5-2020

Divide and Conquer in Neural Style Transfer for Video

Paul Galatic
pdg6505@rit.edu

DIVIDE AND CONQUER IN NEURAL STYLE TRANSFER FOR VIDEO

APPROVED BY

SUPERVISING COMMITTEE:

Dr. M. Mustafa Rafique, Supervisor

Dr. Michael Mior, Reader

Dr. Minseok Kwon, Observer

DIVIDE AND CONQUER IN NEURAL STYLE TRANSFER FOR VIDEO

by

Paul Galatic

THESIS

Presented to the Faculty of the Department of Computer Science

Golisano College of Computer and Information Sciences

Rochester Institute of Technology

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science

Rochester Institute of Technology

May 2020

Acknowledgments

I am grateful for the assistance provided during the development of my thesis by Dr. M. Mustafa Rafique, whose rigorous standards and patient attention helped me lift my work to the highest possible standard of quality.

I am also thankful for the support offered to me by Dr. Michael Mior and Dr. Minseok Kwon, whose suggestions strengthened the content and style of my work.

I owe the inspiration for this thesis to the many wonderful individuals at RIT-AI, who allowed and encouraged my exploration of various concepts in artificial intelligence. In particular, I appreciate the efforts of Dylan Bowald, Dylan Jackson, and Perry Deng in their administration of the club alongside me during my time there as club president.

I would not be writing this thesis without the concentrated efforts of a host of teachers, professors, and family. It takes a village to raise a child and a university to raise a Master.

Finally, I am grateful for the CloudLab platform. Its computers and its staff allowed me to perform the testing required to demonstrate the efficacy of my method.

Abstract

DIVIDE AND CONQUER IN NEURAL STYLE TRANSFER FOR VIDEO

Paul Galatic

Supervisor: Dr. M. Mustafa Rafique

Neural Style Transfer is a class of neural algorithms designed to redraw a given image in the style of another image, traditionally a famous painting, while preserving the underlying details. Applying this process to a video requires stylizing each of its component frames, and the stylized frames must have temporal consistency between them to prevent flickering and other undesirable features. Current algorithms accommodate these constraints at the expense of speed.

We propose an algorithm called Distributed Artistic Videos and demonstrate its capacity to produce stylized videos over ten times faster than the current state-of-the-art with no reduction in output quality. Through the use of an 8-node computing cluster, we reduce the average time required to stylize a video by 92%—from hours to minutes—compared to the most recent algorithm of this kind on the same equipment and input. This allows the stylization of videos that are longer and higher-resolution than previously feasible.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Neural Style Transfer (NST) [1, 3, 4], also called artistic style transfer, is a category of neural algorithms designed to redraw a given image in the style of another image. These algorithms insert characteristics of the style image, e.g. brush strokes and colors, while preserving the major structural features of the input image. NST has potential commercial and research applications in the fields of animation [5, 6], image rendering [7] and data augmentation [8, 9, 10], among others. Various examples of NST are displayed in Figure 1.1.

NST for images has been extensively studied in the past half-decade. The most recent algorithms in the field are extremely fast and flexible, able to combine any two images in nearly real time through the use of feed-forward neural networks [1, 11]. However, videos are currently challenging to stylize in a visually pleasing way for both theoretical and practical reasons.

1.1 The Essential Nature of Neural Video Stylization

In NST for images, input images are independent of each other, carrying no important information between them. A common initial approach to NST for video is to stylize every frame of a video independently of every other frame. This method is fast, yet unsatisfy-

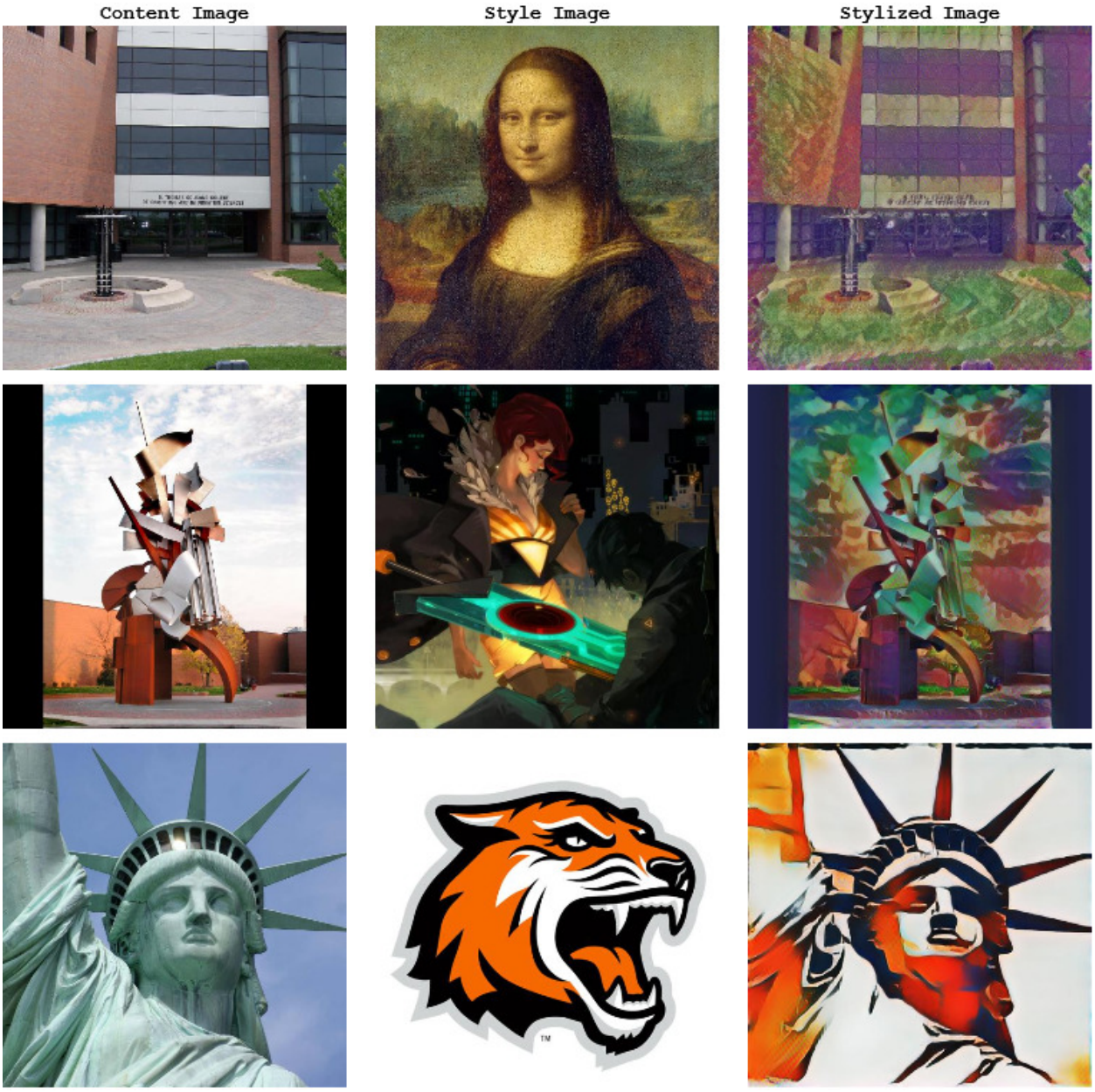

Figure 1.1: Various examples of neural style transfer for images. The input images (left column) are, from top to bottom: An image of the outside of the Golisano building at RIT, an image of *The Sentinel*, a sculpture at RIT, and a close-up of the Statue of Liberty. The style images (center column) are, from top to bottom: The *Mona Lisa*, an image from *Transistor* by Supergiant Games, and a previous version of the RIT Tigers logo. Stylization was performed with an implementation of the technique described by Ghiasi et al. [1]. The implementation is called Magenta [2].

ing, because it cannot consider similarities between adjacent frames. This makes stylized frames differ wildly from each other, producing flickering artifacts and the illusion of frantic movement in a scene.

This effect is clearly undesirable, and solving it requires the introduction of novel constraints that ensure consistency of stylization between related frames. These constraints are extremely effective, yet they come at a severe cost of speed.

1.2 The Current Available Tools

The best neural video stylization algorithm known at the time of writing is Fast Artistic Videos (FAV) by Ruder et al. [12]. FAV promises, with a proper setup, to stylize videos nearly in real time. Despite our best efforts, we were unable to reproduce this rapid stylization speed. We believe the fault for this lies with auxiliary code on which FAV depends.

Near-real-time stylization with FAV requires advanced equipment and software. First, it requires a graphics processing unit (GPU). GPUs are used to parallelize certain expensive computer operations in a process often called "GPU acceleration." The use of GPUs is a de-facto standard for improving the efficiency of many neural algorithms, including FAV.

In order for FAV to take full advantage of GPU acceleration, it must use software libraries that can communicate with the GPU and provide parallelized instructions. Currently, the most popular of these libraries is the Compute Unified Device Architecture (CUDA) [13] library, which is built for general-purpose parallelization. Because neural algorithms benefit so substantially from GPU acceleration, an additional library was developed specifically for their benefit, called the CUDA Deep Neural Network (cuDNN) library [14]. Both CUDA

and cuDNN have various versions, and not all versions are backwards-compatible with each other, or are compatible with every main version of the Ubuntu operating system. This matrix of incompatibilities is at least a manageable inconvenience.

FAV also depends on Torch7 [15], an open-source Lua library based on the Torch deep-learning framework [16]. After FAV was published, several users reported that their GPU-accelerated FAV installations were generating improperly stylized videos that bore no resemblance to either the original video or the style that was supposed to be applied. Despite following all of the authors' installation instructions and suggested workarounds, we observed the same results.

One of the authors proposed that an update to Torch7 or another dependency broke GPU acceleration in FAV after Ruder et al. published their algorithm in 2018[1]. The last substantive update to Torch7 was in September 2017, and so the most likely explanation is a flaw in one of the libraries on which FAV depends. Because FAV has exited active development, there is little hope of this issue being definitively fixed.

The next best alternative is to use a slower version of FAV that runs on the central processing unit (CPU) only, which allows FAV to generate high-quality output at a much slower pace. However, a major strength of Torch7 is its support of GPU acceleration via CUDA and cuDNN [15], and without that strength, its weaknesses are glaring and obvious; for example, sparse documentation made it difficult to determine the purpose of some functions.

To avoid these weaknesses, we base our work on the neural framework pyTorch [17], which is written in Python, a well-documented language that is easy to read and debug. PyTorch

[1]See https://github.com/manuelruder/fast-artistic-videos/issues/7

is considered among the fastest and most straightforward deep-learning libraries currently available, explaining its popularity in recent research [18].

1.3 Thesis Objectives

Our first objective is to design an algorithm that is functionally equivalent to FAV. A distributed version of an algorithm and its serial original must both create the same output in order to ensure that the process of distribution is correct. Because FAV uses Torch7 and our new algorithm uses pyTorch, we must determine whether or not the underlying differences in their respective platforms, if any, significantly alter the quality of the stylized videos they produce—hence, "functional equivalence" rather than equivalence outright.

Our second objective is to find an efficient and consistent algorithm to distribute the work required by FAV across a computing cluster.

1.4 Thesis Contributions

In this thesis, we introduce a novel algorithm called Distributed Artistic Videos (DAV) that makes substantial improvements to FAV. We prove that the discrepancies between the outputs of FAV and DAV are few and trivial.

Our algorithm can intelligently distribute its workload across a cluster of computers and drastically reduce the amount of time required to stylize a video. Over 8 computers, the time required to perform stylization is reduced on average by 92%.

We also introduce the standard frames per cut (SFC) metric, a novel measure which helps quantify the extent to which distribution of labor is possible on some inputs.

1.5 Thesis Layout

In Chapter 2, we briefly summarize the important foundational works up to and including the one that introduced FAV. We also introduce other works on the bleeding edge of NST that provide hints for future research.

In Chapter 3, we introduce the content and style loss functions that form the foundation of NST. We then detail how FAV applies these loss functions together with optical flow data and the temporal loss function. Finally, we describe how DAV distributes the work of stylization across multiple computers and propose several improvements to DAV based on what we learned while testing it.

In Chapter 4, we describe the evaluation criteria and experiments we used to determine to what extent DAV lives up to our objectives. We also introduce the SFC in this chapter.

In Chapter 5, we demonstrate the functional equivalence of FAV and DAV with both quantitative and qualitative results. We then analyze our dataset using the SFC metric and afterward measure the speedup we were able to attain with DAV over FAV in our experiments. At the end of this chapter, we discuss our observations on the process of neural video stylization and describe potential future research on various unsolved, yet potentially solvable problems in NST.

In Chapter 6, we conclude this thesis by restating the significance of DAV and highlighting the key findings of our work.

Chapter 2

Background and Literature Review

In this chapter we review NST, starting with some important general concepts, then moving on to its foundational works, then several important intermediary works, and finally the bleeding-edge algorithms that define the limits of what NST can currently accomplish. We cover only the works that we believe are most important to understanding and contextualizing our own. For those seeking a more comprehensive review of NST, especially for still images, we recommend "Neural Style Transfer: A Review" by Jing et al. [19].

2.1 Topics Related to NST for Video

In the following sections, we describe several topics that we believe are important to understanding our work, though they are used in a far broader context than just NST.

2.1.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) are the most common building blocks for any neural algorithm that processes images. The first CNN was introduced in 1980 by Kunihiko Fukushima under the title *Neocognitron* [20], and its importance cannot be overstated—NST, among many other powerful neural algorithms, are only possible through the use of CNNs.

A CNN is more efficient at recognizing patterns in image data than any other network known at the time of writing. The vast majority of neural algorithms that are designed to solve problems with a visual element use at least one convolutional layer. This is because convolutional layers operate on patches of their input at a time, rather than point-by-point, and so can perform feature extraction more effectively than its alternatives in identifying spatial relationships. CNNs are also used in non-visual problems for the same reason.

2.1.2 Optical Flow

Optical flow is an estimate of how pixels move between two images. It is typically used in the context of object tracking [21] or activity recognition [22, 23]. State-of-the-art video stylization algorithms, including the one we propose, also rely on optical flow files to operate.

The two main types of optical flow are sparse optical flow and dense optical flow. In sparse optical flow, for example the popular Lucas-Kanade method [24], only a few "feature" pixels are tracked. Dense optical flow, on the other hand, tracks every single pixel. In other words, dense optical flow between two images is an estimate of how pixels in the first image become pixels in the second image. NST for video requires dense optical flow data, and so we use methods that can calculate it.

The first algorithm for calculating optical flow was introduced in 1980 by Horn and Schunck [25]. It computes dense optical flow by searching for differences in image brightness and tries to generate smooth estimates. This seminal algorithm has inspired many other optical flow estimators, and so is worth mentioning. However, we will omit for the sake of brevity many intermediary optical flow algorithms and focus on those that are relevant to

our thesis.

In 2003, Farnebäck [26] introduced a simple and efficient method to calculate dense optical flow between two grayscale images. It is based on his dissertation from the previous year [27] that describes in detail how to use polynomial expansion to estimate motion between images. By converting the neighborhood of each pixel of an image into polynomials and comparing how those polynomials change between images, displacement between images is measured, and that measurement is refined to estimate optical flow.

The Farnebäck algorithm is far from the most accurate optical flow algorithm currently available, but it avoids the computational overhead of its modern, neural network-based alternatives.

An example of such an alternative is the combination of Deepflow2 and DeepMatching by Weinzaepfel et al. [28]. Computing optical flow is a task particularly suited to CNNs. Given a large network, high-quality data, powerful equipment, and some time, a model can be trained to compute crisp, accurate flow files. Moreover, neural networks can handle faster motion between frames without degradation in the accuracy of its output.

In order to understand how pixels move between images, it is helpful to know which pixels in each frame refer to the same object. DeepMatching uses a neural architecture similar to a deep CNN, alternating convolutional layers with maxpooling. It takes a heirarchical, bottom-up approach, starting with the fine details of both images and working up to assess which areas exhibit the same features and thus correspond to each other.

Deepflow is a variational approach that uses the output of DeepMatching to improve its inferences. It is evaluated both by the accuracy of its estimates, the smoothness of the flow

it generates, and how well its predictions correspond to a simpler precomputed vector field.

2.1.3 Normalization

There are several different kinds of normalization. Some exist independently of NST and others were created specifically to improve NST. In this section we describe two important normalization techniques that are either used by most NST algorithms or form the basis for later methods.

Ioffe and Szegedy [29] describe a batch normalization (BN) neural layer that has two learnable parameters, one expressing mean and the other expressing standard deviation. The network tunes these parameters while training and so learns to normalize the training dataset in a way that helps it minimize its objective function. BN layers effectively preprocess the input to the next hidden layer. This improves training stability, allowing networks to solve previously infeasible problems and improving convergence speed.

Because normalizing across the entire dataset introduces noise, BN is less useful in noise-sensitive networks [30]. It appears that NST benefits from a slightly different approach. Instance normalization (IN) by Ulyanov et al. [31] adjusts BN by allowing the layer to perform normalization over each input independently of the others. Rather than normalizing across the entire batch, IN has the network normalize over each image, preserving many of the benefits of BN without introducing much noise.

2.2 Research in NST for Images and Video

There are dozens of different approaches to NST, differing primarily in their loss functions and optimization strategies. The most commonly used loss functions measure, at minimum, how well the content of the input image matches the content of the output and how well the network applies the style. An interesting byproduct of the various network structures used across the field of NST is that each of their outputs are qualitatively different from each other even when given the same input and style images. The resulting effect is similar to comparing the work of several different professional artists who are all commissioned to paint the same vase in the same color palette. This phenomenon is difficult to quantify because the majority of work in NST is designed to demonstrate improvements in capacity, flexibility, and speed, rather than visual quality, if such an improvement could be objectively measured at all.

2.2.1 A Neural Algorithm of Artistic Style

The seminal work by Gatys et al. [3,32] begins with an image of random noise and shapes it so that it exhibits both the stylistic aspects of a given style image and the major features of a given input image, which is also commonly called the content image. Of course, these questions must be quantified in order to properly direct a neural network, and so Gatys et al. introduced the content and style loss functions, which are described in more detail in Section 3.1. For now, consider the content loss to be a measure of how well the output image reproduces the major features of the content image and consider the style loss to be a measure of how well the output image reflects the "essence" of the style image. Because

the optimization process happens on an image-by-image basis, Gatys et al. [3] has a very low throughput, which later algorithms sought to address.

2.2.2 Perceptual Losses for Real-Time Style Transfer and Super-Resolution

Johnston et al. [4] train networks to minimize perceptual loss. Perceptual loss compares high-level features between images rather than granular details, and so is much more in tune with the priorities of NST. That said, we observe that there are no significant differences in the calculation of optimization loss by Gatys et al. [3] and perceptual loss by Johnson et al. [4]. The primary difference in this area, therefore, is in how Johnson et al. [4] structure the loss function so that a neural network can learn to apply a specific style to an arbitrary input image.

Such a network, once trained, stylizes images with a feed-forward process, and so can produce output one thousand times faster than Gatys et al. However, they are limited in that each network can only apply one specific style to an arbitrary content image. In order to learn another style, an entirely new network must be trained.

2.2.3 Demystifying Neural Style Transfer

Li et al. [33] describe why NST is able to produce such visually pleasing images, positing that it is in fact a domain adaptation problem. They explain in precise, mathematical language the process by which the style loss is minimized.

Instead of the traditional style optimization equation used in previous works [3, 4], they design style loss using the Maximum Mean Discrepancy (MMD) metric introduced by Gret-

ton et al. [34]. MMD is a popular metric for measuring the difference between two distributions, and so provides a firmer theoretical basis for NST.

2.2.4 Neural Style Transfer via Meta Networks

A meta-network is a neural network that outputs another neural network. Shen et al. [35] designed and trained a meta-network to accept a style image and produce an output network. The output network takes an input image and produces a stylized version of that image.

In terms of functionality, the output networks are similar to those produced by Johnson et al. [4], yet they are much quicker to produce because they require no training if the meta-network already exists. These networks are also more space-efficient than those by Johnson et al. The output networks exist independently of the meta-network, and so can be deployed elsewhere.

2.2.5 A Learned Representation for Artistic Style

The key limitation of Johnson et al. [4] is how each network can only apply a single style. This prompted research into methods to expand the artistic range of NST while preserving the speed of feed-forward networks. Dumoulin et al. [36] addressed this problem by developing a new normalization technique called conditional instance normalization (CIN).

CIN is based on instance normalization. Rather than performing normalization based on individual content images in a batch, CIN performs normalization over every member of a set of N style images. The network is structured to create N outputs for every input, one for each style on which it was trained, allowing a single network to apply a broad range of

styles. This is faster and simpler than training one network for each style.

2.2.6 Arbitrary Style Transfer in Real-Time with Adaptive Instance Normalization

Huang and Belongie [11] designed an NST algorithm that lets the user select an arbitrary content image and an arbitrary style image, producing a stylized output with a quick feed-forward process. This algorithm uses a novel normalization layer called Adaptive Instance Normalization (AdaIN). Rather than learning parameters during training based on features of the content or style images, AdaIN computes the normalization parameters directly from the features of the style image, then uses them to normalize the features of the content image. This transformed image is then decoded by another network to create stylized output.

2.2.7 A Real-time, Arbitrary Neural Artistic Stylization Network

Shortly after Huang et al. [11] was introduced, Ghiasi et al. [1] published their approach, which uses a "style prediction network" that takes a given style and transforms it to a point in an embedding space that can then be used by a "style transfer network" which performs the actual stylization procedure. This method can also take any pair of input and style images and produce stylized output with a quick feed-forward process.

Ghiasi et al. claim that their work is superior to the work by Huang et al. [11] because their loss is lower given a significance test. However, a model can have a higher loss and still be superior. We perceive that in some of the examples they include in their appendix, Ghiasi et al. [1] slightly overfits the NST problem compared to Huang et al. [11]. This judgement is subjective, of course.

2.2.8 Combining Markov Random Fields and Convolutional Neural Networks for Image Synthesis

A common criticism of Gatys et al. [3] and other NST algorithms is that they prioritize replacing the colors of an image over making significant stylistic adjustments to its content. In other words, instead of repainting a photograph in the style of Picasso, NST repaints a photograph using only the same color palette without introducing any characteristics associated with cubism.

Li et al. [37] use Markov random fields (MRFs) to encourage their network to transform a given content image to incorporate these stylistic patterns. They use a different process than Gatys et al. to suit this purpose. Their approach more effectively incorporates style features and performs especially well at photograph synthesis. Still, it has limits—for example, it has little sense of which details of a given input image to change and which to keep the same, and so sometimes changes more than is visually pleasing.

2.2.9 Arbitrary Style Transfer with Deep Feature Reshuffle

The field of NST can be divided into two categories, according to Gu et al. [38]: *parametric* algorithms similar to those by Gatys et al. [3] and *non-parametric* algorithms similar to those by Li et al. [37]. The principal difference between them is that the former prioritizes the overall stylistic form of the output image and ignores opportunities to stylistically adjust the content, replacing only the "low-hanging fruit," e.g. the colors and textures of the scene. The latter makes more daring adjustments but sometimes needlessly stylizes or shuffles content features.

Gu et al. propose two novel loss functions, the local style loss and the reshuffle loss. In combination, they are designed to capture the strengths of both methods while avoiding their drawbacks. By integrating these loss functions into a new NST algorithm, they demonstrate that their approach is able to reproduce features from the style image in its output more effectively than other methods. At present, however, their method is sensitive to various edge cases and requires careful hyperparameter tuning.

2.2.10 Depth-Aware Neural Style Transfer

Another drawback of many NST algorithms is their reliance on VGG-16 or VGG-19 [39]. These networks are excellent at extracting features for image classification and so are commonly used to train NST networks, which rely on an external source of feature extraction, a process explained further in Section 3.1.1.

However, Liu and Lai [40] note that using VGG for this purpose carries an implicit assumption. Because VGG is designed for image recognition, the features it considers are generally in the foreground, and so other important features—especially depth—are not prioritized by the stylization algorithm. To counteract this effect, Liu and Lai introduce an additional constraint, depth loss, which encourages the algorithm to preserve these structures in its stylized output.

2.2.11 Artistic Style Transfer for Videos

Shortly after NST was introduced by Gatys et al. [3], Ruder et al. worked to apply the same concepts to video, accounting for the unique challenges that working on video provides.

The naïve approach to NST for video—for example, applying the algorithm from Johnson et al. [4] to every frame—results in flickering artifacts because the network is provided no information about how to stylize adjacent frames consistently.

Ruder et al. designed several temporal consistency constraints that account for the shared information between nearby frames, allowing the optimizer to account for both short- and long-term changes in the scene. These constraints, described in more detail in Section 3.2, take into account optical flow data to model which areas of a given frame must be kept the same and which must be re-stylized from scratch. Of course, because their implementation was based on that by Gatys et al. [3], it took several minutes to stylize a single frame, let alone the hundreds or thousands of frames comprising even a very short video.

2.2.12 Real-Time Neural Style Transfer for Videos

Since using the original work by Gatys et al. [3] is too slow, Johnson et al. [4] is a clear alternative. Huang et al. [41][1] developed a neural video stylization algorithm that trades flexibility for speed. It uses the temporal loss from [42] and is designed to use optical flow while training to learn how to accommodate this constraint. After training is finished, it uses only information contained in adjacent frames to perform its stylistic inference. While the videos produced are clearly high quality, the authors acknowledge that their videos have more error than videos produced by Ruder et al. [42].

[1] The supplemental video for [41] can be found at https://www.youtube.com/watch?v=BcflKNzO31A

2.2.13 Artistic Style Transfer for Videos and Spherical Images

Ruder et al. [12][2] also adjusted their algorithm to use Johnson et al. [4], and in contrast to Huang et al. [41] their algorithm depends on optical flow data even after training is complete.

Ruder et al. use the loss from Johnson et al. [4] in addition to temporal loss. They train a neural network to solve an optimization problem based on the perceptual and temporal loss functions. This network is given the current input image along with two additional images that help preserve consistency between stylized frames. A detailed explanation of their method is provided in Section 3.2. This approach produces output equivalent to their previous work [42] in only a fraction of the time.

Despite this speed advantage of this method, however, its additional constraints coupled with the necessity of pre-computing optical flow make it computationally burdensome, especially when GPU acceleration is unavailable.

2.2.14 Video Motion Stylization by 2D Rigidification

There are different philosophies about the desirable qualities of a stylized video. Delanoy et al. [43][3] believe that videos stylized with paintings ought to mimic hand-drawn, two-dimensional animation. In pursuit of this goal, they designed a method of intelligently coarsening the optical flow of a video via human-guided motion segmentation. This novel "rigidification" of the flow field creates a distortion effect. The major features of the video appear to be flattened pieces of cardboard that are moved, stretched, and folded on top of each other. This creates output videos with simpler motion patterns that are more reminis-

[2]The supplemental video for [12] can be found at https://www.youtube.com/watch?v=2C3sxtnxpRE

[3]The supplemental video for [43] can be found here.

cent of classic animation than other methods.

2.3 Chapter Summary

All NST algorithms use convolutional neural networks (CNNs) to extract important information from images. Many NST algorihtms improve their stability and capacity through the use of batch normalization, instance normalization, or their own custom type of normalization. The current state-of-the-art algorithms in NST for video require dense optical flow data, which is an estimate of how pixels in one frame become pixels in the following frame.

In 2015, Gatys et al. [3] introduced NST for images. Because their algorithm optimizes on an image-by-image basis, it has poor throughput compared to more recent methods. Shortly afterward, Johnson et al. [4] trained a neural network that can apply a given style to an arbitrary input image with a feed-forward process. This algorithm sacrifices flexibility for speed. While it is much faster than Gatys et al. [3], it must train a new network in order to apply a new style.

A common naïve approach to NST for video is applying [4] to every frame of a video individually. This approach is fast, yet results in flickering artifacts. Ruder et al. [12, 42] address this by introducing temporal consistency loss, which encourages their network to give adjacent frames similar stlyizations. In addition to this loss, they guide the network through the use of dense optical flow, which informs the network which regions of a given frame must be the equivalent to its predecessor and which ones must be altered.

2.3.1 Limitations of Current Approach

Current algorithms in NST for video are limited primarily in the trade-off between speed and quality. The naïve stylization algorithm is fast, but ineffective. Fast Artistic Videos [12] establishes the best balance between speed and quality. It uses feed-forward networks to achieve speeds comparable to the naïve approach. However, its runtime is long. Even when given an ideal hardware and software setup, Ruder et al. [12] takes 0.4 to stylize a frame. At this rate, a 5-minute video at 30 frames per second will still take an hour to stylize.

Chapter 3

A Method for Distributing Neural Video Stylization

In this chapter, we introduce the theory behind NST. We explain in detail how style is represented mathematically and the models and methods used to produce stylized output. We describe the two essential loss functions, *content loss* and *style loss*, and how they combine to form *perceptual loss*. After that, we introduce Fast Artistic Videos (FAV) and its novel constraints, which form the basis of our algorithm, Distributed Artistic Videos (DAV). Finally, we describe how DAV divides the work of stylization across a cluster and suggest possible improvements to the algorithm that we identified during testing.

Below, x_i refers to a given frame at position i of the input video sequence, y_i refers to the stylized frame at position i of the output video sequence, and s refers to the style image the network was trained to apply. All networks used in DAV are pre-trained.

3.1 Neural Content and Neural Style

NST forms the basis of its operation on a mathematical interpretation of artistic style. Two objectives present in every NST algorithm are to minimize the content loss $\mathcal{L}_{content}$ and the style loss $\mathcal{L}_{style}$. Both losses depend on a pre-trained VGG network of the kind introduced by Simonyan and Zisserman in [39]. We use these losses to measure the similarity of FAV

and DAV in Section 4.1.

3.1.1 Using VGG to Measure Loss

The networks designed by the Visual Geometry Group, often simply called the VGG networks, are expensive to use. In Torch7, a pre-trained VGG network can run upwards of 2GB of data, and in pyTorch around 566MB, all of which must be loaded into memory at train time in addition to the rest of the program. The evaluation criteria of NST involve passing images through VGG, causing further slowdown. To understand why VGG is necessary, we must understand the problem it solves.

Consider a scenario where we wish to calculate $\mathcal{L}_{content}$ given x_i and y_i. $\mathcal{L}_{content}$ and $\mathcal{L}_{style}$ both use mean-squared-error (MSE) in their loss functions; ignore $\mathcal{L}_{style}$ for now. Here, $|x|$ is the size of the image, i.e. the product of its dimensions; assume that $|x| = |y|$.

$$\mathcal{L}(x, y) = \frac{1}{|x|} \sum_{i}^{|x|} (x_i - y_i)^2 \tag{3.1}$$

For unstructured data, MSE can be extremely useful and efficient at assessing total deviation. However, images are highly structured data. When used on images, MSE is sensitive to small, granular changes that are imperceptible to the human eye. Because the purpose of NST is to design images based on artistic features humans find visually pleasing, MSE is only one component in the loss functions we use.

VGG is designed to have good performance on a benchmark dataset called *ImageNet* [44, 45], and so its final "predictions" are useless for NST. In order to make predictions, though, it must first extract features from images. NST algorithms use this feature-extraction

process to support $\mathcal{L}_{content}$ and $\mathcal{L}_{style}$.

Consider, then, running both x_i and y_i through VGG and using specific post-activation layers of VGG to compare x_i and y_i. Our loss function is now the MSE between VGG's respective interpretations of x_i and y_i.

For both $\mathcal{L}_{content}$ and $\mathcal{L}_{style}$, the exact pixel-by-pixel details of x_i and y_i are less important than the features that VGG—and, in some sense, that humans—can see. For $\mathcal{L}_{content}$, this means preserving the high-level features of x_i, for example, the edges of a face, in y_i. For $\mathcal{L}_{style}$, it means something slightly different, discussed in Section 3.1.3.

VGG comes in several varieties, and the type used for NST varies slightly between papers; some use VGG-16 [4,35,36], others use VGG-19 [3,11,37,41,42], others use both VGG-16 and VGG-19 in different experiments [12], and still others fail to specify which VGG network they used [1]. We see no theoretical benefits in using one version of VGG over an other for this problem, and so perhaps the choice of network in any particular algorithm is for historical or practical reasons.

There are other examples of subtle differences between implementations, noted in the relevant portions of the following sections. For DAV, we have made every effort replicate the implementation decisions taken by [12] in FAV. We suspect that these discrepancies play a role in why all of these algorithms have subtly different patterns of output.

3.1.2 Content Loss

Theoretically, several VGG layers can be used to calculate $\mathcal{L}_{content}$. In practice, only one content layer is commonly used. In our implementation, we used the layer of VGG-16

called *relu3_3*, i.e. $V_{content} = [relu3\_3]$. In either case, we run x_i and y_i through VGG-16 and extract features from the content layers for the purposes of comparison.

We apply the following equation to calculate $\mathcal{L}_{content}$, where $l(x)$ is the post-activation features of layer l given an input x and where $|l|$ is the size of the layer, i.e. the product of its dimensions.

$$\mathcal{L}_{content}(x, y) = \sum_{l \in V_{content}} \left(\frac{1}{|l|} \sum_{i=0}^{|l|} \Big[l(x)_i - l(y)_i \Big]^2 \right) \tag{3.2}$$

The exact distance metric used in calculating $\mathcal{L}_{content}$ varies. Some use the MSE method shown in Equation 3.2 [12, 42], some use squared error loss [3], and others use the Euclidean distance [1, 4, 36].

3.1.3 Style Loss

The objective of $\mathcal{L}_{style}$ is to encourage the network to search for patterns and colors in s that it can apply to y_i to minimize the MSE between the *Gram matrices* of $l(s)$ and $l(y_i)$. Below is a definition of the Gram matrix calculation used in FAV and DAV, where l is a layer of VGG, $l(x)$ is a post-activation feature matrix flattened into 2 dimensions, and $|l(x)|$ is its size, i.e. the product of its dimensions.

$$G(l, x) = \frac{(l(x) \cdot l(x)^T)}{|l(x)|} \tag{3.3}$$

Note that $(x^T \cdot x)$ is also a suitable numerator; the specific implementation depends on how the dimensions of $l(x)$ are arranged. A correctly-computed Gram matrix is relatively low-dimensional.

In measuring the difference between the Gram matrices of $l(y)$ and $l(s)$, we reward the network for reproducing the same pattern of features in y that are in s and avoid penalizing it for not reproducing those effects in exactly the same places or exactly the same shapes.

Our definition of $\mathcal{L}_{style}$ is below. Different from $\mathcal{L}_{content}$, this time we use several layers from VGG-16: $V_{style} = [relu1\_2, relu2\_2, relu3\_3, relu4\_3]$.

$$\mathcal{L}_{style}(s, y) = \sum_{l \in V_{style}} \left(\frac{1}{|G(l,s)|} \sum_{i=0}^{|G(l,s)|} \left[G(l,s)_i - G(l,y)_i \right]^2 \right) \tag{3.4}$$

There are two common approaches to calculating $\mathcal{L}_{style}$. The first is to use the MSE method of Equation 3.4 [3, 12, 42]. The second is to use the squared Frobeneus norm of $G(l,s) - G(l,y)$ [1, 4, 36].

3.1.4 Perceptual Loss and Layer Weights

After the content and style losses for a given x_i, y_i, and s are computed, tallying the perceptual loss is a simple endeavor. α and β are weights chosen by the user. A higher α value encourages the network prioritize reducing $\mathcal{L}_{content}$, and a higher β value encourages the network to prioritize reducing $\mathcal{L}_{style}$. Because $\mathcal{L}_{content}$ tends to be greater in magnitude than $\mathcal{L}_{style}$, the β value is commonly increased so that both losses are roughly equal.

$$\mathcal{L}_{perceptual}(x_i, s, y_i) = \alpha \mathcal{L}_{content}(x_i, y_i) + \beta \mathcal{L}_{style}(s, y_i) \tag{3.5}$$

$\mathcal{L}_{content}$ and $\mathcal{L}_{style}$ operate on a collection of layers and so accommodate individually weighing the loss of each layer [3]. In FAV and DAV, all layers used to measure loss are weighted 1, and so we omitted layer weighting from our notation for simplicity.

3.2 Fast Artistic Videos

FAV tackles the challenge of incorporating information shared by nearby frames in order to enforce temporal consistency. To eliminate the flickering artifacts present in the naïve methods, the stylization process uses optical flow information to define $\mathcal{L}_{temporal}$, which is then combined with $\mathcal{L}_{perceptual}$ to form $\mathcal{L}_{video}$, the final loss function used to evaluate the performance of FAV and DAV.

In order to avoid restating Ruder et al. [12] in entirety, we will briefly summarize only the elements of FAV that are essential to understanding our work. Constraints of FAV's multi-frame training are not enforced at test-time, and because training a new model is outside our scope, we will not describe any details thereof.

To stylize x_0 and create y_0, FAV and DAV are equivalent to Johnson et al. [4]—the first frame of a video is stylized independent of the rest of the sequence because no prior information exists. The general case for $i \geq 1$ used by FAV is described in the next section.

3.2.1 Frame Initialization

In general, we wish to recreate the same pattern of stylization in y_i that was in y_{i-1}. This means keeping the brush-strokes, the blobs of color, and the shading the same in both frames. In static videos, this can be accomplished by copying y_{i-1} to y_i. In videos with motion, however, we must account for the optical flow of objects in the scene.

Optical flow can be calculated several different ways, and the exact method used is an implementation detail not specific to FAV or DAV. The presence of dense, accurate optical flow files describing movement between x_{i-1} and x_i is henceforth assumed; f_i is the

forward optical flow estimating how x_{i-1} could become x_i, and b_i is the backward optical flow estimating how x_i could have come from x_{i-1}.

Optical flow files are inaccurate in certain circumstances. Before applying them, FAV measures which regions of the files are valid and which are not.

3.2.2 Detecting Occlusions

Optical flow is only useful if it is accurate, and even the most precise flow-computing algorithms stumble when objects emerge from behind others, occlude others, or exit the frame, among other scenarios. In order to account for these, FAV computes a consistency check c_i between f_i and b_i by applying a method for detecting occlusions and motion boundaries that was originally described by Sundarm et al. [46]. In areas where their estimates are valid, f_i is roughly the opposite of b_i. Large disagreements between the flow files denote areas of inaccuracy that must be ignored.

c_i is a ones-tensor masked by the inequality in Equation 3.7, where $f_i(x)$ and $b_i(x)$ denote an image x being warped by the forward and backward optical flows, respectively. w_i denotes an object created by warping an image with the backward optical flow, adding its original version, and finally warping the resulting image by the forward optical flow. Every file is normalized between 0 and 1.

$$w_i = f_i(x_i + b_i(x_i)) \tag{3.6}$$

$$c_i = |w_i + b_i|^2 > 0.01 * (|w_i|^2 + |b_i|^2) + 0.5 \tag{3.7}$$

c_i denotes which areas of x_i are inconsistent and must be stylized again. For example, we re-stylize regions that were occluded in the previous frame. While Ruder et al. [12, 42] and Sundarm et al. [46] explicitly state that motion boundaries are inconsistent regions, the *implementation* of FAV considers them consistent regions. If we choose not to mask out motion boundaries, c_i will have more regions that are the same between frames. Because DAV follows from the implementation of FAV, it ignores the constraint on motion boundaries.

Pixels in the warped versions of x_i that land outside the bounds of the original image, either by f_i or b_i, are also masked out by necessity.

Once c_i is calculated, f_i has no further use in DAV, though certain configurations of FAV or other algorithms use it to compute stylization from the end of the video to the beginning, or for other purposes.

3.2.3 Temporal and Video Loss

Now that the consistency checks are available, we can define $\mathcal{L}_{temporal}$. In this equation, sequential order is important, and so we include the index i of each file and abstract MSE into its own function. $\odot$ denotes the Hadamard, or element-wise, product.

$$\mathcal{L}_{temporal}(y_i, y_{i-1}, b_i, c_i) = MSE\big(c_i \odot b_i(y_{i-1}), c_i \odot y_i\big) \tag{3.8}$$

This leads to the final loss formula, $\mathcal{L}_{video}$. $\mathcal{L}_{perceptual}$ is expanded here for the sake of clarity, and α, β, and γ are weights chosen by the user.

$$\mathcal{L}_{video}(s, x_i, y_i, y_{i-1}, b_i, c_i) = \alpha\mathcal{L}_{content}(x_i, y_i) + \beta\mathcal{L}_{style}(s, y_i) + \gamma\mathcal{L}_{temporal}(y_i, y_{i-1}, b_i, c_i) \tag{3.9}$$

We explain how this equation is used to measure the performance of FAV and DAV in Section 4.1.

3.2.4 Network Architecture

See Figure 3.1 for a diagram of the neural architecture of FAV. Section 3.3 explains how to compute the input tensor; for now, know that it comprises two Blue-Green-Red (BGR) images and one grayscale image. Each network is trained to apply only one style, and so the essence of that style is encoded into its weights. Every convolutional layer save the last is followed by an instance normalization layer of the kind proposed by Ulyanov et al. [31]. The first 3 convolutional layers expand the number of channels from 7 to 32, 32 to 64, and lastly 64 to 128. The process of convolution encodes the input frame into abstract feature space.

Once the tensor is in its expanded state, it goes through 5 residual blocks. The residual blocks are responsible for performing the stylistic adjustments in feature space and allow the network to learn a residual mapping between the input and the desirable stylized output according to the objective functions. This is an application of the ResNet technique introduced by He et al. [47]; more specifically, it follows the residual architecture originally described by Gross et al. [48].

After that, its non-channel dimensions are scaled to double their original size, and its channels are reduced from 128 to 64 by a deconvolutional layer. After more upscaling, the channels are finally reduced to 3. The process of deconvolution is, in this context, similar to decoding from abstract feature space back to image space.

The final output is multiplied by 150 before being returned. The final layer in the

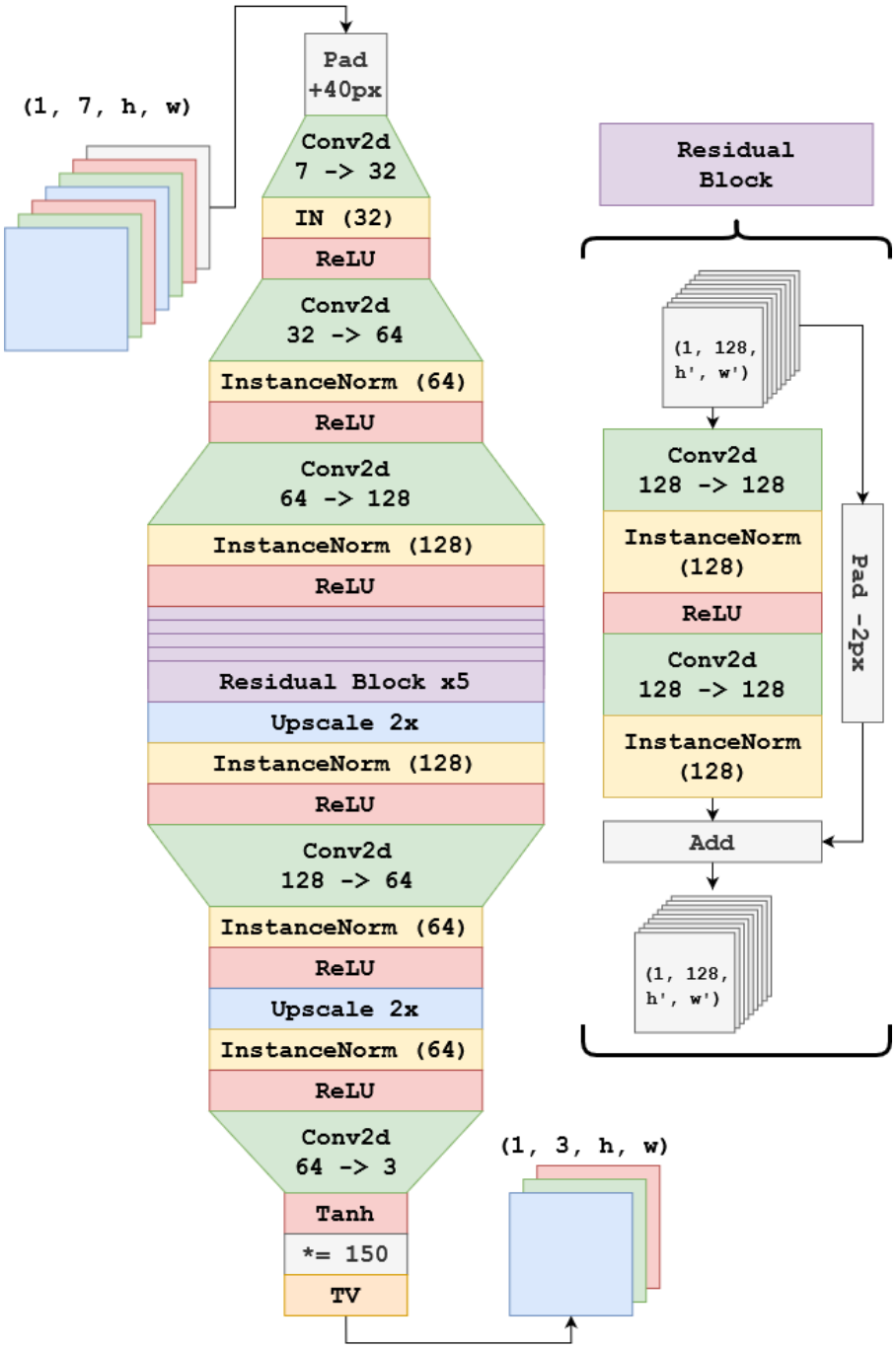

Figure 3.1: A diagram of the neural network architecture used by FAV and DAV. The last three layers are a bit different: We apply the `tanh()` activation function, multiply the final tensor by exactly 150, and then measure Total Variation (TV) regularization, if applicable.

network is called *Total Variation* (TV). It is used to regularize FAV, and is included here for completeness, but has no function outside training.

3.3 Distributed Artistic Videos

The main challege of DAV[1] is distributing the work of FAV across an arbitrarily large number of nodes, and doing so with minimal overhead. See Figure 3.3 for an illustration of our algorithm. DAV-N means running DAV on N nodes.

The process of calculating the optical flow files b_i and c_i for every pair of adjacent frames in a video is easily distributed because no set of files depends on the information of any other set. In contrast, stylization depends on every new operation having access to the result of the previous operation. By understanding the full process FAV uses to stylize an image, we identified a way to work around this dependency and distribute stylization. All of the steps and materials required to stylize x_i are described below and illustrated in Figure 3.2.

1. Preprocess x_i to produce x'_i. Preprocessing is converting the image into a Torch tensor and subtracting the VGG Mean.

2. Warp y_{i-1} according to b_i and apply the same preprocessing in the previous step to produce y'_{i-1}.

3. Erode c_i according to a minpooling operation, producing c'_i.

4. Concatenate $[x'_i, y'_{i-1} \odot c'_i, c'_i]$ into a tensor, in that order. If stylizing the first frame of

[1]Our implementation can be found on GitHub at https://github.com/pgalatic/thesis/tree/repro

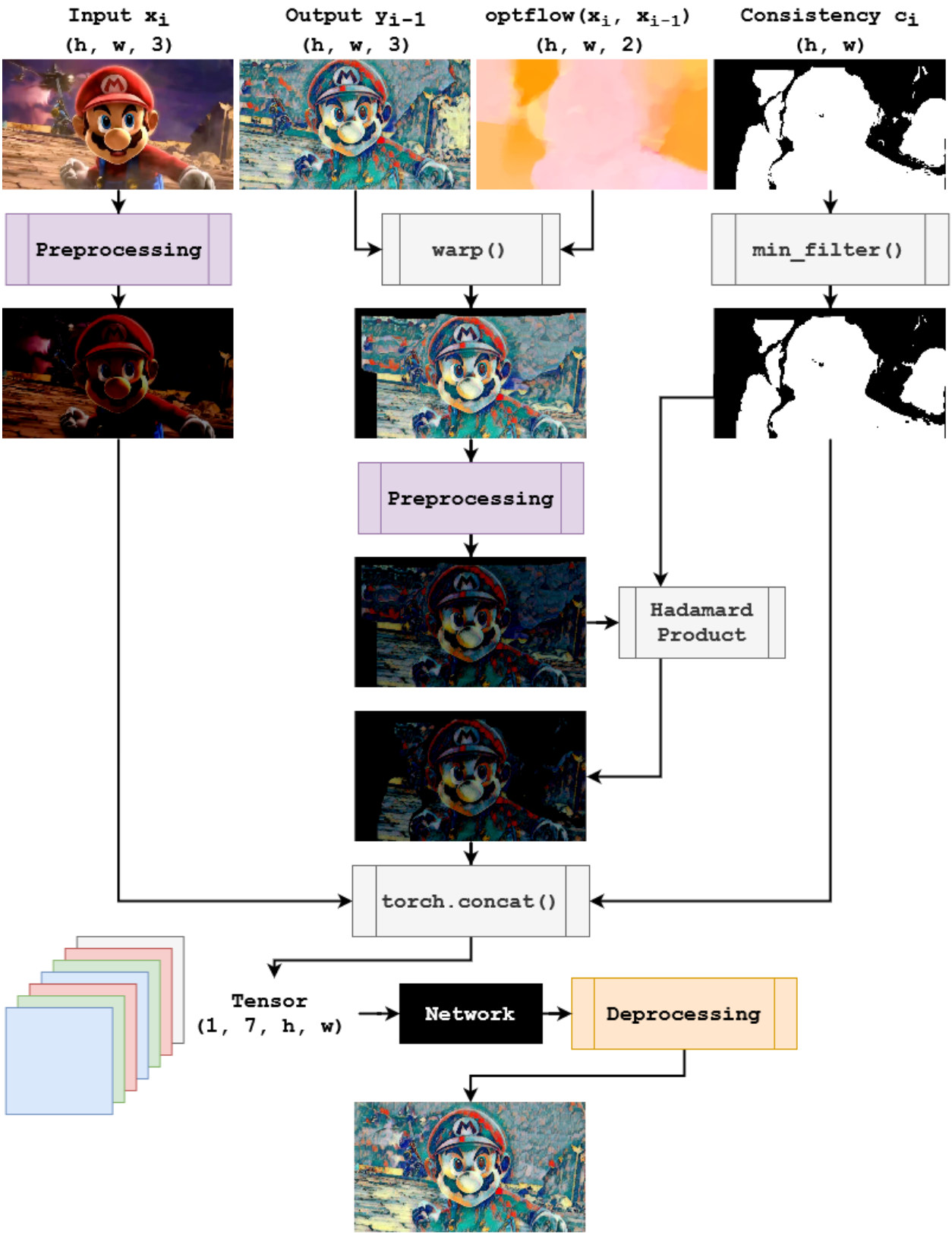

Figure 3.2: A flowchart representing the stylization process of FAV and DAV. The original frames are from https://www.youtube.com/watch?v=cjdfqXIM-Ko

a video, y'_{i-1} and c'_i are substituted with equivalently-shaped zero tensors. Run this new tensor through the network.

5. Deprocess the output (the inverse of preprocessing) and save the resulting image.

FAV has the implicit assumption that all adjacent frames are related to each other, which is not always true. Many videos have cuts, i.e. a form of scene transition where two adjacent frames carry no important information between them. Once identified, these cuts allow the the video to be split into sub-videos that can be stylized independently of one another and later reassembled with no loss of visual fidelity. Thus, videos that have cuts can benefit from even further distribution of labor.

A characteristic of FAV not addressed by DAV is its reliance on pre-trained models. Each model can only apply a specific style to a video, e.g. Picasso's *Self-Portrait, 1907*. The process of training a model is nontrivial and is not replicated in DAV. Instead, the six models from [12] have been converted from Torch to pyTorch. This was done using a modified version of Convert Torch to pyTorch [49]. The equivalence of these converted models, and the equivalence of FAV and DAV more broadly, is demonstrated in Section 5.1.1.

The most critical portions of FAV have also been converted to Python and pyTorch. This migration alone greatly improved its speed, and distributing the newly flexible algorithm has achieved even greater returns, described in Section 5.1.3.

DAV assumes that all computers in a cluster share a file system. The implementation of this file system has a significant impact on performance; see section 3.3.1.3. DAV uses this shared directory to store placeholder files that denote which tasks have been claimed

by which nodes. The standard input to DAV is the path to the shared directory, the path to the target video, and the path to the pyTorch stylization model. The output is a stylized version of the input video, which is placed in the shared directory.

Claiming a task involves creating a file in this directory with a name unique to that task. If two nodes attempt to claim the same task, DAV is designed so that one of them will fail to create the file, resolving the conflict. Depending on the circumstances, the node or nodes who failed to claim the task will either wait until it has finished or proceed to the next available task. No messages are directly exchanged between nodes. Tasks are served on a first-come, first-serve basis.

DAV has four phases. The first and the last are serial components that cannot be effectively distributed, yet because they are so fast, the impact they have on computation time is insignificant. The two in the middle are slow, but can be distributed to great effect. See Figure 3.3 for an illustration.

3.3.0.1 Phase 1: Sync and Split

First, one node splits the video into a collection of frames, storing them in the main directory. This phase also calculates the partitions used in Phase 3. Partitions are chunks of a video defined by its cuts, and can be specified manually or determined automatically via pySceneDetect [50].

When computing cuts automatically, we use the Content Detector from pySceneDetect at a threshold of 45. This detector looks for rapid transitions between the colors of adjacent frames. Automatic detection performs admirably, but not perfectly. It is particularly sen-

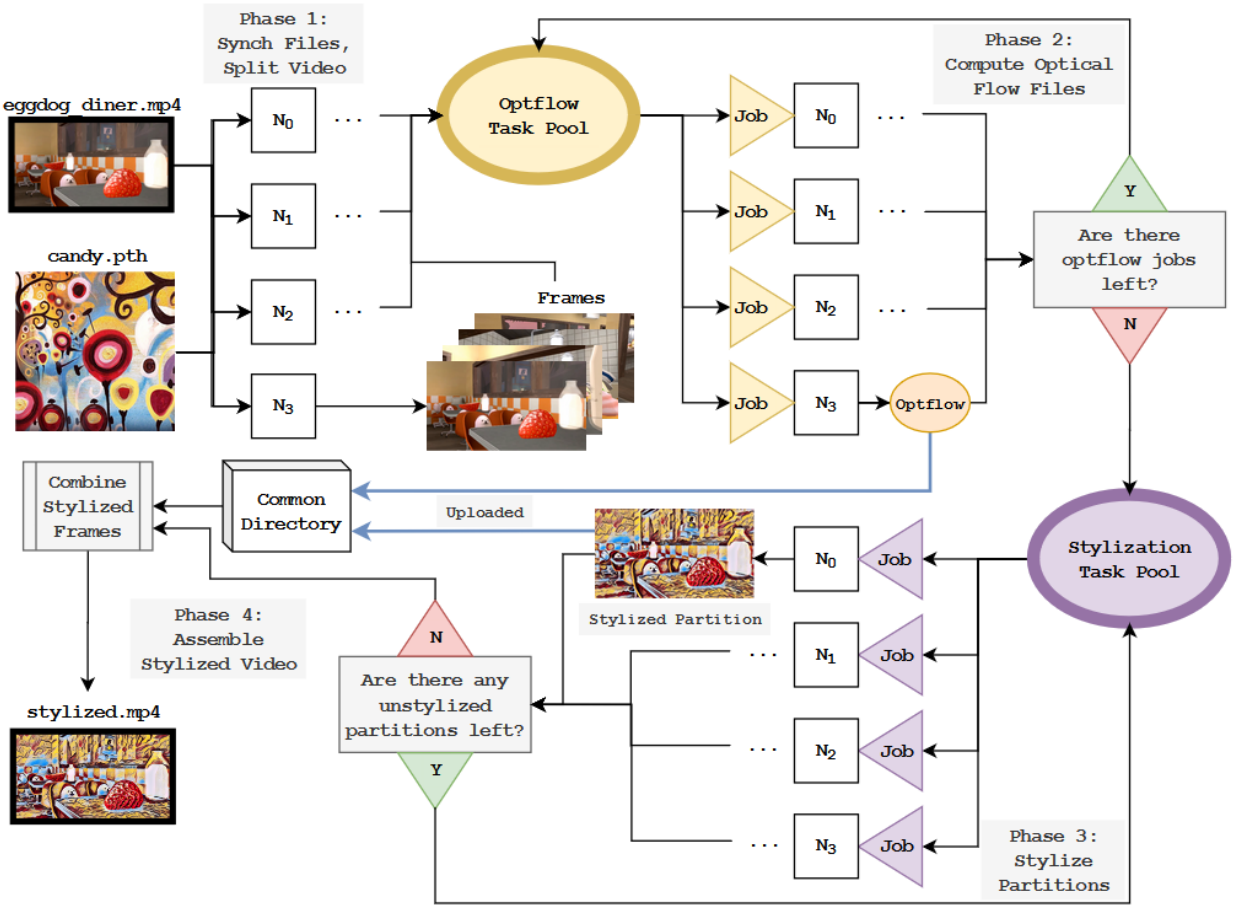

Figure 3.3: A depiction of the distributed algorithm over a cluster of four nodes. It is, of course, easily extended to an arbitrarily large number of nodes. The "Optflow Task Pool" and "Stylization Task Pool" are operations on the same directory in practice (middle-left). They are delineated more specifically here to better illustrate the control flow of the program. The example video is https://www.youtube.com/watch?v=l0Nc0-dFwAI

sitive to bright flashes, e.g. an explosion, or prominent occlusions, e.g. a hand passing in front of a camera. We used the output of the Content Detector to inform a more accurate manual determination of cuts later. All testing was performed with manually-specified cuts.

3.3.0.2 Phase 2: Optical Flow

Once they have access to all the frames, nodes begin calculating optical flow. Optical flow is organized in the form of 'tasks,' where a task is computing the three optical flow files between a pair of consecutive frames. Once a task is claimed, the node that claimed it will compute the optical flow between a pair of adjacent frames, forward and backward, and the consistency check, placing all 3 resulting files in the shared directory. Nodes use threading to claim and execute multiple 'optflow' tasks at the same time.

3.3.0.3 Phase 3: Stylization

After the pre-computation of optical flow is finished, nodes claim partitions the same way they claimed optical flow tasks. When a partition is claimed, the node stylizes it using the method described in Section 3.2. The stylized output frames are then moved to the common directory. While threading can be used to claim and execute multiple stylization jobs at the same time, we found that this caused so much computational strain that it actually decreased the speed of stylization instead of increasing it. So, stylization jobs are completed one by one.

3.3.0.4 Phase 4: Combine and Complete

Once the stylized output is computed, one node combines the frames into the final stylized video.

3.3.1 Planned Improvements to DAV

While testing DAV, we identified several weaknesses that we will fix going forward to improve its performance and capacity.

3.3.1.1 On Scalability

When DAV was first implemented, it generated an enormous amount of intermediary files at runtime. Not only is writing these files to disk a source of slowdown, but storing them can be challenging in and of itself—a two-minute-long high-resolution video can generate over 100GB of data. This flaw drastically limited the scalability of DAV.

Shortly after testing concluded, DAV was adjusted so that optical flow jobs are allocated from within the stylization procedure. That way, optical flow files are "consumed," i.e. deleted, after they are processed by the network. However, this improved procedure is still imperfect. In a scenario where optical flow files are computed and written to disk rapidly, they may overwhelm disk storage before the stylization algorithm has an opportunity to consume them. A better solution is to avoid writing optical flow files to disk at all; for example, by using the Python wrapper of DeepFlow2. In this design, DAV computes optical flow just before it can be used by the stylization procedure and the entire process occurs in memory.

This idea of stylizing images exclusively in memory could also apply to the frames of a video and their corresponding stylized outputs, which are currently written to disk in Phases 1 and 3 respectively. However, this strategy is likely more expensive than writing the files to disk and reading them when they become necessary according to the original algorithm.

These updates, along with the considerations for load balancing according to Section 3.3.1.2, will improve the scalability of DAV. However, its efficiency is fundamentally limited by the inputs it must process. In a video with 7 cuts where the vast majority of frames are in only 1 partition, DAV must wait the same amount of time while the largest partition is stylized whether it has access to 2 nodes or 200.

3.3.1.2 On Load Balancing

Another flaw in the original implementation of DAV is how partitions were allocated in the order they appeared. On edge-cases where a video ends with a particularly long partition, DAV always allocates that partition last, which causes a long period at the end of Phase 3 where only one node is working. This is the primary reason, we believe, that DAV-4 and DAV-8 perform so similarly with respect to video stylization.

All other concerns equal, the most effective load balancing strategy for DAV is to allocate the largest partitions first. After testing concluded, DAV was updated to behave in this way.

3.3.1.3 On the Common Directory

Originally, DAV was designed to place the files it produced in local directories and only move them to the common directory when absolutely necessary. Since our testing environ-

ment used fast LAN connections, this somewhat overbearing strategy was unnecessary for the purpose of evaluating DAV, and so it was scrapped. Because of this, DAV relies on quick access to the common directory to perform well. A slow connection will strangle its throughput.

3.3.1.4 On Neural Networks for Optical Flow

Deepflow2 is used in DAV for two reasons. First, it is easy to implement on CPU-only machines. Second, it is one of the optical flow algorithms used by Ruder et al. [12], and so was necessary to include in order to demonstrate equivalence.

However, without GPU acceleration, computing optical flow using a neural network can be very slow. For the benefit of users who have weak computers or are simply experimenting for fun, DAV now supports using the optical flow calculation algorithm introduced by Farnebäck [26].

We find this method produces acceptable results far faster and with a smaller amount of computational overhead. Furthermore, the visual effects created are interesting in their own right. At times, the background slides past the foreground, or the perspective on an object in the foreground is lost, giving a two-dimensional appearance. These effects are reminiscent of the videos created by Delanoy et al. [43].

3.3.1.5 On GPU Acceleration

GPUs were avoided for this thesis both for reasons of practical necessity and because GPUs are not required to demonstrate how effectively the work of stylization can be dis-

tributed across a cluster.

According to the results by Ruder et al. [12], running FAV on a GPU is faster per-frame than running DAV-8. In practice, we were unable to reproduce this speed despite considerable effort; we described why in Section 1.2. That said, GPU-accelerated nodes are easily integrated into DAV alongside CPU-only nodes. Now that DAV is based on pyTorch, updating it to support GPU acceleration is a simple endeavor.

3.4 Chapter Summary

Similar to other machine learning problems, NST algorithms require a mathematical description of their objective. In NST for images, this takes the form of the loss functions $\mathcal{L}_{content}$ and $\mathcal{L}_{style}$, which combine to form $\mathcal{L}_{perceptual}$. These losses measure the mean-squared error (MSE) between two images, but not directly.

First, each image is run through a pre-trained network designed by the Visual Geometry Group, i.e. a VGG network. Then, specific post-activation layers of the VGG network are taken and their features compared. This strategy encourages the NST algorithm to minimize differences in human-perceptible features rather than differences in granular, pixel-by-pixel details.

$\mathcal{L}_{content}$ compares the features of the input image, sometimes called the content image, to the features of the output image. Ideally, the high-level features, e.g. edges, are kept the same between both the content and output images.

$\mathcal{L}_{style}$ compares the Gram matrix of the features of the style image to the Gram matrix of the features of the output image. It measures the extent to which the pattern of features

in the style image are applied in the output image.

$\mathcal{L}_{perceptual}$ is a weighted sum of $\mathcal{L}_{content}$ and $\mathcal{L}_{style}$. In order to stylize videos, we need one more loss, $\mathcal{L}_{temporal}$, and an understanding of how to apply optical flow.

Given forward and backward optical flow, we can calculate how well they correspond using inequality formulas defined by Sundaram et al. [46]. These occlusion files, when given to the network, help it decide which areas to change and which to keep the same when stylizing a new frame.

The backward optical flow is applied to the previously stylized frame to predict the next frame, and $\mathcal{L}_{temporal}$ measures how well the actual output of the network compares, ignoring regions that are inconsistent according to the occlusion detection procedure. This loss uses MSE directly without first passing either image through VGG.

$\mathcal{L}_{video}$ is a weighted combination of $\mathcal{L}_{content}$, $\mathcal{L}_{style}$, and $\mathcal{L}_{temporal}$. FAV and DAV are compared in Chapter 5 using $\mathcal{L}_{video}$.

DAV distributes the work of FAV by spreading the optical flow and stylization calculations across several nodes. Optical flow is easily distributed, but stylization is more difficult. FAV has an implicit assumption that every two adjacent frames carry important information between them, but this is not always the case. DAV splits videos into sub-videos that are stylized individually and later recombines them with no loss of visual fidelity. Nevertheless, it is not a perfect algorithm, and future versions will correct various issues that we identified during testing.

Chapter 4

Evaluation

DAV must demonstrate two important qualities. The first is functional equivalence to FAV, measured based on the original loss measure from [12]. The second is massive speedup when run on multiple nodes. Evaluation was performed by stylizing videos with DAV and FAV on several differently-sized clusters, sampling different inputs and styles. We measured how the stylized output videos compare to each other in terms of quality, measured qualitatively and quantitatively, and how long they took to produce.

4.1 Equivalence of FAV and DAV

The equivalence of FAV and DAV was measured by taking the composite average loss of the first 15 frames of videos produced by each algorithm given the same style and content inputs. For our experiments, following from details provided in Ruder et al. [12], we used a content weight α of 1 and a temporal weight γ of 50. These hyperparameters are defined in Section 3.1. In Ruder et al. the weight β was given for 5 different styles, and so FAV and DAV were evaluated using these 5 styles. Finally, the composite losses of all trials are averaged and presented to allow for an even easier comparison of FAV and DAV. Image samples are also provided for visual comparison.

For each trial and for the average of all trials, the total loss of each algorithm is shown divided by 10^6, the scale used in Ruder et al. [12], to emphasize their true proximity. This is because the range of the loss function is difficult to communicate without training a model from scratch and comparing convergence patterns. When comparing raw loss values without this context, differences can appear much more significant than they actually are. We also provide frames stylized by each algorithm to aid comparison.

4.2 Dataset Analysis

Videos are naturally heterogeneous, having different lengths, resolutions, and number of cuts. Our dataset has 30 videos, each of 720p resolution. This means that every frame has a height of 720 pixels, while the width can vary, generally between 720 pixels and 1280 pixels. This variation affects performance on individual videos, which is compensated for by using averages in the final results.

The videos in the dataset range from short 'vines' featuring scenes in real life (hundreds of frames) to video game and movie trailers in high fantasy settings (thousands of frames). These videos are used because they best approximate the most likely use case for DAV in the near-term: People stylizing short videos for fun. The sources of the videos we used, along with the code used to perform evaluation, can be found in our reproducibility repository[1].

We wanted to learn more about the distribution of cuts in each dataset and the extent to which partitioning a video by its cuts actually reduces the time it takes to perform stylization. However, the heterogeneity of our dataset makes identifying patterns difficult.

[1] https://github.com/pgalatic/thesis/tree/repro/videos

Each video not only has a different length, but also a different number of cuts. The lowest number of cuts in a video in our dataset is 0, and the largest is 80. Assuming a pair of videos share the same number of frames and the same resolution, the time taken to stylize them will be determined by the number of cuts they have—the more cuts, the more easily the work of stylization is distributed across a cluster.

Beyond that, analysis become hazy. A pair of videos of the same length that both have three cuts will not necessarily take the same amount of time to stylize. Consider a scenario where the first video has all three cuts evenly spread throughout, while the second video has all three cuts in its second half. The first video is more easily distributed than the second because the second will need to wait longer for its longest partition to be stylized.

We used a novel statistic called *standard frames per cut* (SFC). It is defined below, where H is Shannon entropy, V is a video, V_f is its number of frames, V_c is its number of cuts, and V_p is an array of the number of frames in each of its partitions.

$$H(V_p) = -\sum_{i=1}^{n} p(x_i) log_{len(V_p)}(p(x_i)) \tag{4.1}$$

$$SFC(V) = \frac{V_f}{1 + (V_c * H(V_p \odot \frac{1}{V_f})^2)} \tag{4.2}$$

This metric measures the *effective average* number of frames per partition; an average augmented by how helpful a partitioning is to distributing the work. The SFC assumes that every partition is given its own node so that load balancing is not a factor. Videos with no

cuts are defined to have $SFC = V_f$. Videos with perfectly even cuts will have $SFC = \frac{V_f}{V_c+1}$. All others will be somewhere in-between.

We predict that videos with a small SFC will be easier to stylize across a large cluster because the stylization work will be more easily and evenly distributed between nodes.

4.3 Speedup

The speedup of DAV-N over FAV is easily computed by measuring the amount of time each algorithm takes to stylize a video. In order to account for the length of a video, we count the number of seconds it took to stylize divided by the its number of frames. This is called the *seconds per frame* (S/F) ratio. These ratios are then averaged across all trials for FAV, DAV-1, DAV-2, etc. We predict that when DAV is run on larger and larger clusters, it will yield faster and faster speeds.

Due to time constraints, FAV was run on only the shortest 10 videos. DAV-1 and DAV-2 were run on only 18 videos and 22 videos, respectively, prioritizing shorter ones. DAV-4 and DAV-8 were run on the entire 30 video dataset.

Our experiments for DAV measure how long each phase described in Section 3 takes to complete. In FAV, optical flow calculations occur concurrently with stylization, and so we were only able to measure total stylization time in our experiments with FAV.

4.4 Implementation and Setup Details

Cloudlab provided the testing hardware. Because no GPUs were necessary for the completion of this work, all testing was performed on nodes in the Utah facility. Nodes were

initialized with the Ubuntu 16.04 distribution and ran on Python 3.5. We used FFMPEG [51] to split videos into frames and assemble stylized frames back into videos. We used Deepflow2 [28] to calculate optical flow. Our models were run on pyTorch 1.4.0 [17].

4.5 Chapter Summary

The equivalence of FAV and DAV is determined by comparing the average loss of stylized frames they produce. Videos are naturally heterogeneous, and we express a wide variety of possible inputs in our custom dataset, from videos of a few hundred frames to a few thousand frames. Some videos lack cuts, and others have dozens. The dataset is comprised of 30 videos curated for what we believe is an effective distribution of the typical use cases.

The standard frames per cut (SFC) metric is a way of assessing the practical effect of partitioning a video. In a scenario where an arbitrarily large number of nodes is available, videos with no cuts are going to take longer to stylize than videos with cuts. A video that is evenly cut into 4 pieces will be easier to distribute than a video cut unevenly, and the SFC metric measures this effect.

The speedup of DAV over FAV is assessed by measuring the average *seconds per frame* (S/F) ratio. This is simply the total number of seconds it takes to stylize a video divided by the number of frames in that video. We measured the average S/F across all videos stylized by a cluster of a given size in order to compare the speed of FAV to DAV-1, DAV-2, etc.

Chapter 5

Results

In this chapter, we analyze the results of the experiments we described in Chapter 4. Afterward, we discuss our experiences with NST for video and lay out a path for future research.

5.1 Analysis

We first demonstrate that FAV and DAV are functionally equivalent. Next, we examine our dataset using the SFC metric, and lastly we analyze the overall speedup that DAV yields over FAV. The raw data we gained from our experiments can be viewed on GitHub[1]. We also have a supplementary video on YouTube[2].

5.1.1 Equivalence of FAV and DAV

FAV and DAV have nearly the exact same performance even across different models and inputs. The specific loss values for each trial can be found in Table 5.1. At times FAV slightly outperforms DAV, and at times DAV slightly outperforms DAV. Because the loss function has a range up into the millions for an uninitialized network, a difference in loss of

[1] https://github.com/pgalatic/thesis/tree/repro/products
[2] https://www.youtube.com/watch?v=P6OLHvJHQpY

Name/Style	FAV $\mathcal{L}_{content}$	DAV $\mathcal{L}_{content}$	FAV $\mathcal{L}_{style}$	DAV $\mathcal{L}_{style}$
chicken/picasso	28602	30013	58250	51991
dance/picasso	25615	26050	25707	23025
face/scream	11284	12291	13988	21396
floating/candy	25888	25789	17999	18107
jordan/WomanHat	19359	19995	60753	51533
night/mosaic	25317	27157	59419	51947
sonicfan/scream	11920	12506	15181	20835
Name/Style	FAV $\mathcal{L}_{temporal}$	DAV $\mathcal{L}_{temporal}$	FAV Norm. $\mathcal{L}_{video}$	DAV Norm. $\mathcal{L}_{video}$
chicken/picasso	4123	4509	0.091	0.087
dance/picasso	1173	975	0.052	0.050
face/scream	2761	2966	0.028	0.037
floating/candy	7919	7939	0.052	0.052
jordan/WomanHat	3238	3168	0.083	0.075
night/mosaic	14328	13836	0.099	0.093
sonicfan/scream	1699	1742	0.029	0.035

Table 5.1: The loss function by Ruder et al. [12] was reimplemented in pyTorch and used to evaluate 15 frames of videos produced by FAV and DAV. Both algorithms used the same input video and style model and so we expect them to both produce the same output. The final normalized loss is produced by dividing the total loss by 10^6 to convey the range of the loss function.

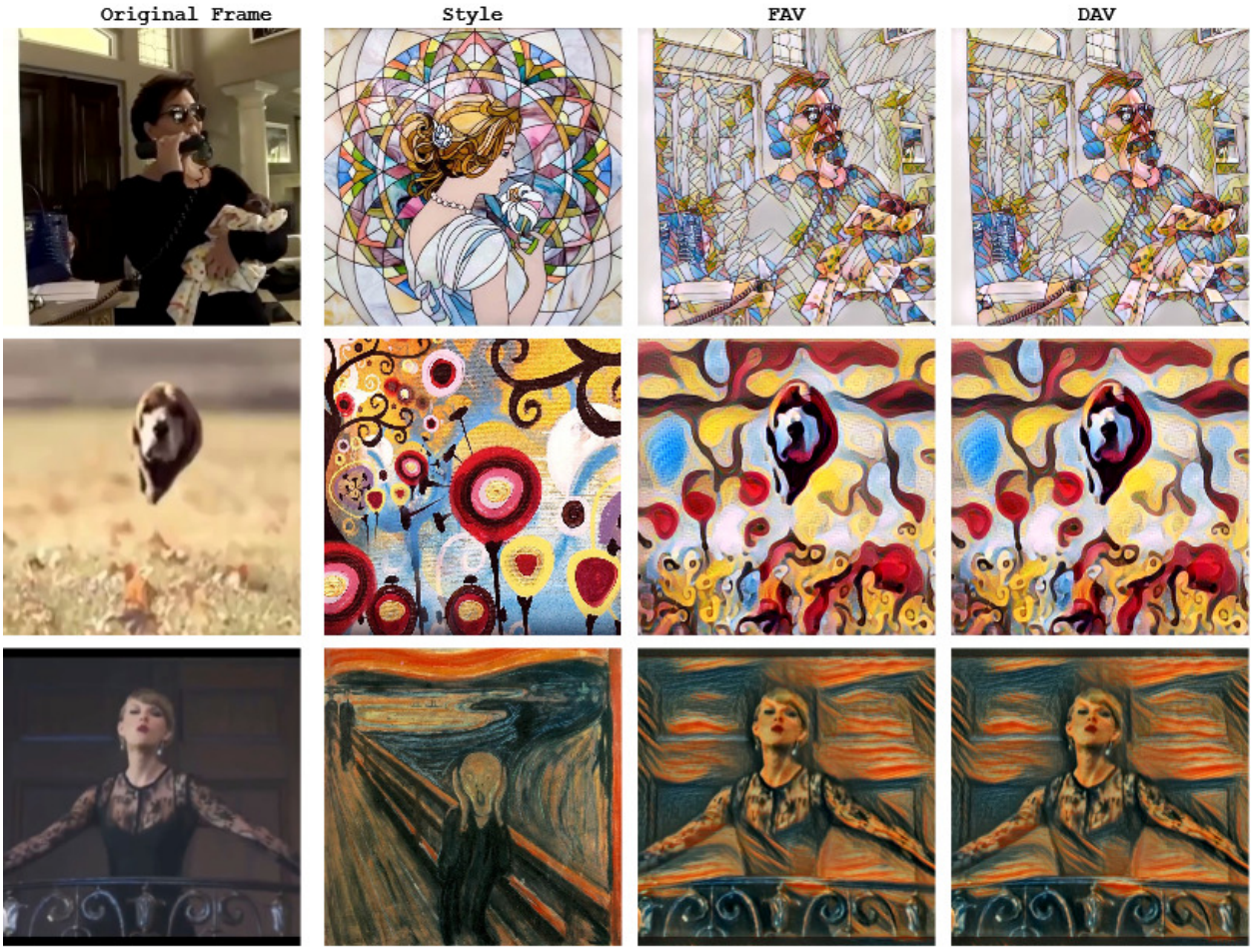

Figure 5.1: Examples of outputs from FAV and DAV given various frames of videos. There are only faint differences between the outputs of FAV and DAV if there are differences at all. Some of the images are square-cropped here for ease of presentation. Each frame is at least ten frames after the start of the video. From top to bottom, the input frames (left) are from `chicken.mp4`, `floating.mp4`, and `face.mp4`, while the styles (center-left) are *Mosaic*, *Candy*, and *The Scream*. `floating.mp4` is of only 360p resolution and so was not used for evaluating speedup.

Overall	FAV	DAV
Average $\mathcal{L}_{content}$	21140	21943
Average $\mathcal{L}_{style}$	35899	34136
Average $\mathcal{L}_{temporal}$	5034	5019
Average Normalized $\mathcal{L}_{video}$	0.062	0.061

Table 5.2: Above is the average composite loss across all equivalence trials. The normalized loss is the total loss divided by 10^6.

a few thousand is inconsequential, which is why we normalized the total loss by 10^6.

The average loss across all the trials is displayed in Table 5.2. The slight variations in loss are acceptable for this domain. Furthermore, the actual images produced are virtually identical; see Figure 5.1.

Some slight differences between the two networks may remain. Any such difference is most likely a discrepancy between the underlying implementations of the libraries used. For example, where Ruder et al. [12] used the Torch7 `image.warp()` function, we used a custom function based on `cv2.remap()` from OpenCV [52]. In this particular example, the two functions seem to have the same effect, but there may be subtle differences elsewhere.

5.1.2 Dataset Analysis

Determining how helpful a cut is to a video depends on an array of factors external to the cut itself. The resolution of the video, the number of nodes performing computation, and the stylization inference used can all have affect the final result. For example, the

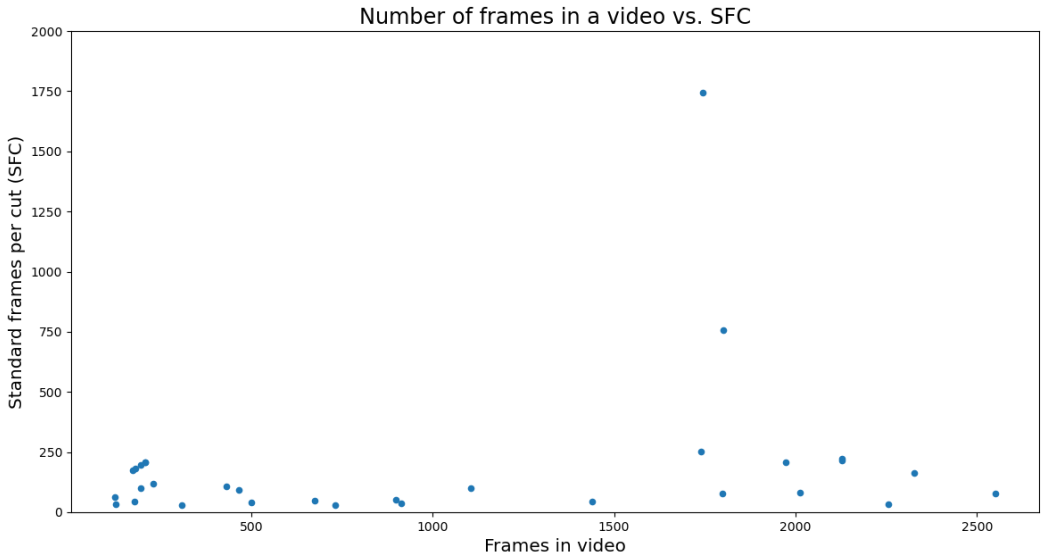

Figure 5.2: A scatterplot of the number of frames in a video versus that video's SFC. The distribution is nearly uniform, except for a couple outliers, which are long videos that have few to no cuts.

impact of partitioning a video dropped drastically when DAV was updated from Torch7 to pyTorch, a phenomenon discussed in more detail in Section 5.1.3. That said, there are two basic observations that can be drawn from examining the dataset. The first is that the SFC of videos seems to follow a roughly uniform distribution; see Figure 5.2. This distribution is rough in part because intentionally heterogeneous videos were chosen to comprise the dataset, including a couple worst-case-scenario examples—long videos with few to no cuts.

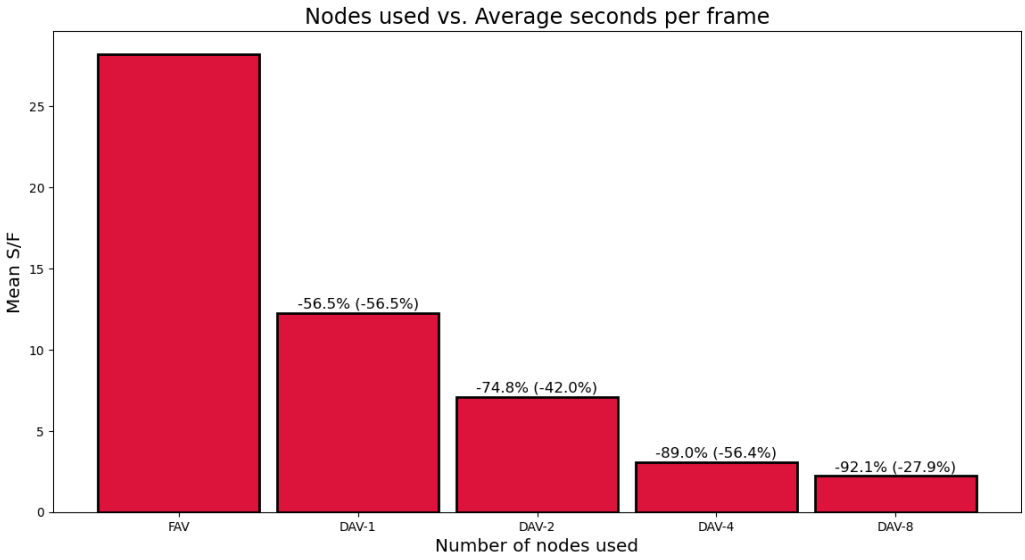

Figure 5.3: The average amount of time required to stylize a video drops dramatically when the number of nodes increases. On top of each bar, the first number refers cumulative speedup compared to the first bar, and the second number refers to the relative speedup when compared to the previous bar.

5.1.3 Speedup

Our overall results can be seen in Figure 5.3. With DAV-4 the time to stylize a video is reduced by 89% on average, and with DAV-8, the time is reduced by 92%. Stylization procedures that used to take days can be completed in hours, and those that used to take hours now take minutes.

While the difference in speedup between DAV-4 and DAV-8 may appear lackluster, a reduction in time by nearly 28% is still significant time-savings, especially considering the

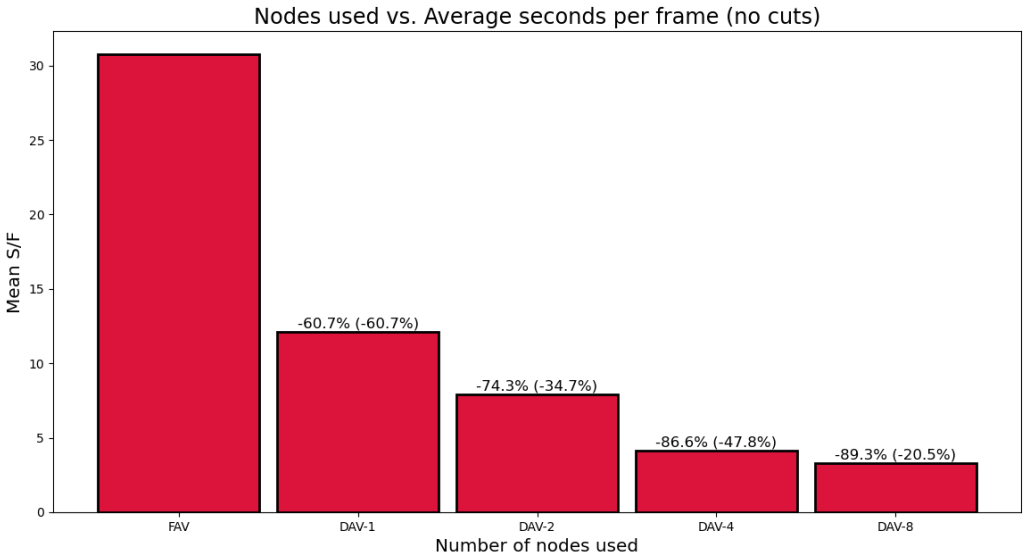

Figure 5.4: There are still significant time savings even in videos that have no cuts.

large portion of the dataset that was unsuitable to distribution, e.g. videos with 0 cuts. The benefits of distribution are potent even for these videos; see Figure 5.4.

Note that FAV is not included all figures because FAV does not measure its time to compute optical flow independently of its time to compute stylization, and so we only measured total processing time when we ran FAV.

Figures 5.5 and 5.6 together demonstrate the intuitive fact that the number of cuts in a video does not impact the relative amount of time it takes to compute optical flow for that video. When the number of nodes is doubled, the time required to calculate optical flow is roughly halved. We believe these relative speedup effects will persist no matter the algorithm used to calculate optical flow.

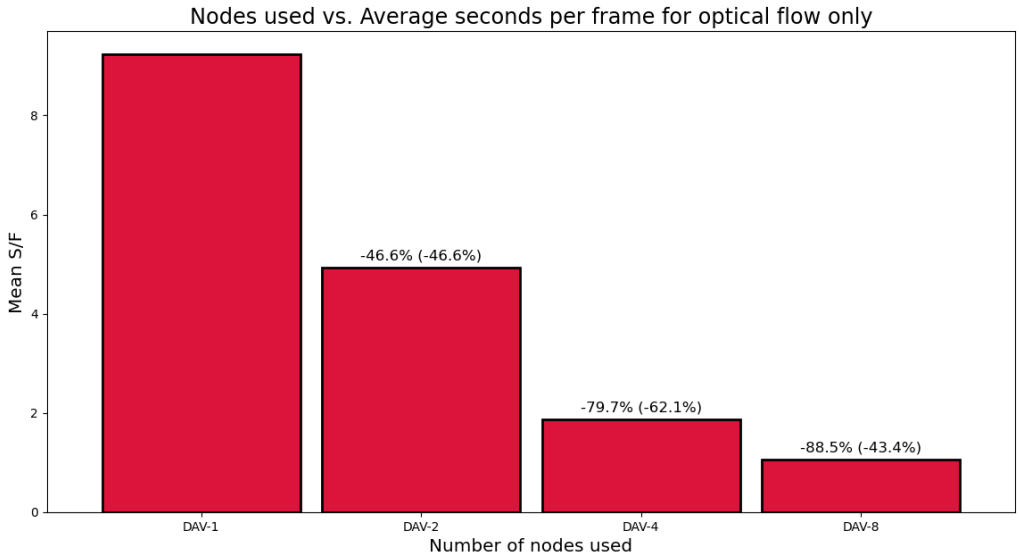

Figure 5.5: Considering only the time taken to compute optical flow files, we observe a roughly linear speedup when the number of nodes in DAV is doubled.

In the original design of DAV, the majority of runtime was spent stylizing videos with Torch7. After updating to pyTorch, calculating all the required optical flow files takes roughly 2 or 3 times longer than stylization. Because the majority of speedup is now achieved by reducing the time taken to compute optical flow, distributing stylization has less impact.

Considering only the amount of time spent performing stylization, the speedup of DAV-4 and DAV-8 over DAV-1 was just under 65%; see Figure 5.7. We had hoped to achieve greater speedup. Of course, our numbers are skewed because distributing stylization over videos with no cuts yields no speedup; see Figure 5.8.

Even among videos that are supposed to benefit greatly from stylization, however, there

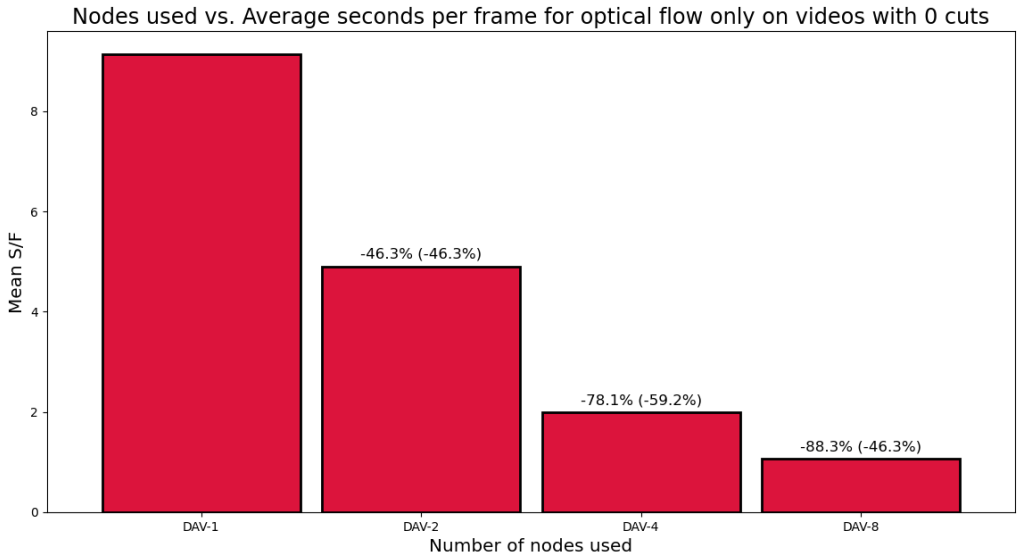

Figure 5.6: Intuitively, flow computation is sped up regardless of now many cuts a video has. This figure is similar to Figure 5.5, but counts only videos that have 0 cuts.

is a sharp diminishing-returns effect from DAV-4 to DAV-8; see Figure 5.9. This is likely due to a lack of load balancing in the implementation used for testing; see Section 3.3.1.2.

5.2 Discussion

During experiments, FAV crashed surprisingly often when trying to load its own occlusion files. These errors halted stylization in its tracks. They did not occur consistently—some runs crashed at frame 50, others at frame 120, and others completed perfectly fine. It seemed to occur only when running the algorithm on Cloudlab. We were unable to determine the root cause of these errors.

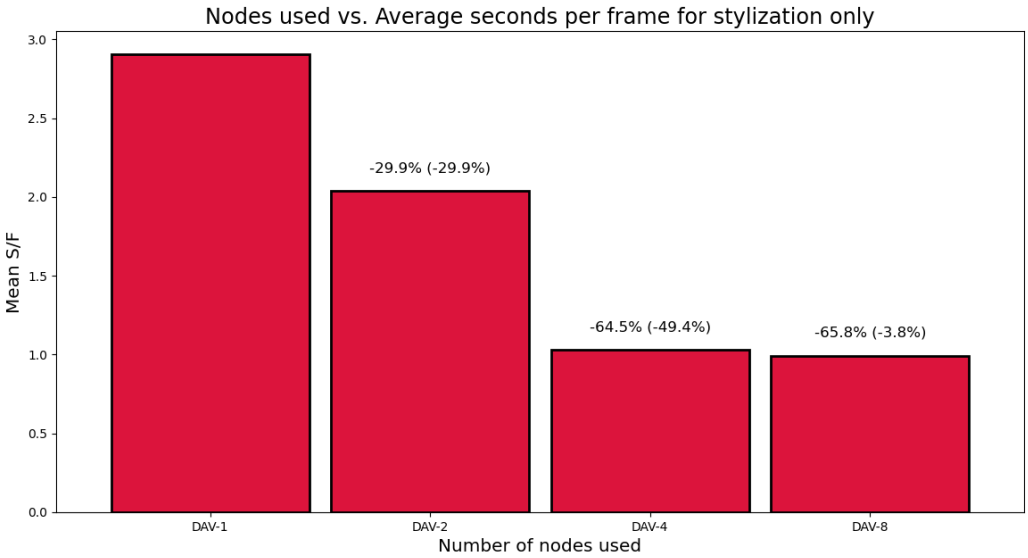

Figure 5.7: The speedup provided by distributing stylization is significant, but tapers off rapidly after DAV-4. This effect is accentuated by the fact that our dataset has several videos have zero cuts and thus cannot be parallelized in this way.

When crashes occurred, the total amount of stylization time was extrapolated based on the number of frames that had already been stylized. Because every frame after the first takes on average the same amount of time to stylize, we believe this is a fair approximation. Thankfully, DAV did not experience the same problem.

It is challenging to determine to what extent cuts are helpful to stylizing a video, partly because SFC is an imperfect metric, and partly because patterns in the data, if they exist at all, are difficult to describe with only a relatively small dataset of 30 videos. A small dataset is necessary in part because cuts must be determined manually to ensure their accuracy with

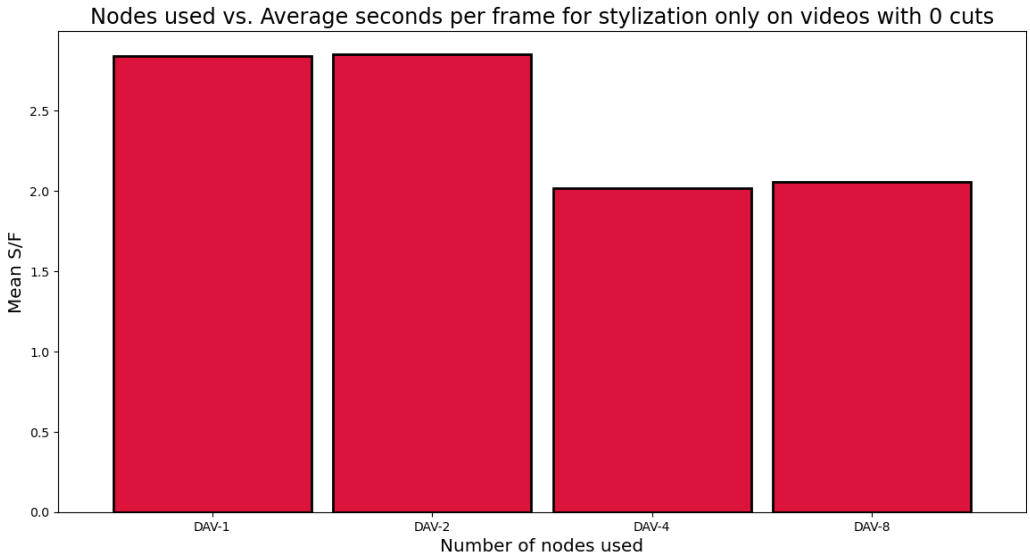

Figure 5.8: The effect in Figure 5.7 is even more pronounced when only considering results over videos with 0 cuts. The curious drop from DAV-2 to DAV-4 is mostly likely due to differences in the hardware provided by Cloudlab in each experiment. This is essentially a measure of the compute power in each experimental setup and is discussed further in Section 5.2.

our current implementation. A larger dataset would alleviate this somewhat, though this in and of itself would not improve the capacity of SFC to describe the relationship between a video's partitioning and its distributed stylization speed.

Figure 5.8 exhibits a strange quality in the data: The stylization time for DAV-4 is significantly less than DAV-2 even though we expected them to be exactly the same. This is likely because Cloudlab provided more powerful hardware for our DAV-4 and DAV-8 exper-

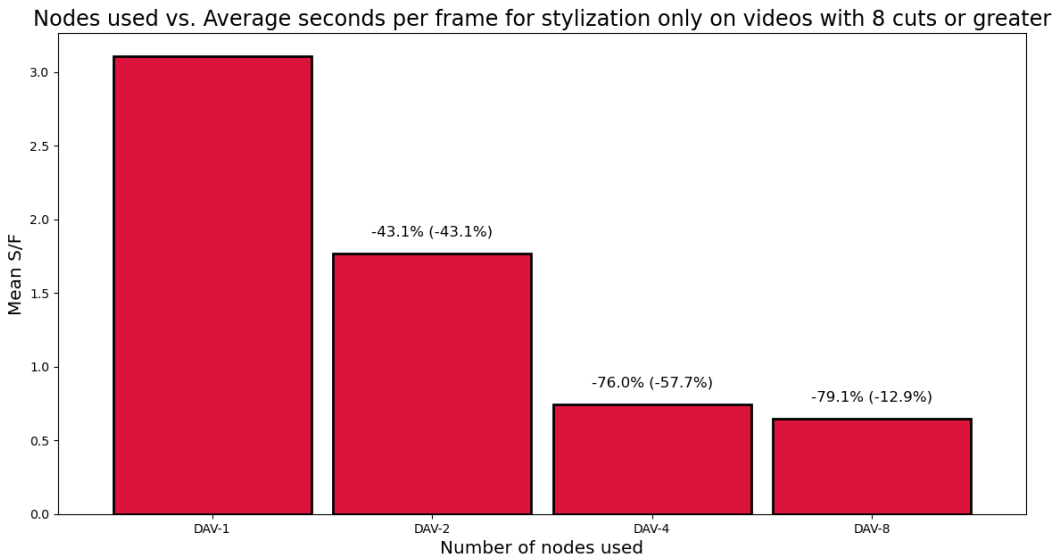

Figure 5.9: Even when looking at only videos that had greater than eight cuts, which are supposedly ideal for distributed stylization, the effect is less pronounced than we anticipated. This is possibly due to a lack of load balancing; see Section 3.3.1.2.

iments. We did not request more powerful computers—when launching each experiment, we requested the same setup from the same facility. Nevertheless, Cloudlab can only apportion what it has available, and so the machines provided can vary significantly in quality.

5.3 Future Work

There are many exciting opportunities for future research. In this section, we describe ideas we consider particularly promising in the order from most practical to most theoretical.

5.3.1 Advanced Optical Flow Calculations

DAV relies heavily on the provision of fast and accurate optical flow data. Currently, users of DAV have the option of fast optical flow data from the Gunnar Farnebäck algorithm [26] or slow optical flow data from Deepflow2 [28]. There are several better alternatives currently available. FAV uses FlowNet2 [53], for example, though FlowNet2 has a large memory footprint. LiteFlowNet [54], SPyNet [55], or PWC-Net [56, 57] are also alternatives worth considering.

Lowering the computational burden of optical flow will make stylizing longer videos possible, both by decreasing the overall time the program takes to run and by improving the efficiency of dividing a video into individually-stylized partitions.

5.3.2 Precise Scene Detection

Our work is centered more around the distribution of labor than automatically identifying scenes in videos, and so we prioritized ensuring that our algorithm produced videos that looked identical to those produced by FAV. To support this goal, we used manually-determined scenes to eliminate the chance of an improper partitioning introducing flickering to our stylized videos.

For the convenience of users who are in a hurry, we included pySceneDetect [50], a library that performs admirably yet not perfectly. We recommend that future versions of DAV or successor algorithms include a better-performing scene detection algorithm if one exists. Recent works by Haroon et al. [58] or Baraldi et al. [59, 60] are worth investigating for this purpose.

DAV is flexible on when partitions are determined. They can be computed anytime before stylization begins without blocking the program, so even a computationally burdensome scene detection algorithm could suffice provided it was more accurate than its faster competitors.

5.3.3 Flexible Consistency

We observe that no theoretical justification was provided Ruder et al. [12,42] or Sundaram et al. [46] for the current means by which consistency checks are calculated. Indeed, one factor in calculating occlusions, the motion boundaries, are handled differently in the description of FAV and its actual implementation. That it took so long for us to notice this discrepancy, long after our experiments were complete, suggests that the motion boundaries are not particularly important. The importance of the other constraints in NST for video should therefore be investigated and reassessed.

Ruder et al. [42] mention that, rather than a binary mask, it is possible to use values between 0 and 1 in occlusion tensors to express various degrees of confidence. We find this avenue of research promising. We further recommend that future research experiment with different values for the inequalities used in the definitions of occlusion detection.

5.3.4 Arbitrary Style Transfer for Video

While style transfer algorithms have long relied on pre-trained networks that can apply only one specific style [4, 12, 41, 42], new works [1, 11, 38] have introduced NST algorithms that can successfully apply style inputs they have never seen before. This has obvious

implications in NST for video. Applying an arbitrary style to a video is firmly within the realm of practical possibility.

An alternative avenue of research worth pursuing is designing a meta-network to create lightweight video stylization networks using an approach similar to Shen et al. [35]. This approach would benefit low-power devices in particular, e.g. mobile phones or tablets, because the models produced by Shen et al. are simple and lightweight.

5.3.5 Blending Frames for Arbitrary Cuts

An obvious limitation of DAV is its inflexibility when partitioning videos. DAV relies on its inputs having many cuts where two adjacent frames are functionally unrelated. Long videos with few to no cuts are the least efficient to stylize with DAV.

Deciding how to place the cuts that divide a video into partitions is ultimately an artistic decision. We desire the smoothest videos possible, and so we only split videos on clear boundaries. Introducing cuts in the middle of scenes would make a video easier to distribute across a cluster but would also introduce sudden and distracting changes to these scenes on playback. Therefore, any algorithm which could smooth the stylistic transition between two partitions would allow DAV to perform consistently and efficiently on all inputs.

5.3.6 Comprehensive Style Transfer

When we speak to most people about their ideas for style transfer, be they laypersons or computer scientists, they immediately jump to ideas that, while interesting, are not possible with current NST algorithms.

For example, we received the suggestion to take a photo of former president Barack Obama and stylize him to appear drawn in the style of the animated show *The Simpsons*. Unfortunately, at present, most algorithms can only take the colors of *The Simpsons* and apply them to the photo of Obama, giving him a pastel, watercolored appearance.

While interesting, this result does not satisfy the original request. NST can no more perform this transformation than it can draw a caricature. In effect, because we cannot reproduce shapes and patterns from the style image, many NST algorithms amount to not much more than a particularly complex photographic filter.

Isolated weaknesses in NST have been identified and solved various ways; for example, adjusting the constraints to prioritize texture synthesis over consistency [6], encouraging the network to perceive depth [40], or coarsening optical flow to provoke a two-dimensional flow of movement [43]. These solutions are specific to a subset of NST's domain—human faces, for example. While several works have sought to address this problem more holistically [37, 38], the field still awaits an algorithm that can convincingly forge a Picasso.

5.3.7 Neural Video Stylization via a Recurrent Convolutional Network

In 2015, the first convolutional recurrent neural network (CRNN) was introduced by Tang et al. [61] for the purpose of document classification, and we believe that a similarly powerful model may remove the need for optical flow calculations entirely.

Previous work has shown that optical flow files are not essential for creating reasonably high-quality videos. In [41], optical flow is only used during training and is not required for neural inference. Thus, neural networks are capable of learning how to apply temporal

consistency—they simply perform better with access to optical flow.

We believe a video NST algorithm that uses optical flow at train time could produce results similar in quality to those produced by DAV. CRNNs are a promising candidate architecture on which to base such an algorithm. It is possible to redesign temporal loss to work with recurrent connections, and therefore train a network to apply temporal relationships when stylizing videos.

5.4 Chapter Summary

DAV is functionally equivalent to FAV. Using the scale of loss from Ruder et al. [12] to evaluate FAV, we demonstrated that the losses of both algorithms across several inputs are functionally equivalent. On some inputs FAV performed slightly better and on other inputs DAV performed slightly better; on average, their loss is virtually identical. The visual samples they produced are also virtually identical.

In general, the SFC of a video is independent of its length, though outliers are easy to identify by their high SFC values. Most videos have a relatively low SFC, and so take a similar amount of time to stylize per frame.

Given the same experimental setup, DAV-8 is on average ten times faster than FAV. This is primarily due to the distribution of optical flow calculations. Distributing stylization, while useful, does not appear to yield the same amount of speedup.

Still, this brings into possibility the stylization of videos that were previously infeasible. A reduction in processing time of 90% is the difference between stylizing a video in 2.5 hours and stylizing it in 15 minutes.

There are several promising directions to take NST for video in the future. The most promising is the prospect of adapting one of the algorithms proposed by Huang et al. [11], Ghiasi et al. [1], or Gu et al. [38] to form the basis of DAV instead of Johnson et al. [4]. This would allow it to apply an arbitrary style to a video, greatly improving its capacity.

Chapter 6

Conclusions

In this thesis, we introduced the algorithm Distributed Artistic Videos (DAV). We proved that DAV is functionally equivalent to its predecessor, Fast Artistic Videos. The neural inference of the original Torch7 model is completely translated to pyTorch and can be distributed with no loss in artistic quality.

Basing the architecture on pyTorch instead of Torch7 gave DAV an initial speed boost, and distributing the work of optical flow and stylization across a collection of nodes allowed even further speedup. Across 8 nodes, the time taken to stylize a video was reduced by an average of 92%, pushing stylization time from a scale of hours to a scale of minutes and bringing into possibility the stylization of long, high-resolution videos that were previously intractable to process.

Bibliography

[1] G. Ghiasi, H. Lee, M. Kudlur, V. Dumoulin, and J. Shlens, "Exploring the structure of a real-time, arbitrary neural artistic stylization network," *CoRR*, vol. abs/1705.06830, 2017.

[2] S. Sidor, "Magenta: Music and art generation with machine intelligence." https://github.com/tensorflow/magenta, 2020.

[3] L. A. Gatys, A. S. Ecker, and M. Bethge, "A neural algorithm of artistic style," *CoRR*, vol. abs/1508.06576, 2015.

[4] J. Johnson, A. Alahi, and L. Fei-Fei, "Perceptual losses for real-time style transfer and super-resolution," in *European Conference on Computer Vision*, 2016.

[5] D. Sýkora, O. Jamriška, O. Texler, J. Fišer, M. Lukáč, J. Lu, and E. Shechtman, "StyleBlit: Fast example-based stylization with local guidance," *Computer Graphics Forum*, vol. 38, no. 2, pp. 83–91, 2019.

[6] J. Fišer, O. Jamriška, D. Simons, E. Shechtman, J. Lu, P. Asente, M. Lukáč, and D. Sýkora, "Example-based synthesis of stylized facial animations," *ACM Trans. Graph.*, vol. 36, July 2017.

[7] A. Semmo, T. Isenberg, and J. Döllner, "Neural style transfer: A paradigm shift for image-based artistic rendering?," in *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*, NPAR '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[8] P. T. Jackson, A. Atapour-Abarghouei, S. Bonner, T. Breckon, and B. Obara, "Style augmentation: Data augmentation via style randomization," *arXiv preprint arXiv:1809.05375*, pp. 1–13, 2018.

[9] X. Zheng, T. Chalasani, K. Ghosal, S. Lutz, and A. Smolic, "Stada: Style transfer as data augmentation," *arXiv preprint arXiv:1909.01056*, 2019.

[10] A. Mikolajczyk and M. Grochowski, "Style transfer-based image synthesis as an efficient regularization technique in deep learning," *CoRR*, vol. abs/1905.10974, 2019.

[11] X. Huang and S. Belongie, "Arbitrary style transfer in real-time with adaptive instance normalization," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1501–1510, 2017.

[12] M. Ruder, A. Dosovitskiy, and T. Brox, "Artistic style transfer for videos and spherical images," *International Journal of Computer Vision*, vol. 126, pp. 1199–1219, Nov 2018. online first.

[13] D. Kirk *et al.*, "NVIDIA CUDA software and GPU parallel computing architecture," in *ISMM*, vol. 7, pp. 103–104, 2007.

[14] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[15] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS workshop*, 2011.

[16] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: a modular machine learning software library," tech. rep., Idiap, 2002.

[17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, pp. 8024–8035, 2019.

[18] H. He, "The state of machine learning frameworks in 2019." https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry, 2019.

[19] Y. Jing, Y. Yang, Z. Feng, J. Ye, Y. Yu, and M. Song, "Neural style transfer: A review," *IEEE transactions on visualization and computer graphics*, 2019.

[20] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[21] A. K. Chauhan and P. Krishan, "Moving object tracking using gaussian mixture model and optical flow," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 4, 2013.

[22] S. S. Kumar and M. John, "Human activity recognition using optical flow based feature set," in *2016 IEEE international Carnahan conference on security technology (ICCST)*, pp. 1–5, IEEE, 2016.

[23] M. H. Kolekar and D. P. Dash, "Hidden markov model based human activity recognition using shape and optical flow based features," in *2016 IEEE Region 10 Conference (TENCON)*, pp. 393–397, IEEE, 2016.

[24] B. D. Lucas, T. Kanade, *et al.*, "An iterative image registration technique with an application to stereo vision," 1981.

[25] B. K. Horn and B. G. Schunck, "Determining optical flow," in *Techniques and Applications of Image Understanding*, vol. 281, pp. 319–331, International Society for Optics and Photonics, 1981.

[26] G. Farnebäck, "Two-frame motion estimation based on polynomial expansion," in *Scandinavian conference on Image analysis*, pp. 363–370, Springer, 2003.

[27] G. Farnebäck, *Polynomial expansion for orientation and motion estimation*. PhD thesis, Linköping University Electronic Press, 2002.

[28] P. Weinzaepfel, J. Revaud, Z. Harchaoui, and C. Schmid, "DeepFlow: Large displacement optical flow with deep matching," in *IEEE Intenational Conference on Computer Vision (ICCV)*, (Sydney, Australia), Dec. 2013.

[29] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[30] T. Salimans and D. P. Kingma, "Weight normalization: A simple reparameterization to accelerate training of deep neural networks," *CoRR*, vol. abs/1602.07868, 2016.

[31] D. Ulyanov, A. Vedaldi, and V. S. Lempitsky, "Instance normalization: The missing ingredient for fast stylization," *CoRR*, vol. abs/1607.08022, 2016.

[32] L. A. Gatys, A. S. Ecker, and M. Bethge, "Image style transfer using convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2414–2423, 2016.

[33] Y. Li, N. Wang, J. Liu, and X. Hou, "Demystifying neural style transfer," *arXiv preprint arXiv:1701.01036*, 2017.

[34] A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. Smola, "A kernel two-sample test," *Journal of Machine Learning Research*, vol. 13, no. Mar, pp. 723–773, 2012.

[35] F. Shen, S. Yan, and G. Zeng, "Neural style transfer via meta networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[36] V. Dumoulin, J. Shlens, and M. Kudlur, "A learned representation for artistic style," *arXiv preprint arXiv:1610.07629*, 2016.

[37] C. Li and M. Wand, "Combining markov random fields and convolutional neural networks for image synthesis," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2479–2486, 2016.

[38] S. Gu, C. Chen, J. Liao, and L. Yuan, "Arbitrary style transfer with deep feature reshuffle," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[39] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[40] X.-C. Liu, M.-M. Cheng, Y.-K. Lai, and P. L. Rosin, "Depth-aware neural style transfer," in *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*, NPAR '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[41] H. Huang, H. Wang, W. Luo, L. Ma, W. Jiang, X. Zhu, Z. Li, and W. Liu, "Real-time neural style transfer for videos," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[42] M. Ruder, A. Dosovitskiy, and T. Brox, "Artistic style transfer for videos," *CoRR*, vol. abs/1604.08610, 2016.

[43] J. Delanoy, A. Bousseau, and A. Hertzmann, "Video Motion Stylization by 2D Rigidification," in *Expressive 2019 - 8th ACM/ Eurographics Proceedings of the Symposium*, (Genoa, Italy), May 2019.

[44] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[45] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[46] N. Sundaram, T. Brox, and K. Keutzer, "Dense point trajectories by gpu-accelerated large displacement optical flow," in *European conference on computer vision*, pp. 438–451, Springer, 2010.

[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[48] S. Gross and M. Wilber, "Training and investigating residual nets." http://torch.ch/blog/2016/02/04/resnets.html, 2016.

[49] C. Carwin, "Convert torch to pytorch." https://github.com/clcarwin/convert_torch_to_pytorch, 2018.

[50] B. Castellano, "pyscenedetect." https://github.com/Breakthrough/PySceneDetect, 2020.

[51] F. Bellard, "Ffmpeg." https://github.com/FFmpeg/FFmpeg, 2020.

[52] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[53] E. Ilg, N. Mayer, T. Saikia, M. Keuper, A. Dosovitskiy, and T. Brox, "Flownet 2.0: Evolution of optical flow estimation with deep networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[54] T. Hui, X. Tang, and C. C. Loy, "Liteflownet: A lightweight convolutional neural network for optical flow estimation," *CoRR*, vol. abs/1805.07036, 2018.

[55] A. Ranjan and M. J. Black, "Optical flow estimation using a spatial pyramid network," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4161–4170, 2017.

[56] D. Sun, X. Yang, M. Liu, and J. Kautz, "Pwc-net: Cnns for optical flow using pyramid, warping, and cost volume," *CoRR*, vol. abs/1709.02371, 2017.

[57] D. Sun, X. Yang, M.-Y. Liu, and J. Kautz, "Models matter, so does training: An empirical study of cnns for optical flow estimation," *arXiv preprint arXiv:1809.05571*, 2018.

[58] M. Haroon, J. Baber, I. Ullah, S. M. Daudpota, M. Bakhtyar, and V. Devi, "Video scene detection using compact bag of visual word models," *Advances in Multimedia*, vol. 2018, 2018.

[59] L. Baraldi, C. Grana, and R. Cucchiara, "A deep siamese network for scene detection in broadcast videos," in *Proceedings of the 23rd ACM International Conference on Multimedia*, MM '15, (New York, NY, USA), p. 1199–1202, Association for Computing Machinery, 2015.

[60] L. Baraldi, C. Grana, and R. Cucchiara, "Analysis and re-use of videos in educational digital libraries with automatic scene detection," in *Italian Research Conference on Digital Libraries*, pp. 155–164, Springer, 2015.

[61] D. Tang, B. Qin, and T. Liu, "Document modeling with gated recurrent neural network for sentiment classification," in *Proceedings of the 2015 conference on empirical methods in natural language processing*, pp. 1422–1432, 2015.

Paul Galatic

Curriculum Vitae

Education

2015-2020 **Bachelor of Science in Computer Science**, *Rochester Institute of Technology*, GPA – 3.87.
Graduated with Honors

2018-2020 **Master of Science in Computer Science**, *Rochester Institute of Technology*, GPA – 4.0.
Specialized in Artificial Intelligence
Graduated with Honors

Masters Thesis

Title *Divide and Conquer in Neural Style Transfer for Video*

Supervisor Dr. M. Mustafa Rafique

Description Applying neural style transfer for video is a time-consuming process, and distributing that work among eight computers can reduce the amount of time required to perform video stylization by an average of 92% with no degradation of quality.

Experience

Summer 2019 **Software Engineer Intern**, Microsoft, Boston MA.
Added live updates to Azure Deep Learning Experiments website. The updates are delivered in less than 10 seconds 99.9% of the time, and under normal circumstances appear in 1 second.

Summer 2018 **Machine Learning Intern**, Assured Information Security, Rome NY.
Developed and trained Deep Convolutional Generative Adversarial Network for steganographic image synthesis. The network reached 99% accuracy with realistic-quality images. It could successfully encode/decode 0.4 bits per pixel on a 64x64 image.

Spring 2018 **Software Engineer Intern**, GE Aviation, Clearwater FL.
Developed script to visualize flight plans in Google Earth for testing and verification purposes. Refactored flight planning software to be functional in new package version.

Summer 2017 **Software Engineer Intern**, MITRE Corporation, Boston MA.
Wrote Amazon Echo app that could take voice input, search company database, and provide summaries of research articles. App uses Echo to interface with Watson Conversation.

Awards and Extracurriculars

2017 Co-founder of RIT Artificial Intelligence Club (RIT-AI)

2018-2019 President of RIT-AI

2019 RIT Outstanding Undergraduate Scholar Award

2020 RIT Excellence in Student Life Award