when we wish to connect to the server from any place. And the third thing is the communication crashes if the video is not streaming to/from the Chrome browser. Nevertheless, the project is gathering greater community what makes it promising in the time of growing popularity of Flutter.

4.4 Environment

Both model and server require multiple dependencies incl.:

- graphic card drivers
- Python packages
- Linux programs
- repositories from Github

Even if we have installed all needed dependencies, there still remains a problem with configuration, which can take even hours. The target location of our program was one of the shared machines in Poznan University of Technology (PUT), so we had to move the whole environment very quickly without making a mess. Finally, we decided to put our application into Docker container to deal with all those problems.

4.4.1 Docker

Thanks to the OS level virtualization, Docker allows us to create containers where we can put our program and all its dependencies. We can have multiple containers and manage them easily using command line or one of the GUI clients, like *Kitematic*. Docker containers are separated from each other so there is no risk to do any harm to other containers or hosts. However, we can make our containers communicate by mapping some of container ports to host.

Docker containers allow us to *pack everything up* and deploy on PUT machines. We deploy the model, the server, graphic card drivers, configuration files and all other dependencies into single container. Docker gives us the possibility to define whole building process in single file called *Dockerfile*. During container building we can copy files from local drive, download programs from the Internet or even run bash scripts. After having it built, we converted our container into image and uploaded it to the *Dockerhub* - a docker images library where everyone can upload their images. Last thing to do was just download image on computer and run graphic card configuration script which prepares software for particular GPU type. Project container is run with ports 8080 and 5000 mapped with same ports on host machine - where WebRTC and Flask servers are deployed. Client application connecting to computer on these ports are in fact connecting with container.

4.5 Mobile Application

This part describes the main technology used in client application which is written in Flutter. We also clarify why we have chosen specific technologies.

4.5.1 Flutter

Nowadays there are several frameworks that allow us to create mobile applications for both Android and iOS simultaneously, eg. React Native, Ionic, Xamarin. In 2015 Flutter joined this family and changed drastically situation on the mobile application market. Today, Flutter is the most popular multiplatform framework, which enables us to create not only mobile applications but also web from a single codebase.

Flutter is a framework created by Google but it's also an open-source project so everyone can propose new features and improvements. When we write Flutter application we use Dart - language developed by Google with C-style syntax. It runs on the Dart VM included in the SDK. A very interesting feature of Dart is that it optionally transcompiles to JavaScript so you can also create web applications.

Basic idea of Flutter sounds "Everything's a widget" [7] and widgets are quite similar to React "components". They can be combined in order to create more complex ones. The main purpose of widgets' usage is to build a whole UI and some of them are:

- Layout widgets - e.g. rows, columns, lists, grids, tables
- Style widgets - e.g. colors, fonts
- Structure widgets - e.g. buttons, sliders, switches
- Animation widgets - e.g. transition, fade in

Flutter comes with loads of prebuilt Material Design and Cupertino(iOS style) widgets which makes easy and fast to create great looking applications. There are Material Design icons, colors, typical UI interface elements like top and bottom bars, buttons, list, animations and many many more.

Widgets are split into two main categories [8]:

- Stateless - build only when the input data changes.
- Stateful - build when input data or state property changes

Difference between this two kinds of widgets is that the Stateless widgets never changes while Stateful ones can be modified during their lifetime. One of the examples of stateful widget is a simple checkbox. When we press checkbox it changes it's state to checked or unchecked and triggers rebuild on itself. When we want change to state of widget we must use special *setState* to made flutter re render this particular widget.

Flutter architecture is composed of many layers [7] but we can divide it into three main layers presented at Figure 4.3. Programmers have more contact with top layers of framework - these are Material and Cupertino widgets which are composed of basic widgets. Basic widgets use lower layers of framework. The lowest one is foundation which contains core classes, functions and constants.

From the beginning of our project we wanted to be able to develop new ideas very quickly. Python and PyTorch are technologies which definitely helped us keep high productivity during this time. First choose was Kotlin - a new modern programming language for Android applications. Main advantages of Kotlin are:

- officially supported by Google
- high performance - native application

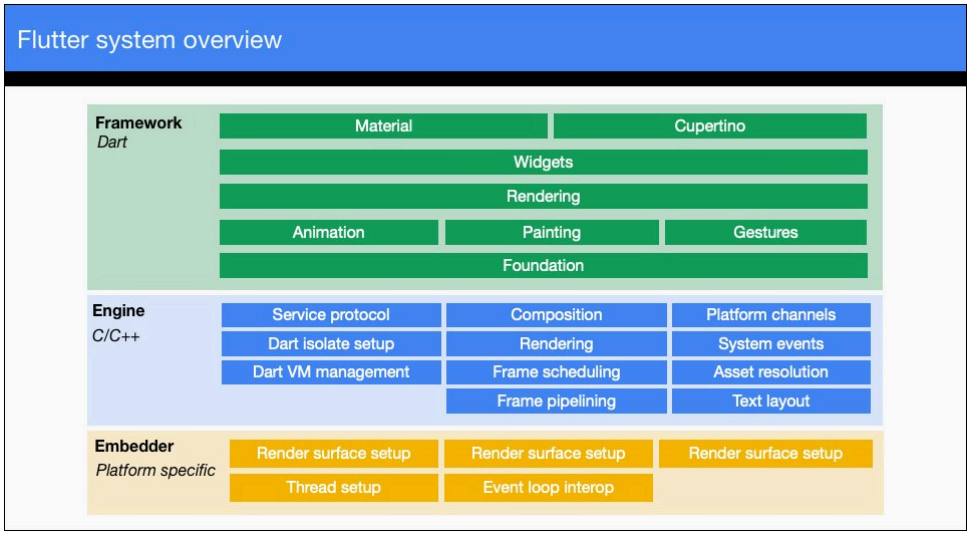

Figure 4.3: Flutter layers description [7]

- some great language features like null safety, readability or async functions
- having an experience in this language

The main drawback of creating mobile applications with Kotlin is that creating some simple things can often take a lot of time as well as effort. This was a main cause of rejecting this approach. We were looking for some other tools to create our client application and eventually we found Flutter - relatively new tool framework from Google which is still developing quite fast. The main advantage of Flutter is that we can develop your apps very quickly as there are numerous efficient, ready-to-use and highly customizable widgets. There is also a large community which creates new great widgets add publishes them on official website so everyone can download them and use in project. You can also modify or build new widgets on the top of it and then publish. If some widget would become very popular and useful it will may be include to official Flutter SDK. Flutter also makes it easy to create apps in beautiful Material Design style or its iOS equivalent - Cupertino. The main feature of Flutter, which is always mentioned by Google is of course creating applications for Android and iOS at the same time. Our main goal was to create application for Android so we treated this feature like an opened door. Flutter is also declarative, react style framework which we prefer while creating user interfaces. We have had previous experience with React on web so Flutter seemed easy to learn.

5 Mobile Application

This section describes functionality and design principles of implemented mobile application for style transfer. We present the whole application, screen by screen, and some of the difficulties we were facing during development.

5.1 Requirements

During planning process we decided that our application ought to meet all of the undermentioned requirements. User should be able to:

- manage filters in the gallery, where it is easy to pick filter image
- take photo directly inside the application and then use it as a filter
- set style transfer settings - like color preservation or scaling
- transfer style from filter to a real-time video

5.2 Design

Our assumption was to fill the application with marvelous image content. We used the greatest artworks from Wikiart [30] so a user should *feel communication* with the real art. To achieve this goal we let images take the whole screen and put them in the foreground. Most of the UI elements are transparent at some level so they do not cover the images. Buttons and icons are very subtle as well as kept in Google Material Design style. Animations are also gentle and they are introduced in order to improve user's experience rather than make spectacular visual effects.

5.3 Difficulties

To perform style transfer with application and meet all of the requirements we had to overcome two main difficulties:

- sending video to server and then receiving it back with applied style transfer (real-time streaming)
- implementing efficient image gallery which can load photos from smartphone's storage

Our main concept of this project was real-time streaming, so solving first problem was crucial. A great, relatively new tool for streaming video is WebRTC which, in general, enables real-time communication in browser. Someone may wonder how this can allow us to stream video in a

mobile application. Actually, we use special Flutter InAppWebView plugin which can display websites inside application. With this plugin we are able to move almost all logic associated with communication into server. The application is just opening website under hardcoded specific address.

As mentioned, the second task is to create an effective custom image gallery. Most smartphones take high resolution pictures which size is around 3-4 MB. It takes long time for Flutter to load big pictures like these, especially when we have to load twelve or even more photos. We need to load plenty of of pictures, but they could be in fact very small since they ought to be presented as the thumbnails. In addition, there is a problem with memory overflow. Namely when we are loading all pictures at once application was crashing after few second. So as, to face memory overflow and high loading times, we apply two key improvements: lazy loading and image resize in line with compression. The gallery loads only twelve pictures at the same time so memory will not be filled so quickly. The improved gallery is also resizing file to 400×400 resolution and then compression is applied. Thus, it lets to save memory and time.

5.4 Application walk-through

This part presents final client application and all its capabilities. Every screenshot comes from Huawei P10 Lite with Android 8.1 operating system. In the pictures we can see application design which was described in previous chapters.

5.4.1 Start screen

The star screen appears just after application is launched. It has only aesthetic purpose and is an introduction to the main part of the application. We can see here Vincent van Gogh picture *Seascape near Les Saintes-Maries-de-la-Mer* and it is just one of the 27 pre-built image-filters. The screen also contains the application name *ARTEXTURE*. If we would like to go to next screen we have to swipe up or tap arrow button at the bottom.

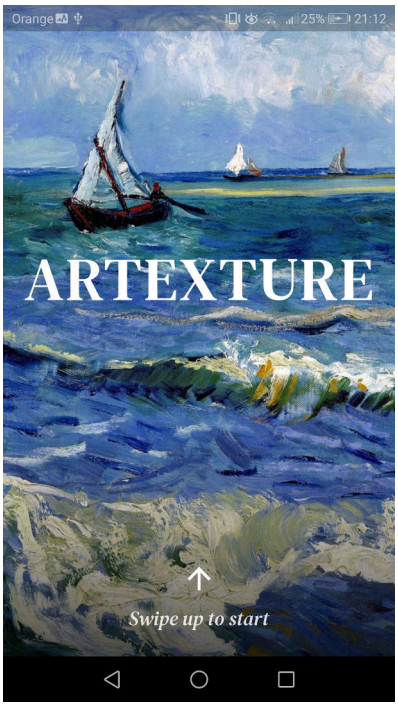

Figure 5.1: Start screen

5.4.2 Gallery

The gallery is the main screen of the application. It allows us to pick an image that we want to use as a filter. The bottom half of the screen is taken by a list of images and it is presented in Figure 5.2. There are 27 built-in paintings images built in the application. We can also use images from storage. To switch between these two you have to click the pallet icon on the image icon on top of the list. If we select one of the images from the list it appears immediately in the background. In order to have a good look at the whole selected image, there is a possibility

to hide the list by clicking on a down arrow icon (Figure 5.3). In the right upper corner there is settings icon and after tapping we can see a alert box (Figure 5.5) with two filter settings: *intensity* and *color preservation*. Next to the settings icon, there is a plus icon which leads us *Pick a filter* to the screen which is described on the next pages.

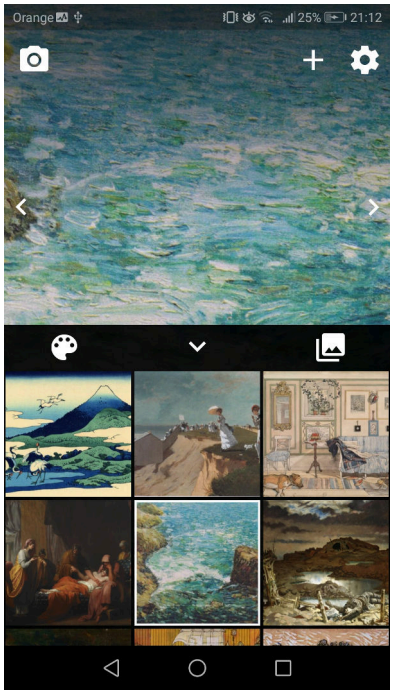

Figure 5.2: Unfolded gallery

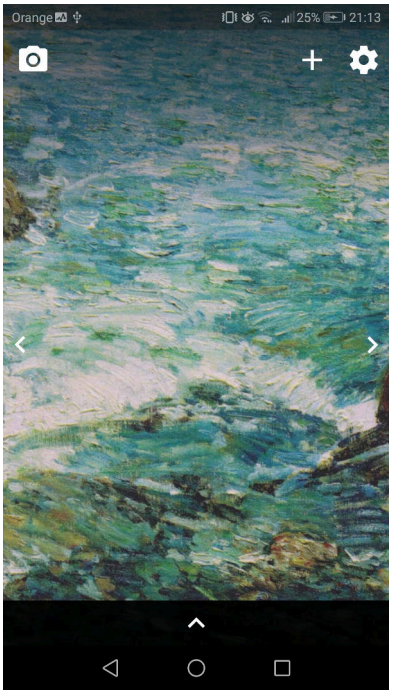

Figure 5.3: Folded gallery

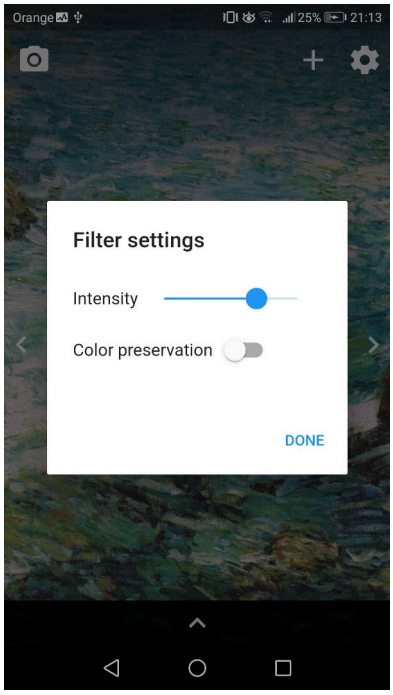

Figure 5.4: Settings

5.4.3 Style transfer screen

When we finally pick our filter and settings we can start style transfer. Whole screen screen is filled with transformed video. If we do not like the current filter or we want to stop style transfer all we need to do is push *X* button at the bottom of the screen. Afterwards, we go back to the gallery.

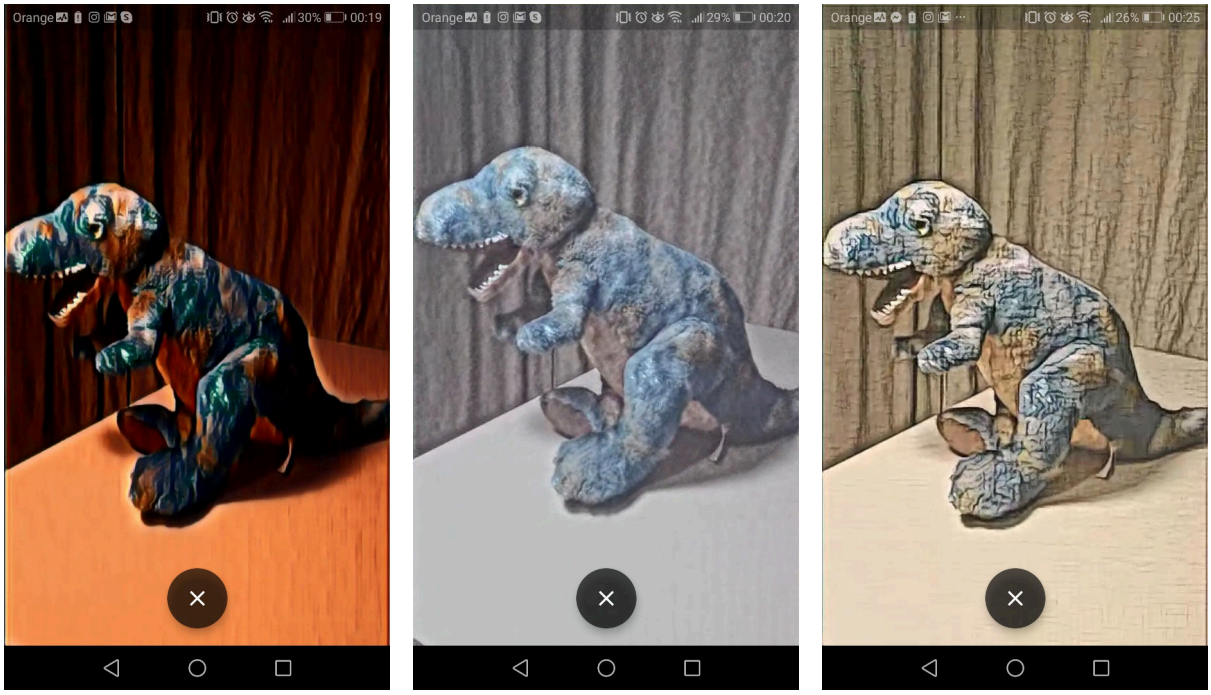

Figure 5.5: Style transfer with different filters

5.4.4 *Pick a filter* screen

While working on this project, we anticipated that there might be a situation that the users will notice a ravishing scenery or a breathtaking view. In these circumstances, they can use it as a filter. There is an opportunity to pick a photo quickly and then start style transfer immediately. In Figure 5.6 we can see the camera preview from the back camera. Users can also switch to front camera by clicking *switch camera* the icon in the top right corner. After we take a picture we are redirected to the screen presented on Figure 5.7 and then we can decide whether to use the picked photo as a filter or not. If we are satisfied with the filter image we have to press *Style it* button. After that, the style transfer begins.

Figure 5.6: Camera preview

Figure 5.7: Taken photo

6 Experiments

6.1 Model comparison

We begin experiments by comparing style transfer features and quality of the original and pruned network. We compare models on detailed styles with complex structure (Figure 6.1) and minimalistic, low-entropy styles (Figure 6.2). As we can see, pruned model performs on par with the original on complex styles. The only difference we notice for the first style is slightly lighter hue of sofa in the dog photo. Otherwise it's hard to find any differences in this column. In the second column the red jackets in kayak image stand out. This certainly is a drawback, considering image was supposed to imitate a sketch. Area under a bridge in last image is also less detailed and darker. In general pruned model, as expected, seems more likely to discard small details - compare for example surface of water in kayak image or elephant's back in first image. For styles with one dominant color, smaller model produces more uniform distribution of colors. It's visible in virtually all images in the last column; in this case images seem to be "watered down". Both of these traits are not necessarily drawbacks, for example the blue area under the bridge in last image of last column looks less coherent than darker version produced by small model. Discarding details might even be desirable for many styles.

We now compare networks on very simple styles. In degenerate case of entirely black style image they perform almost the same. Original model preserves a bit more structure, though it might not be visible. It makes sense results are "empty" since black color is represented by 0s and the first convolution layer amounts just to it's bias. Pruned network performs visibly worse in the second case. Very low contrast of images makes it hard to recognize the original image, while the original model has no problem approximately reconstructing content. The third column is very informative - even though it consists mostly of 2 colors (unlike a sketch image where various tones of grey are prevalent) both networks reconstruct content very well. This tells us structure and presence of some kind of edges in style image is very important for reconstruction - images in the last column are more blurry, even though color information is richer. Fourth column shows smaller model doesn't work very well when structure of style image is very sparse. While original model acts as (not very good) edge detector, pruned network produces seemingly overexposed images. Overall pruned model performs visibly worse on styles with very simple structure or no structure at all.

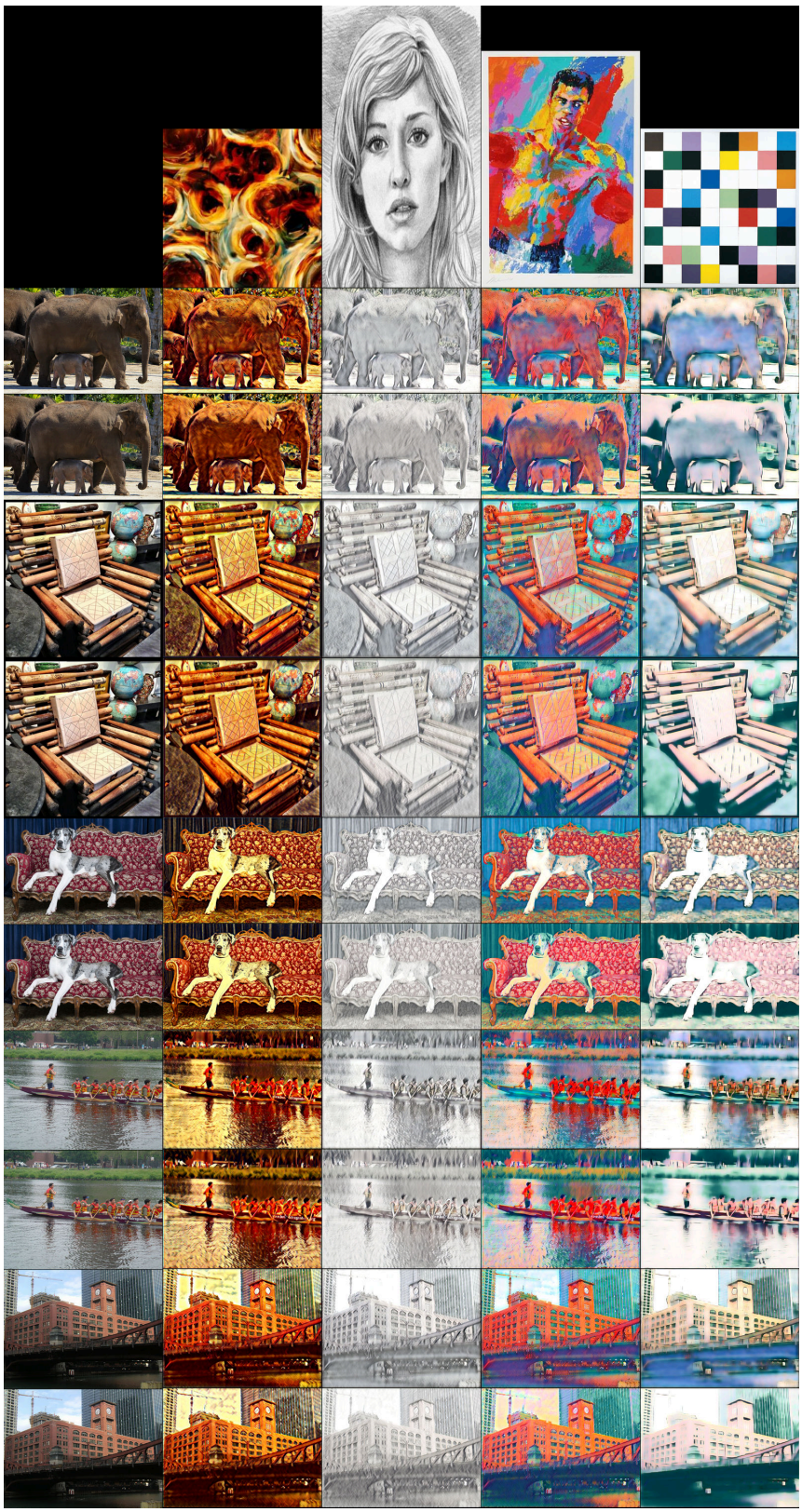

Figure 6.1: Comparison of original and pruned network on complex styles. For every pair of rows original model's results are placed in the top row and pruned model's results in bottom row.

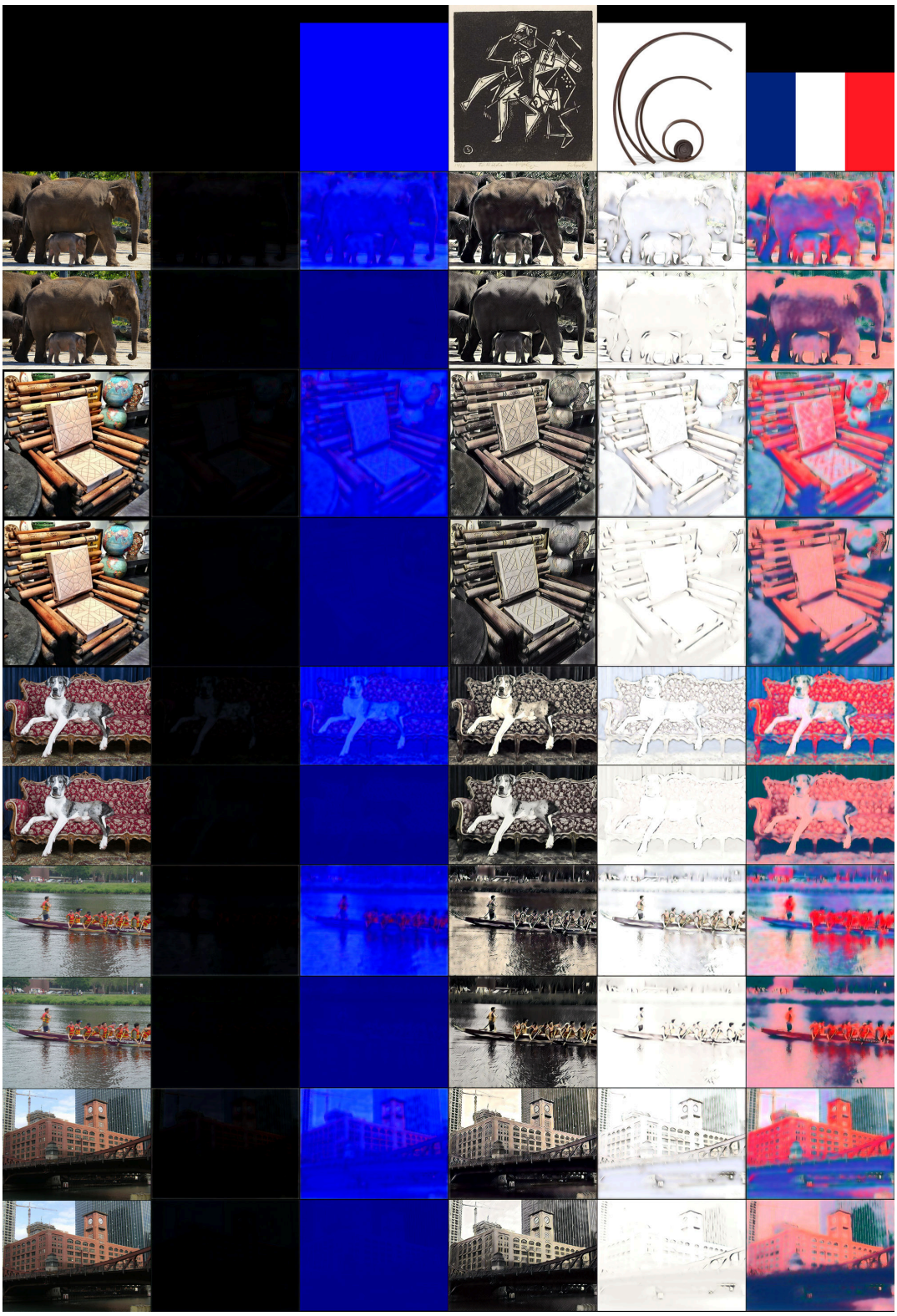

Figure 6.2: Comparison of original and pruned network on simple styles. For every pair of rows original model's results are placed in the top row and pruned model's results in bottom row.

6.1.1 Technical comparison

In this section we confront models' speeds and memory footprints. First we compare speed gains across different devices by taking measurements on three different NVIDIA GeForce GPUs of

	940M	960	1080 Ti
Original model	0.75	5	23
Pruned model	4.2	22	79
Pruned + TensorRT	7.1	26	70

Table 6.1: FPS of achieved by models on different GPUs

varying computational power. In all tests we use RGB images of resolution 1024x576. As we see speedup gained by pruning the original model is tremendous on every architecture, though it decreases with computational power. For pruned models using TensorRT this trend is even more emphasized. While the mobile 940M GPU speeds up by 75%, middle-range 960 only gains couple FPS, which at that point don't really make any difference from user's perspective. The most interesting case is that of high-end GTX 1080 Ti, which slows down when TensorRT is used. This might be exactly the result of it's computational power. Pruned model is no longer bottlenecked by factors the two weaker GPUs introduced, and as a consequence, it can be ran very efficiently. Since there is nothing to improve, tensor reshapes required by TensorRT might slow down inference.

In tables 6.2 we present memory requirements of original and pruned model. Both encoder and decoder get whole order of magnitude smaller which is very satisfying result. Size of transformation module however is only reduced by 25%. This is due to fact it contains two fully-connected layers, which can not be pruned by Distiller, each one transforming 32×32 matrix into a matrix of the same size. This gives $2 \times (32 \times 32)^2) = 2097152$ parameters. Simple calculation shows these fc layers constitute "just" 75% of original transformation module, but pruning increases their share to 97%. This is consistent with well-known fact that a VGG-like network's memory footprint stems mostly from fc layers but majority of inference time is spent on sequential execution of Conv layers.

6.2 Style scaling

We can control how much the result video or photo will be styled. For this purpose we added factor α - which is style weight set in mobile application. Using method of style scaling based on interpolation between feature maps given as input to decoder, what can be present as following equation:

$$T(c, s, \alpha) = g((1 - \alpha)f(c) + \alpha(f(c), f(s)))$$

When $\alpha = 0$ then the network will reconstruct the content style, $\alpha = 1$ is maximum level of styled, which was default style weight in previous experiments.

	Original model	Pruned TensorRT model
Content encoder	2.12 MB	152 KB
Style encoder	-	152 KB
Transformation module	11 MB	8.28 MB
Decoder	2.12 MB	150 KB
Sum	15.24 MB	8.73 MB

	Original model	Pruned TensorRT model
Content encoder	555 340	35 164
Style encoder	-	35 164
Transformation module	2 890 464	2 158 848
Decoder	555 075	35 091
Sum	4 000 879	2 264 267

Table 6.2: Total memory footprints (top) and number of parameters (bottom) of individual parts of each model. Style encoder is omitted in original model since that network only has one encoder.

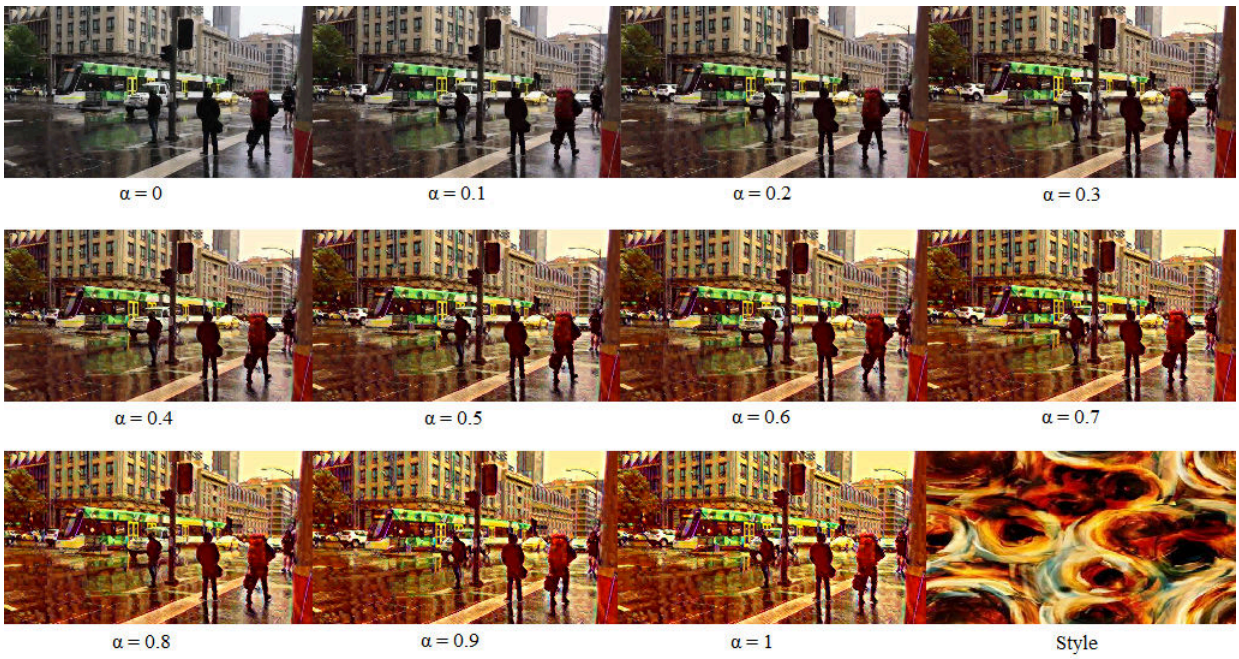

Figure 6.3: Comparison between different values of style weight using style scaling method

6.3 Color preservation

Style transfer gives us a content image in the style of the chosen style image. The result has all details like lighting, shapes, and colors. Using a method of the linear color transfer we can preserve the color of the content image and then transfer the style.

The key is to create new style S' by changing colors of style picture to colors matching with content, and then use style transfer algorithm.

We want each pixels $x = (R, G, B)$ to match mean and covariance of RGB values of new style

using following transformation:

$$x_s \rightarrow Ax_s + b$$

- where A is a 3x3 matrix and b is a 3-vector. We selected A and b in order to meet the requirements:

$$b = \mu_C - A\mu_S$$

$$A \sum_S A^T = \sum_C$$

where μ_S and μ_C are mean colors of the style and content, $\sum_S$ and $\sum_C$ are pixel covariances. That gives us a family of solutions to A. Then we use the Cholesky decomposition which depends on colors channel ordering, or 3D color matching formulations. We have chosen second method. The eigenvalue decomposition of a covariance matrix will be $\sum = UAU^T$, and we can define matrix square-root as $\sum^{1/2} = UA^{1/2}U^T$, with this and the transformation is following by:

$$A_{IA} = \sum_C^{1/2} \sum_S^{-1/2}$$

This linear color transfer method can not give satisfying results when content and style colors are too different. Therefore the color distribution may not match and we have a mismatch between result and content colors.

6.4 Video stability

One of goals for high-quality style transfer is stability. Let F_1 and F_2 be two video frames relatively close in time and T a style transfer algorithm transforming a single frame. We can define stability of T by mean difference between $|F_1 - F_2|$ and $|T(F_1) - T(F_2)|$. The smaller that difference, the more stable T is. This definition says that if a certain region doesn't change between F_1 and F_2, then it also shouldn't change between $T(F_1)$ and $T(F_2)$, that is the transformed video shouldn't flicker too much. Definition also implies changes in original video should be reflected in the stylized video, but it's an obvious requirement and it's the former property that we usually call stability. Stability can be explicitly guaranteed by use of special loss functions and proper training procedure [38, 11]. The model we base our work on however learns stability by design [24], so there is no need for any auxiliary techniques. We examine to what extent network pruning damages this property. We conduct the experiment using a single style image and 3 videos with motionless camera. Results are presented in Figure 6.4. For each of 3×3 grids, the third row shows heatmaps of differences between two images above it. A hypothetic perfectly stable style transform algorithm would produce the same heatmaps as the ones in left column. While both models perform reasonably well for second and third video,

they completely fail in the first case. The reason for such behaviour is progressive change of brightness of the sky which can't be seen in shown frames. Sky's brightness changes significantly as a result of some vehicle driving in front of camera. In other words stylizing poorly recorded video results in instability and noise. Comparing original and pruned model, we also see the latter is less stable for this video, though stripes produced by original model may seem even worse than overall noise output by pruned model. In the second case both networks perform equally well and their heatmaps are only slightly more noisy than the original heatmap. For the third video pruned model is only a bit more unstable than original model. Overall pruning seems to reduce stability only by a small margin provided original network preserves stability. In situation where original network fails to do so, pruned model magnifies errors though.

6.5 Style and content encoding

As mentioned in section 3.1 the encoder had to be copied due to pruning constraints. Since two resulting encoders were pruned and fine-tuned separately their sets of weights are different. In this section we shortly examine effects of using wrong combinations of encoder in pruned model. Results for four possible combinations of encoders are presented in Figure 6.5.

The first row presents default variant with proper configuration. We see that while style encoder S_E is able to encode content features to some degree (2nd row), content encoder C_E fails completely at style extraction (3rd and 4th row). This is expected, since C_E can usually safely discard most of color information in content image. On the other hand, S_E should pay attention to both color and shapes. Consequently images in second row are of a lot worse quality than first row, but content and style are still readily recognizable. Comparing 3rd and 4th row, we see content is reconstructed better in the latter one. Again, this should be the case, since content is encoded with S_E in 3rd row and with C_E in 4th row - C_E produces more accurate shape encoding than S_E so images in 4th row are sharper and less blurry.

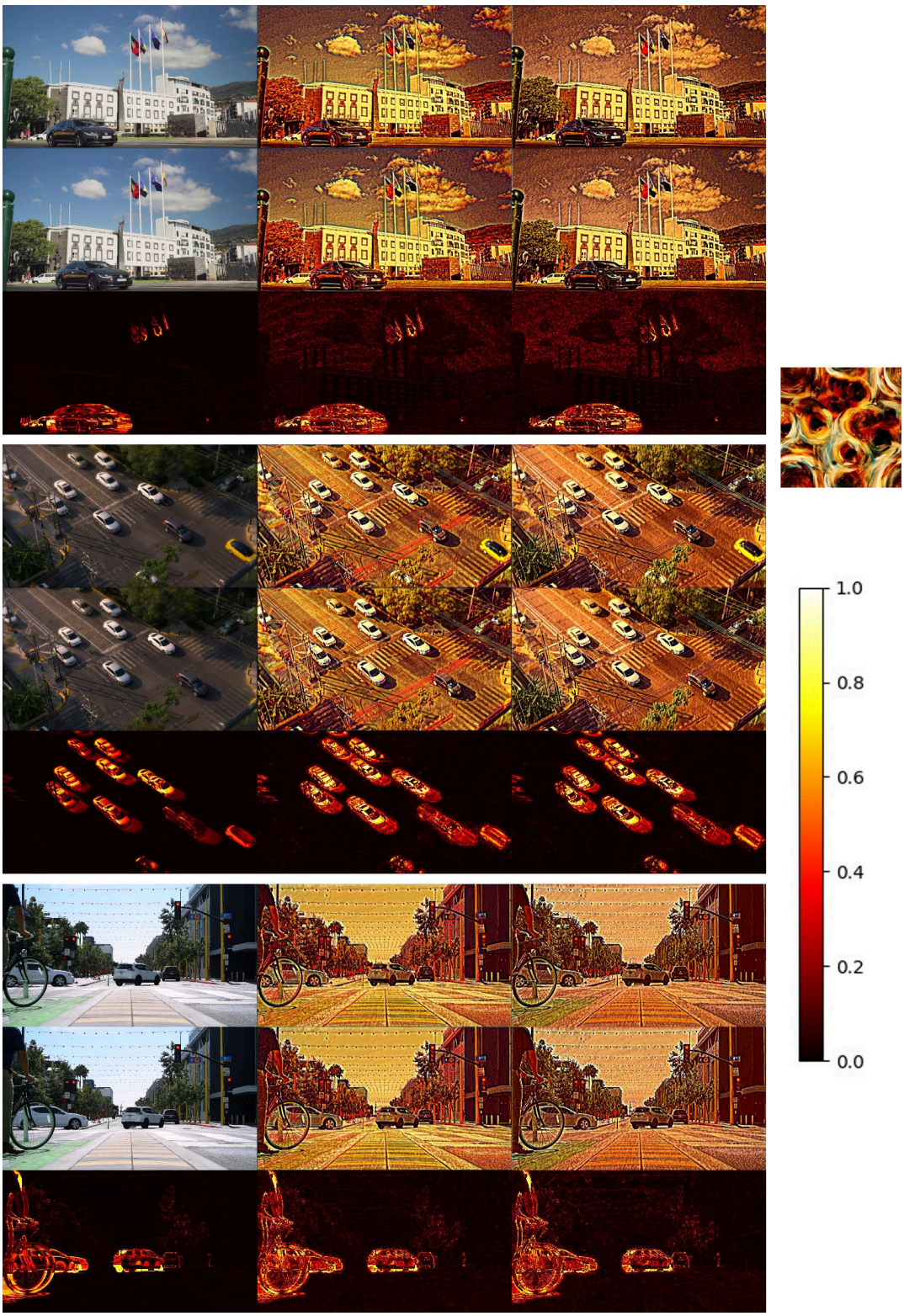

Figure 6.4: Comparison of stability between original (middle column) and pruned (right column) model. Left column shows original video frames. For each of 3×3 grids first two rows show two frames close two each other (1 to 5 frames of difference, depending on video's framerate) and the third row is heatmap of difference between them. To the right of the grid: used style image and heatmaps' scale.

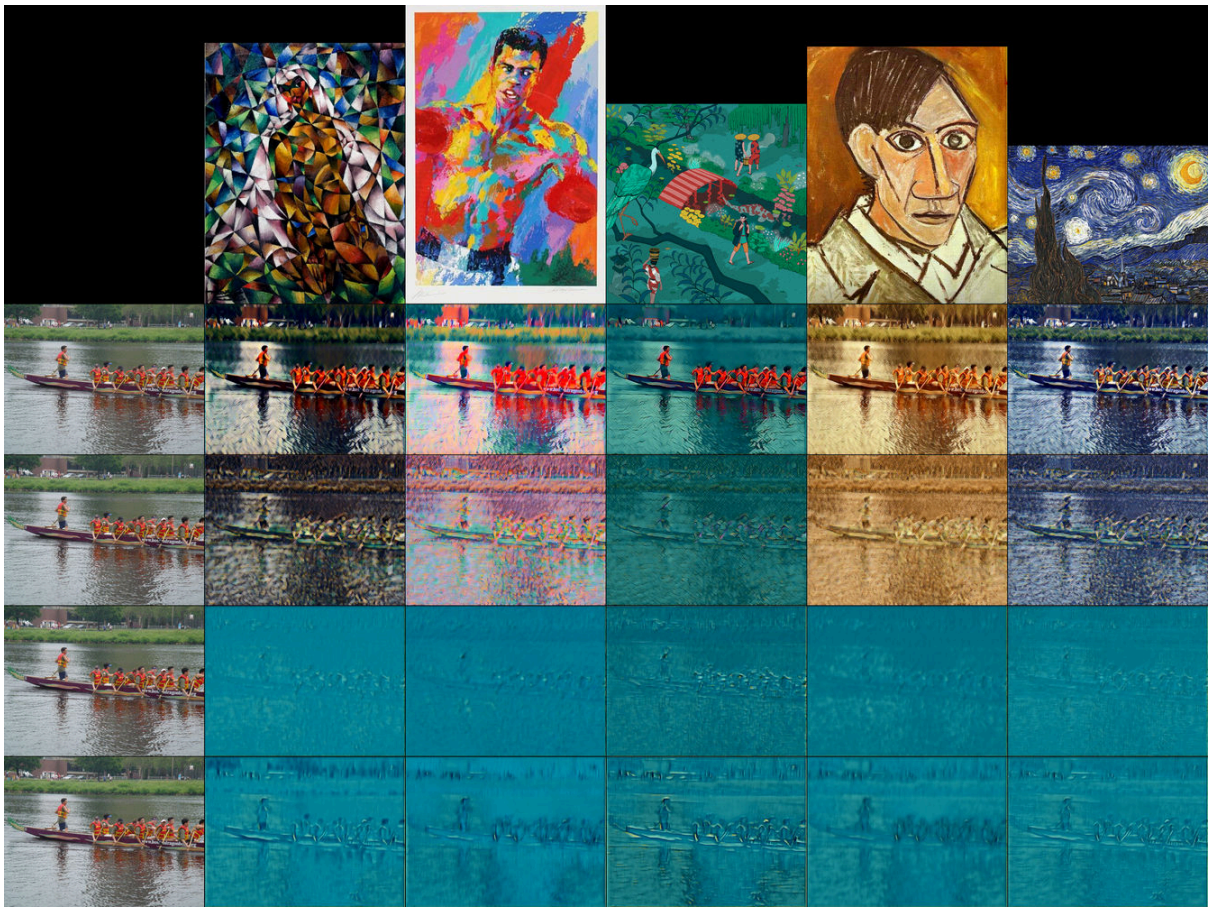

Figure 6.5: Comparison of various configurations of network with respect to encoder used for encoding style/content image. From top to bottom: $S_E(S_I)$ and $C_E(C_I)$, $S_E(S_I)$ and $S_E(C_I)$, $C_E(S_I)$ and $S_E(C_I)$, $C_E(S_I)$ and $C_E(C_I)$, where S_E and S_C are style and content encoders, S_I and C_I are style and content images.

7 Conclusion

7.1 Results

The final result of the project is a mobile application which allows users to pick any photo and then apply it as a filter in real-time video. Figure 7.1 presents screenshots taken from style transfer screen of an application. They show how filters with different textures and colors affects photos. In order to make our application work we had to also create a back-end environment which is mainly a server with style transfer neural network.

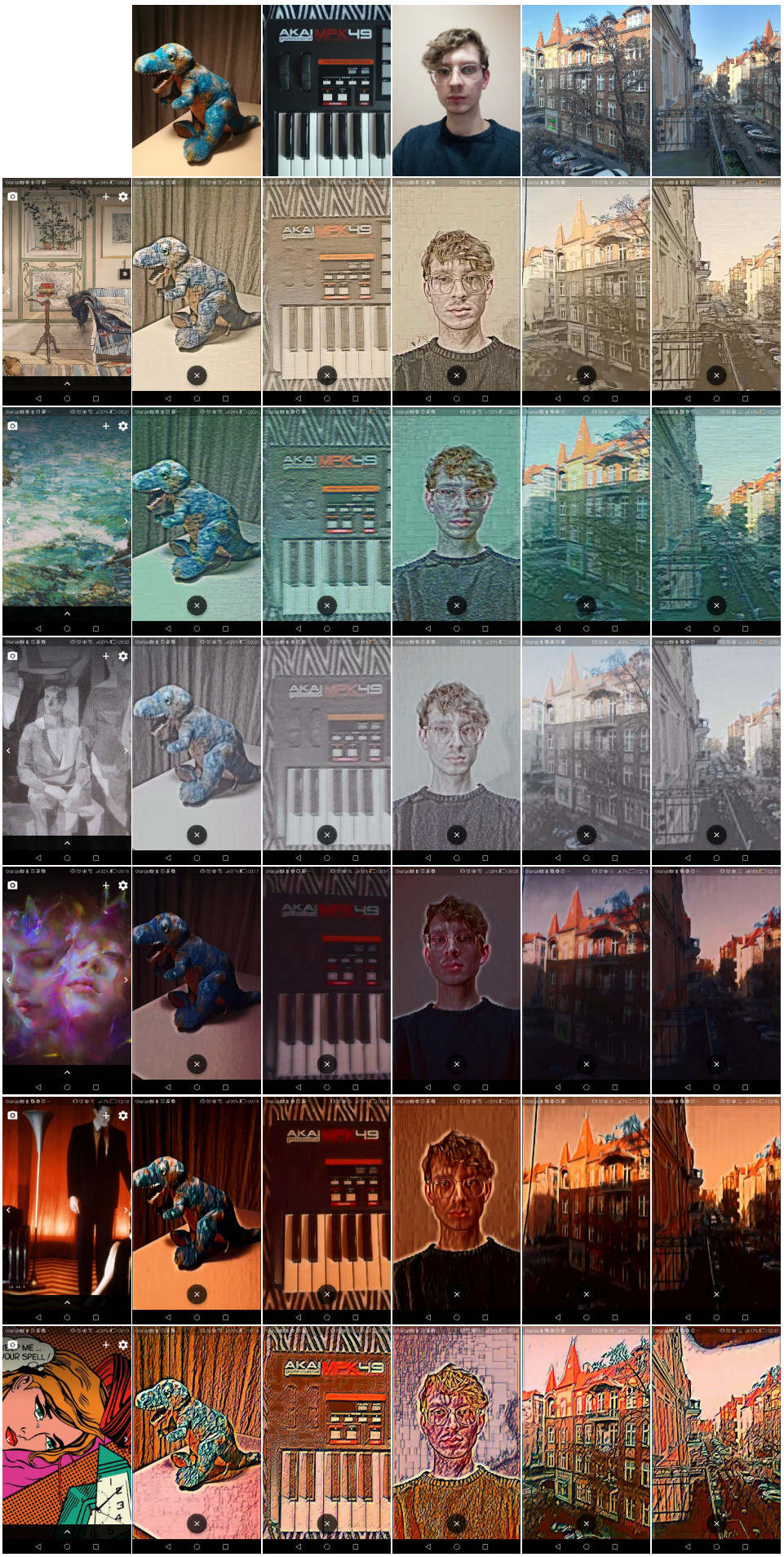

Figure 7.1: Style transfer in mobile application

7.2 Future Works

In this section we outline ideas and development directions we found interesting throughout the development of this project.

- **Feature encoding** To construct encoder [24] train VGG-like architecture to classify MSCOCO images. While dataset is rather large (80000 images), training classifier is probably the simplest way to obtain deep extractor of meaningful image features. Training encoder to perform other, more demanding, computer vision tasks like segmentation or image tagging might result in more robust features better suited for style transform. It would be particularly interesting to examine whether use of encoders trained in unsupervised manner (e.g. [45, 18]) would make any difference. Such models need to be able to perform tasks demanding even for humans, like generating full image based on only a port of it. As a side effect representations produced by their encoders must contain useful information for range of computer vision tasks. Even if it didn't influence network's output, the it might perhaps ease the training and accelerate convergence of style transform models

- **More sophisticated training** Image classification is a well-defined task with clear metric. Because of this simplicity, papers introducing new pruning algorithms tend to benchmark performance on classifiers. As a result methods for efficient pruning of picture to picture models were not studied so thoroughly. Expanding loss function used during retraining of pruned network by some form of knowledge distillation might enable even more severe pruning of such models. One might for example penalize ℓ_2 distance between outputs of network being pruned and original network. This would help shorten retraining phase since ℓ_2 gradient would be very large right after removal of some filters and quickly steer pruned model's parameters towards original. Probably even better idea would be matching encodings of images produced by respective networks. This would help smaller network closely mirror stylization performed by the original model.

- **Style interpolation and mixed styles using spatial control** We also want to add some new functionalities to the application like Style interpolation, co mixed at least two different styles to styled result. By giving on input to decoder convex combination, we can interpolate between styles and create new one.

  Next feature are mixed styles using spatial control. We can choose parts of content image to styled each part in diferent style. This effect can be reached by using style transfer separately using different input statistics. Also it could be extend to color preservation in chosen part of image.

References

[1] Frankle, J. and Carbin, M. (2018). The lottery ticket hypothesis: Finding sparse, trainable neural networks.

[2] Gatys, L. A., Ecker, A. S., and Bethge, M. (2015). A neural algorithm of artistic style.

[3] GitHub (2019a). Onnx simplifier. https://github.com/daquexian/onnx-simplifier.

[4] GitHub (2019b). Onnx-tensorrt. https://github.com/onnx/onnx-tensorrt.

[5] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

[6] Google (2019). Machine learning glossary. https://developers.google.com/machine-learning/.

[7] Google Flutter (2019a). Flutter technical overview. https://flutter.dev/docs/resources/technical-overview.

[8] Google Flutter (2019b). Stateful and stateless widgets. https://flutter.dev/docs/development/ui/interactive.

[9] Google TensorFlow (2019). Tensorflow playground. https://github.com/tensorflow/playground.

[10] Google WebRTC (2019). Webrtc documentation. https://webrtc.org.

[11] Gupta, A., Johnson, J., Alahi, A., and Fei-Fei, L. (2017). Characterizing and improving stability in neural style transfer.

[12] He, Y., Zhang, X., and Sun, J. (2017). Channel pruning for accelerating very deep neural networks.

[13] Hertzmann, A., Jacobs, C. E., Oliver, N., Curless, B., and Salesin, D. H. (2001). Image analogies. *Proc. SIGGRAPH 2001.*

[14] Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network.

[15] Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., Le, Q. V., and Adam, H. (2019). Searching for mobilenetv3.

[16] 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.

[17] Huang, X. and Belongie, S. (2017). Arbitrary style transfer in real-time with adaptive instance normalization.

[18] Hénaff, O. J., Srinivas, A., Fauw, J. D., Razavi, A., Doersch, C., Eslami, S. M. A., and van den Oord, A. (2019). Data-efficient image recognition with contrastive predictive coding.

[19] Jing, Y. (2019). Neural style transfer papers. https://github.com/ycjing/Neural-Style-Transfer-Papers.

[20] Johnson, J., Alahi, A., and Fei-Fei, L. (2016). Perceptual losses for real-time style transfer and super-resolution.

[21] Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25.

[22] Lainé, J. (2019). Aiortc library documentation. https://aiortc.readthedocs.io/.

[23] Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. (2016). Pruning filters for efficient convnets.

[24] Li, X., Liu, S., Kautz, J., and Yang, M. (2018). Learning linear transformations for fast arbitrary style transfer.

[25] Lin, T.-Y., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C. L., and Dollár, P. (2014). Microsoft coco: Common objects in context.

[26] Ma, N., Zhang, X., Zheng, H.-T., and Sun, J. (2018). Shufflenet v2: Practical guidelines for efficient cnn architecture design.

[27] Mihajlovic, I. (2019). Everything you ever wanted to know about computer vision. https://towardsdatascience.com/.

[28] Missinglink (2019). 7 types of neural network activation functions. https://missinglink.ai/guides/neural-network-concepts/.

[29] Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. (2016). Pruning convolutional neural networks for resource efficient inference.

[30] Nichol, K. (2016). Painter by numbers wikiart. https://www.kaggle.com/c/painter-by-numbers.

[31] Nvidia (2019a). Nvidia cuda documentation. https://developer.nvidia.com/cuda-toolkit.

[32] Nvidia (2019b). Nvidia tensorrt documentation. https://developer.nvidia.com/tensorrt.

[33] OpenCV (2019). Open source computer vision documentation. https://docs.opencv.org/master/.

[34] Pallets (2019). Flask documentation. http://flask.palletsprojects.com/.

[35] Polyak, A. and Wolf, L. (2015). Channel-level acceleration of deep face representations. *IEEE Access*, 3.

[36] PyTorch (2019). Pytorch documentation. https://pytorch.org/docs/.

[37] Reed, R. D. and Marks, R. J. (1998). *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. MIT Press, Cambridge, MA, USA.

[38] Ruder, M., Dosovitskiy, A., and Brox, T. (2016). Artistic style transfer for videos. *Pattern Recognition*, page 26–36.

[39] Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:210–229.

[40] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks.

[41] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition.

[42] Stanford University (2019). Deep learning tutorial by stanford university. http://deeplearning.stanford.edu/tutorial/.

[43] Team, D. (2019). Machine learning glossary and terms. https://deepai.org/machine-learning-glossary-and-terms/.

[44] Ulyanov, D., Vedaldi, A., and Lempitsky, V. (2016). Instance normalization: The missing ingredient for fast stylization.

[45] van den Oord, A., Li, Y., and Vinyals, O. (2018). Representation learning with contrastive predictive coding.

[46] Zeiler, M. D. and Fergus, R. (2013). Visualizing and understanding convolutional networks.

[47] Zhang, X., Zhou, X., Lin, M., and Sun, J. (2017). Shufflenet: An extremely efficient convolutional neural network for mobile devices.

[48] Zhou, H., Lan, J., Liu, R., and Yosinski, J. (2019). Deconstructing lottery tickets: Zeros, signs, and the supermask.

[49] Zhu, M. and Gupta, S. (2017). To prune, or not to prune: exploring the efficacy of pruning for model compression.

[50] Zmora, N., Jacob, G., Zlotnik, L., Elharar, B., and Novik, G. (2018). Neural network distiller.