Fast and Temporally Consistent Neural Style Transfer in Videos

Devarti Mahakalkar

Master of Science in Data Science
The University of Bath
2021-2022

Fast and Temporally Consistent Neural Style Transfer in Videos

Submitted by: Devarti Mahakalkar

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Master of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

The main aim of this research project is to design a Machine Learning model that will convert animated or real-time videos to artistic style videos while maintaining the temporal consistency in the output video. Although image processing tasks have been in the spotlight for a while now, their video counterparts present additional challenges. I am proposing a methodology to transfer style in videos faster with stability in the output. A lightweight feed-forward neural network is used which will help in training and testing the model faster, meanwhile making it feasible to use it in real-time. While converting the video into artistic style from the style transfer methods used for images can result in a flickering effect in videos. I am using two temporal loss functions to prevent the flicker in the stylized output video. The first temporal loss function takes two adjacent video frames to calculate the difference in stylization using optical flow and occlusion mask with luminance difference constraint. Second function checks for long-term consistency in the video, which will try to compare the video frame with previous video frames in longer intervals and deals with maintaining the consistency in the long term for an object in the video frame that is occluded and reappears again. This helps in making the video more stable.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Peter Hall for his constant support and guidance throughout my dissertation. Without his valuable inputs, completing this research project would not have been possible. I am grateful for his willingness to wade through the relative long proofs.

Devarti Mahakalkar
University of Bath
September 2022

Chapter 1

Introduction

One of the interesting applications of image processing has been artistic style transfer which can convert any photographs into artistic style painting using brush strokes, tiling, etc. With the resurgence of Artificial Intelligence in the past decade, powerful computing methods have made it possible to approach classic image processing problems in entirely new ways. With the recent advancement, a number of ground-breaking approaches for style transfer in images and videos have been made possible by artificial intelligence.

1.1 The Problem

The Style Transfer method in images takes a real-time photograph, i.e., a content image and an artistic photograph, i.e., a style image, and tries to redraw the content image with the texture of the artistic image. There have been many advancements recently to do the style transfer task using convolutional neural networks. The convolutional network takes the high-level feature representation of both the images and minimizes the distance between them to give an output.

For style transfer in the video, we take a video and a style image and try to blend the style image into the video. But the video cannot implement the same technique as the still image. A video is taken as a sequence of frames; these frames keep changing with time and are independent of each other. We can process each frame with the style image to get the stylized output, and the neural network converts each frame with a different texture of style which when seen in the video, can have inconsistent texture. This inconsistent texture in the video causes flickering in the video. Figure 6.5 demonstrates the inconsistent style transfer in two adjacent video frames.

This flicker can occur for various reasons, like suppose the object is moving and pixel value in the corresponding frames occluded or the object disappears in next frames, and pixel value cannot be detected or complex optimization technique used in the style transfer method can fall into different local minima of the frame, or a filter used for rendering the style is not stable in terms of statistics, like the average colour. These problems make the stylized video temporally inconsistent, making style rendering correct in some regions of the frame while not changing the content for the next frame. To counter this problem, we need to take a new dimension into account which is time. This will maintain the temporal consistency so that the video is stable.

1.2 Project Objectives

In this project, I am proposing a feed-forward neural network called ReCoNet. I am using the improved neural network compared to Gao et al. (2018). This neural network will generate a temporally stable video with rich artistic textures and strokes. The network stylizes the video frame by frame which is passed through the encoder and a decoder. Then, a VGG loss network is used to capture the perceptual style of the output produced from the decoder.

To solve the problem of inconsistency in the videos, I am incorporating the neural network with the temporal loss which uses optical flow(2.2) and occlusion masks(4.4). The temporal loss will be calculated between two adjacent frames. The effect can be observed in Figure 6.5. I am improving the video's temporal loss function for better stability by adding a luminance warping constraint. There exists luminance differences(Horn, 1974) in real-world videos and animations and it cannot be captured by the optical flow only. This constraint will track the luminance difference on pixels between the two consecutive video frames.

To get better consistency in the video, I am introducing a new temporal loss function called the long-term temporal loss function which will maintain the long-term consistency and give the same texture for the object in later frames as well and not just consecutive frames. An example of long-term consistency can be seen in Figure 6.6.

Another advantage of this method is that it takes less time to train the model on modern GPUs and also produces the stylized output faster compared to Chen et al. (2017); Huang et al. (2017). This is because the neural network design is lightweight and I am not estimating the optical flows but taking the ground-truth optical flows to calculate the loss function in training only.

1.3 Structure

The structure of the dissertation is given below:

Chapter 2 reviews the state-of-the-art techniques related to style transfer before the neural network was introduced in this field. Then some techniques use neural networks to stylize the image using artistic images when images are taken as input and videos are taken as input.

Chapter 3 provides the details of the dataset used to train the model and the environment used to implement the code and execute it.

Chapter 4 gives an idea about the design and architecture of the proposed method used to generate artistic style videos. It also includes the temporal loss functions used.

Chapter 5 includes the workflow, implementation of the method to train the model and testing it on different video sequences to get stylized video.

Chapter 6 provides the details about the evaluation and results, the metrics used to check the stability in the videos and compares the model with previous methods.

The last Chapter 7 provides the conclusion, limitations of the proposed method and discussion on future work.

Chapter 2

Background and Literature Review

2.1 Stylization Techniques

The fast development in photorealistic computer graphics over the past few decades has inspired many inventions of photograph rendering artistic diversity. Non-Photorealistic Rendering (NPR) became a new research area and it was dedicated to developing a tool that would produce an artistic photograph. Many of these techniques were based on rendering the artistic effects by brush strokes, regional style characteristics, pen and ink strokes, tiling, tonal depiction, and example-based rendering (Kyprianidis et al., 2012).

Early the research was mainly based on brush strokes and regional-based rendering. Now the research has shifted to example-based rendering which includes deep learning methods to implement rendering. Although earlier methods can produce only a specific type of artistic facet, example-based rendering can stylize any type of art.

2.1.1 Stroke Based Rendering

Stroke-based rendering is the earlier stylization technique that reproduces the artwork by constructing rendering marks such as lines, brush strokes, and patches using an algorithm(Vanderhaeghe and Collomosse, 2013). There are two algorithms that would cover the canvas with these primitives, which are local strokes and global strokes algorithm.

The local strokes algorithm would find the strokes displacement and edges on the pixels within a specified range of spatial neighbours. The varying rectangular strokes in the pixel, including the Sobel gradient, were sufficient to render the artwork. This was automated by Litwinowicz (1997), who also extended the method in videos that would convert impressionist painting brush strokes and push them from frame to frame to detect pixel motion through video frames which would create a temporally coherent stylized video.

On the other hand, the global strokes algorithm optimizes all the strokes displacement in the whole image so that they match with the movement of the real strokes. It brought many advancements in the retention of the artistic details using the Sobel gradient and using it in video frames to get a temporally coherent video(Collomosse, Rowntree and Hall, 2005). It also created the brush strokes in video frames by tracking regions over time by using rotoscoping and created motion data that would give a non-photorealistic animation with flicker.

2.1.2 Example Based Rendering

Example-based rendering was advanced by Hertzmann (1998). In this, an exemplar artistic image is compared with an image in an attempt to render the brush strokes in many images of different sizes. This method was expanded rapidly as this method would not create only a single artwork but could reproduce any images from a given artistic style.

The example-based rendering can be divided into two branches: colour transfer rendering and texture transfer rendering. The texture transfer rendering reproduces the colours of the exemplar artistic image by matching the regional colour histogram of the artwork with the content image(Neumann and Neumann, 2005). A similar method was done by optimizing the gradient and histogram(Xiao and Ma, 2009).

One of the first example-based methods using the texture transfer called 'Image Analogy'(Hertzmann et al., 2001) uses machine learning to render the artistic effect. The machine learning algorithm learns the mapping of the source image and this mapping can be used in texture transfer between the artwork and the source images. The patch in the texture uses the nearest neighbour algorithm to find the nearest matching patch in the unaltered image and the image being stylized. One of the limitations of the example-based techniques is that it reduces the domain of the artistic image while rendering. The method basically relies on the artistic image texture being copied to the source image and then optimizing the global difference with the output images.

2.1.3 Neural Style Transfer

The most recent example-based rendering technique uses Neural networks to automatically render the artistic effects on the source image. The neural network used in the style transfer is Convolutional Neural Network (CNN).

CNN is one of the most common neural networks used for pattern recognition in images(Fukushima and Miyake, 1982). It is based on human visual systems that use the convolutional function for classification, segmentation and creating new objects. Convolutional layers are very effective in image processing as they operate on a square patch called a window which progressively moves from top to bottom and left to right to cover the entire image. Neural Style Transfer (NST) uses CNN as it optimally combines the artistic and source image by extracting both complex texture representations and high-level content.

The first successful style transfer by neural networks was proposed by Gatys, Ecker and Bethge (2016) using the texture synthesis(Gatys, Ecker and Bethge, 2015) of the source image called content image. He took the image with random noise that would consider the texture from the style image and extract major features of the content image. The technique used a VGG19 deep neural network that would take the high-level features of the images to get the spatial information. The loss function is added to minimize the distance between the content and texture of the image to get the stylized image. Each layer in the neural network with an activation function will respond to the image from feature maps which is a set of filtered images. The Euclidean distance between the output image and the content image after each neural network layer is taken as the content loss. Then finding the correlation between the feature maps called the gram matrix for style loss, that matches the statistical information between the content and style. Each of these losses will be contributing to the calculation of the loss function. The pixels of

the images are matched with the brushstrokes. After combining the style and content image, the optimization is done using gradient descent. The VGG19 used in the model made the image style transformation results good and faster but the solution provided was computationally heavy.

Johnson, Alahi and Fei-Fei (2016) tried to improve the above method(Gatys, Ecker and Bethge, 2016). The differences were in stochastic gradient descent optimization and loss function. The perceptual loss function in this method would consider the high-level features of the images rather than the pixel values of the image. The high-level features gave better results. Johnson used two networks: image transformation and loss network. Image transformation will take the features of the images and the loss function will calculate the loss function to check the difference between features. The method used a feed-forward network that would produce output faster as compared to the earlier methodGatys, Ecker and Bethge (2016). The drawback of this method was that each network could only apply one style to the content image; to train a new style, a completely new network had to be used.

AdaIN framework(Huang and Belongie, 2017; Li et al., 2019) for style transfer had the advantage of stylizing the content image with any style image as previously feed-forward networks were limited to performing stylization on a single style. The method was inspired by the instance normalization layer which will normalize the feature statistics of the image. Adaptive Instance normalization will take the mean and variance of the content image and try to match the style image. A decoder generates the stylized image. Normally, feed-forward networks use Batch normalization, using mini-batches to train and replace the features while generating the desired output. This autoencoder uses normalization which makes training by normalization of features easy. The feature channel detects brush strokes in the style image and these strokes will have high activation for the features. The AdaIN will generate output with the same high activation. The limitation of this technique is that training is time-consuming as we have to train the model for many epochs.

Many of the neural style transfer techniques used have a limitation that they depend on the VGG model and sometimes result in a content leak. To improve this, a depth compound was added to this that would calculate the depth loss and preserve the stylized output(Liu et al., 2017).

2.2 Neural Style Transfer in Videos

While rendering the style in videos and achieving a stable artistic video sequence, we require to compare the two consecutive frames of video using Optical flow. Optical flow is used while discussing object tracking and activity recognition. It is the measure of how the pixels move between frames in a video. There are two types of optical flow: dense and sparse. The sparse optical flow will take a few features while dense will consider all the features in an image. The dense optical flow will see the pixel movement on consecutive frames. Style transfer requires dense optical flow.

The first CNN architecture that used optical flow estimation was called FlowNet (Dosovitskiy et al., 2015). Better results were achieved in making the video consistent when the CNN was taught at each pyramid level to measure the optical flow and then up-sample to

the next level with the second image warped to the first level(Ranjan and Black, 2017).

Another optical flow method included a disparity map with the spatial-temporal loss (Hosni et al., 2011). The method improves the quality of reconstruction of the stylized output by computing a disparity map using the local stereo approach. The assumption inside the filter kernel is that every pixel in the image has equal disparity. But this method fails when the filter kernel overlaps with depth discontinuity.

Now, we will see the previous neural style transfer techniques used in the videos. The style transfer method in images presented earlier (Gatys, Ecker and Bethge, 2016) was extended to implement the style transfer technique on videos (Ruder, Dosovitskiy and Brox, 2016). Processing each frame of the video causes inconsistency in the stylized video. To smoothen this inconsistency, the regularization method was applied by taking the temporal consistent loss function. It would take the optical flow into account to improve the video performance. To enforce stronger consistency,two temporal functions were used called the short-term consistency(also used in Huang et al. (2017)) and long-term consistency. Short-term consistency is used where areas that have not changed between the two frames are then initialized with the desired appearance, while the rest of the image must be rebuilt through the optimization process. The long-term consistency would take the temporal loss for more than just two consecutive frames. This helped to maintain consistency when the object is occluded and dis-occluded in the later frames.

There are various viewpoints about the desired quality of the stylized video. A method was proposed to mimic hand-drawn paintings on two-dimensional animation(Delanoy, Bousseau and Hertzmann, 2019). This method was designed to coarsen the optical flow of the video by human-guided motion segmentation, but it created a distortion effect. The main features of the video appeared to be flattened in cardboard that moved, and rendered on top of each other. This method created output that is more similar to classic animation with simple movement patterns.

Wang et al. (2020) introduced compound regularization which will fit the temporal variation and proposed that regularization in a single frame can result in underfitting and does not render the style between two frames effectively thus making videos temporally inconsistent. This gave better smoothness to the stylized video. To get consistency in the video, the method passed both content and style images to the filter predictor network (Avatar-Net) and dynamically gave the linear transformation in different channels. Avatar-Net (Sheng et al., 2018) neural network gave consistent correlation and better results by matching patches.

Another method to get an output video after stylization dealt with temporal inconsistency by finding the distance of the difference between two consecutive stylized frames and the difference between two consecutive original frames. This frame-difference loss gave a good result as using the optical flow method (Xu, Xiong and Hu, 2021). Extending the Liu et al. (2017) method of using the depth combined with optical flow, Liu et al. (2021) provided a better result by achieving good rendering quality. This method would not lose the semantics of the pixel and provide flicker-less video.

Chapter 3

Requirements

3.1 Dataset

The dataset used to train the model in the proposed method is MPI-SintelButler et al. (2012). This dataset is obtained from an animated short film Sintel. The dataset contains varied motion, illumination, blur, atmospheric effects, scene structure, etc. The dataset contains the images from albedo, clean and final pass, the occluded region of the image and the ground-truth optical flow file with their visualization.

The neural network in the proposed method is taking images from the clean pass. For calculating the temporal function, we are taking .flo files from the flow folder which is the optical flow of the images and the ground truth mask from the occlusion folder. For testing, we are taking 12 video sequences from the test folder from the MPI-Sintel dataset, Davis dataset(Davis, 2018) and real-world videos from Videvo.net(Videvo, 2018). These real-world videos are split into image frames for stylization.

3.2 Setup

I am using PyCharm IDE for the implementation of my method. The code is written in Python and Pytorch framework. I used command line argument to run the code. For training the model, I used GPU provided by Google Colab Pro. The GPU used to train the model here was Nvidia Tesla P100 which had 16GiB GPU memory using the 11.3 version of CUDA. Command line arguments were executed in the Colab notebook to train the model and test the result. It took 3 hours to train the model with 992 images for 60 epochs.

Chapter 4

Design

In this chapter, the theory behind neural style transfer is explained. The design of the proposed method and models used to get the stylized output is shown. Different loss functions are used to get the output with desired quality, including the content reconstruction loss, style loss and total variation loss. To get a stable style transfer in the videos, temporal loss functions are introduced.

4.1 Network Architecture

An improved feed-forward network called ReCoNet is used in the stylization network. This network contains Autoencoder(Baldi, 2012) which is a pure CNN-based design. The network is divided into two separate parts called encoders and decoders and both perform different functions. The encoder is used to expand and encode the images to the filtered images containing the perceptual information, called feature maps. The decoder is used to decode the feature maps to the stylized image.

Each network is trained on one style only such that the texture information of the style can be encoded in its weight. Every layer in the network can be seen as the image filter that will extract certain information from the image and give a feature map. The image will be passed to the next convolutional layers and the processing in this hierarchy will provide more explicit information about the input and resemble more with the content of the input image.

The content image comprised of a Red-Green-Blue (RGB) image is passed to the network in form of a tensor. The input tensor is padded first using Reflection Padding which will help to extract the semantic information better and then passed to the convolutional layer to get the feature map of the image. To keep the data distribution uniform in the feature map, the next layer used is Instance normalization (Ulyanov, Vedaldi and Lempitsky, 2016). The instance normalization has an advantage over batch normalization as we calculate mean and standard deviation across each feature space rather than image space making it more impactful. After normalization of the image filter, it is passed to rectified linear unit(ReLU) activation function, being a non-linear function, it increases sparsity making them more efficient in terms of time and space complexity.

The convolution layer, which includes Reflection Padding, Conv2d, Instance Normalization and Relu layer, expands the image from 3 channels to 32, making it from 32 to 64 channels

in the second convolutional layer and finally from 64 to 128 channels. These convolution layers encode the image frame to abstract feature space. After the three convolution layers, we have five residual layers in the encoder. The expanded tensor goes to the residual network which performs stylistic adjustments to the feature space. The residual layer also learns a residual mapping between the image frame and the desired stylized output image(Gross and Wilber, 2016).

Let's assume that x is the feature map of the image, and the residual network is F(x). The residual layer will give

$$R(x) = F(x) + x$$

where R(x) is the residual network. The residual network works better as the CNN has many multiplication operations whereas the addition convergence in the residual structure makes the style transfer faster and gives enhanced performance(Huang et al., 2021). Figure 4.2 shows a visual representation of the Residual layer.

The image frame is now passed to the decoder to get the stylized output from the feature maps. The image is first passed to the upsampling layer. This upsampling layer uses bilinear interpolation, which is used to recover the resolution of the processed image compared to the original input frame. It also helps in reducing the checkerboard artefacts from the frame(Odena, Dumoulin and Olah, 2016). The tensor is then reduced from 128 channels to 64 using the deconvolutional layer, then from 64 to 32 channels and finally to 3 channels. The process involved in the deconvolution layer is decoding from the abstract feature space to the image space. The last deconvolution layer which is reduced from 32 to channels uses tanh activation function.

The stylization network design can be seen in Figure 4.1.

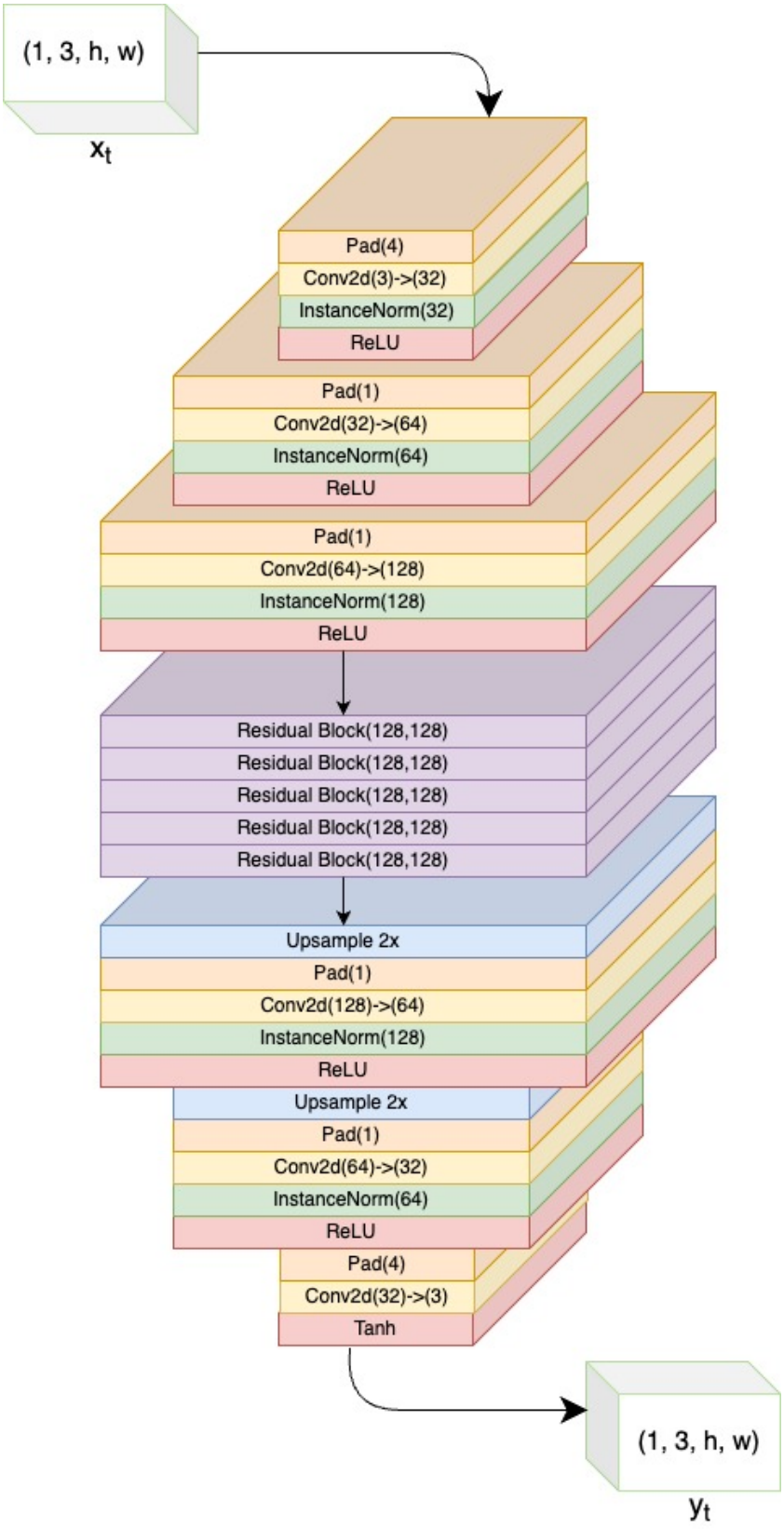

Figure 4.1: The neural network design of ReCoNet. Input image x_t of size(1,3,h,w) represents (batch size, no. of channels, height, width)

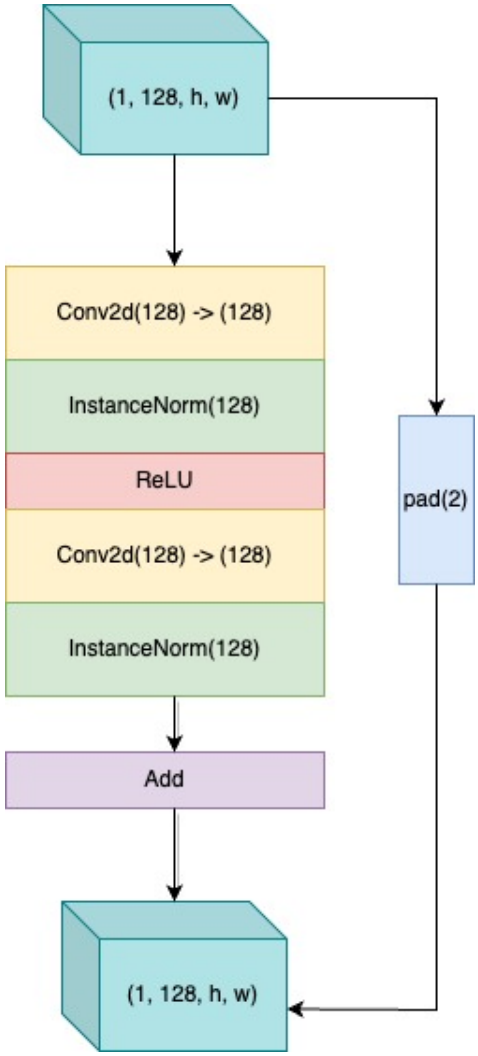

Figure 4.2: Residual Block of the neural network ReCoNet

4.2 Loss Network

The loss network is the neural network which we are using to extract the features of the images and calculate the loss function for neural style transfer. I am using a pre-trained VGG network (Simonyan and Zisserman, 2014) to extract the content and style features from the image that has been generated by the style network. The loss network will help to define the output image better by calculating the loss functions.

VGG is a Convolutional Neural Network. It stands for Visual Geometry Group. The network has 16 layers of depth and it was trained on over 1 million image datasets using ImageNet dataset(Deng et al., 2009; Russakovsky et al., 2015). VGG has good performance over the ImageNet dataset for classification as the VGG network has two tasks to do, those are feature extraction and classification. As this classification is useless for style transfer operation, we remove the last two layers of the network which are used for classification. The network is used for the extraction of features from the images to calculate the content and style loss functions.

The feature extraction block is the sequence of several layers where the output of the layer is the input of the following one. Each layer can be seen as a feature extractor. The

closer to the input layer the image is, the lower-level features detected are. For instance, the first layer of the VGG network detects small edges in the image; the second layer combines them into corners and curves. The third layer combined the corners and curves in circles, rectangular forms or textures and so on. Finally, we reach the last layer of the network, which detects the high-level features.

VGG network has different variety, and the type of VGG used in neural style transfer varies. Some have used VGG-16(Johnson, Alahi and Fei-Fei, 2016; Dumoulin, Shlens and Kudlur, 2016) and others have used VGG-19(Gatys, Ecker and Bethge, 2016; Li and Wand, 2016). Theoretical, there is no benefit of using one network over the other. I am using the VGG-16 neural network, which is publicly available. The pre-trained model weight is downloaded, which is around 566MB in PyTorch; the network along with the weights must be loaded into memory during the time of training in addition to the rest of the program.

The purpose of Neural Style Transfer is to get an image with an artistic feature which humans find pleasing. For this, we need to use the mean squared error between the content and generated image and style with the same generated image. We use mean squared error because it is highly sensitive to small changes which can be perceived by human eyes.

Now suppose, if we run the style image and the generated image through the VGG network and use the post-activation layers of VGG. The loss function is defined by the mean squared error between the interpretations of style and generated images of each respective VGG layer. Passing the image to the VGG network is the evaluation criteria for Neural Style Transfer and this is why VGG is necessary.

In order to quantify the high-level perceptual and semantic differences between images using the VGG network, we define two perceptual loss functions.

4.2.1 Style Loss

Style loss is used to check whether the features, like colour and pattern, of the style image, are rendered into the generated image. It is the measure of how different the lower-level features of the generated image are from the style image.

We calculate style loss from all the layers of VGG, whereas we calculate content loss from higher layers only. The style loss cannot be calculated directly by the intermediate features of two images. To extract the style of the image, we need to extract the feature maps from different layers of the image(RGB) and find the correlation between these feature maps or filters. As stated, style information is the measure of the correlation between the feature maps of the image.

To find the correlation between the filters, we calculate the dot-product between the vectors across the activations of the two filters. The matrix we get from the dot-product is called Gram-matrix. The Gram-matrix encodes the correlation between different features but not the presence of specific features. It tries to match the distribution of features by minimizing the difference between the feature distributions(Li et al., 2017b).

Let $A^l_{ij}(I)$ be the activation of the I^{th} layer, and i and j are the i^{th} and the j^{th} channel in the layer respectively in an image I. H is the height and W is the width of the input image. The gram matrix G is given by:

$$G^l_{i,j}(I) = 1/H^l W^l \sum_{h=1}^{H^l} \sum_{w=1}^{W^l} A^l_{(h,w,i)}(I) A^l_{(h,w,j)}(I)$$

The loss for each layer is calculated by finding the difference between the gram matrix of style and the output image.

$$Loss^l_{style} = 1/M \sum_{i,j} (G^l_{i,j}(s) - G^l_{i,j}(g))^2$$

where M is the number of used layers in the network. $G^l_{i,j}(s)$ is the gram-matrix of the style image at layer l. $G^l_{i,j}(g)$ is the gram-matrix of stylized output image at layer l.

The loss of each layer is added to get the total style loss:

$$Loss_{style} = \sum_l w^l Loss^l_{style}$$

where w^l is the weight given to each layer.

The first convolution in each layer will be used for the extraction of style features as more weight will be given to the front layers than the deeper ones. The gram matrix has an advantage for calculating the style loss as it captures the feature distribution of feature maps in the given layer.

4.2.2 Content Loss

Content loss is to check whether the content is present in the generated image while the algorithm changes the style. The features of the generated image are compared with the content image. In this manner, the authenticity of the content image is preserved, and the style elements are added from the style image. In the VGG model or any CNN model, it is seen that the higher layers of the network focus on the content of the image and the lower layers focus on the individual pixel values and don't capture the content but only the texture. In my implementation, I am using `relu3_3` layer of VGG-19 to calculate the content reconstruction loss.

$$Loss_{content} = 1/2 \sum_{i,j} (y^l_{i,j} - x^l_{i,j})^2$$

where y is the generated image at layer l and x is the content image at layer l.

Unlike style loss, taking a combination of layers for calculating content loss doesn't make much difference. Optimizing the content loss from one layer can optimize it for all subsequent layers. The content loss deals with the activation at higher layers of the loss network. Evidence shows that the feature maps in higher layers are activated in the presence of different objects, such that if two images have the same content, they should have alike activations in higher layers of the network, i.e., content and generated image should have alike activations(Ganegedara, 2019).

4.3 Total Variation Loss

Total variation measures the relation between neighbouring pixels, assuming that neighbouring elements have meaningful relations. While rendering the style into the content, it can be seen that the generated image has a lot of high-frequency noise, meaning the uneven bright and dark pixels in the image. To reduce this noise and make the neighbouring elements close to each other in value, i.e., making the spatial continuity between the pixels in the image, we use the total variation loss function. This regularization loss function can be found by the sum of absolute differences for the neighbouring pixels in an image(Johnson, Alahi and Fei-Fei, 2016).

$$Loss_{tv} = \sum_{i,j} |x_{i,j} - x_{i+1,j}| + |x_{i,j} - x_{i,j+1}|$$

where $x_{i,j}$ is the pixel value at coordinate i,j .

This function will help reduce the total variation loss by making the values of the neighbouring pixels closer to each other.

4.4 Temporal Loss

To enforce temporal consistency in the videos, we use the temporal function. When two adjacent image frames are given as input to the model with the same style image, the styling elements in the output of two frames can change even with the same content. This inconsistency in the stylizing of the two adjacent frames results in flickering artefacts in the output video. We need to incorporate the optical flow information between two frames and define a temporal loss function.

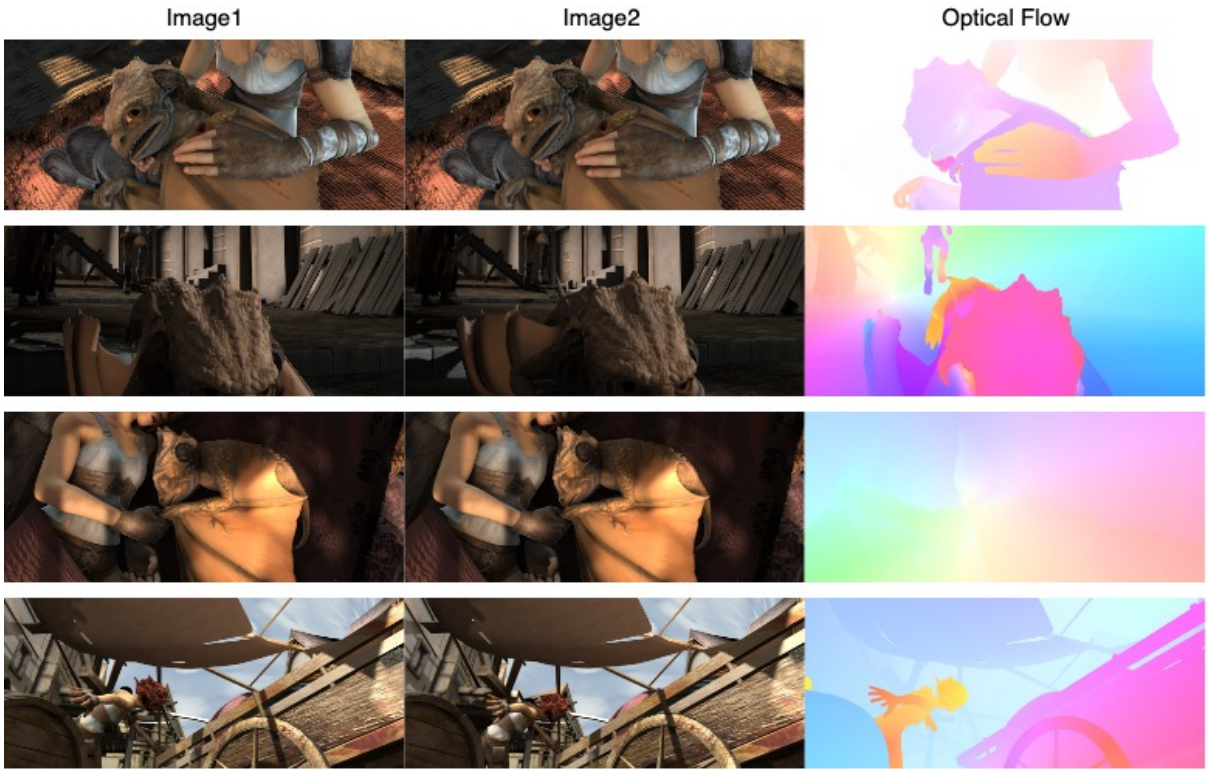

Figure 4.3: Optical Flow Examples

To minimize the flickering, we need to keep the styling consistent in the adjacent image frames. While considering two adjacent frames of the videos, most of the content in both frames remains constant, so ideally, the stylization should also remain unchanged and only the areas in the frame which are changed from the previous frame should be stylized differently than the previous frame. This method will significantly enhance the coherence and smoothness of the video.

The simple way to solve this problem can be to reduce the difference between the stylized frames(Xu, Xiong and Hu, 2021). The problem will arise when the video has the motion in the objects that will blindly minimize the distance between the two adjacent stylized frames and the effect of the motion will not be considered. To solve this problem, the optical flow of objects between the two adjacent frames should be considered.

Optical flow is calculated here using the presence of dense, accurate optical flow files describing the motion between two adjacent frames. The forward optical flow will be used to estimate how the frame x_{t-1} at time step t-1 might look like by warping the frame x_t at time step t.

Warping the frame means the process in which the location of the pixels of the input frame is changed by using the information provided by the optical flow estimation. Since the optical flow estimation gives information about the motion of the objects in the frame of the video, it will inform the direction in which the pixel will be moving and how strong the movement will be in the next frame. The warping function uses this information for shifting the pixel location and estimating what the next frame might look like.

We warp the stylized frame y_{t-1} at t by warping function and using the optical flow estimation between the frames x_{t-1} and x_t. The warped frame includes the motion

information which can be used to compare with the stylized frame y_t at time t+1 and help to calculate the temporal loss function(5.3).

When an element in one frame is not visible and suddenly appears in the next frame, this is called occlusion. This element can either be blocked by another object from a point of view or result of motion discontinuities. This dis-occlusion region doesn't have any reference point in the previous frames for comparison and we want to exclude these regions when calculating the temporal loss function(Sundaram, Brox and Keutzer, 2010).

Before removing the dis-occluded regions from the frames, we have to take luminance differences into account. There can be luminance and colour differences, in real-world videos, on the same object in adjacent frames due to the luminance effect. For such cases, the assumption of bright consistency constraint in direct optical flow warping(Dosovitskiy et al., 2015; Weinzaepfel et al., 2013; Ilg et al., 2017) fails(Horn, 1974). Animated videos can also give a difference in luminance and colour appearance when they use albedo pass to find the ground-truth optical flows and later include smooth shading and reflections in the final images. To encourage the same luminance change on the adjacent frames of videos, we add a luminance constraint to the temporal loss function.

For luminance change, we have to take the temporal warping error in both XYZ and RGB colour space. From Gao et al. (2018), we can deduce that XYZ and RGB colour space share the same warping error distribution. In XYZ space, the Y channel is taken as a relative luminance channel and X and Z represent the chromaticity and both contribute to the inter-frame difference. Here we are considering only relative differences to check the stability of the frames. The relative luminance is given as :

$$Y = 0.2126R + 0.7152G + 0.0722B$$

in XYZ colour space(Vaughan, 2021). The relative luminance will help stylize real-world videos better. For stylizing the animated videos, we need to use different values for relative luminance. It is given as:

$$Y = 0.2989R + 0.5870G + 0.1140B$$

This is added as a warping constraint for all channels in RGB colour space.

Let x_{t-1} and x_t be the input frames at time t-1 and t respectively, y_{t-1} and y_t be their stylized output frames. W_t is the ground truth forward optical flow from frames at t-1 and t, and M_t is the ground truth forward occlusion mask. The temporal loss function is as follows:

$$Loss_{temp} = \sum(1/D)M_t||(y_t - W(y_{t-1})) - (x_t - W(x_{t-1}))||^2)$$

where D is the product of dimensions of the input image frame.

While the temporal loss between the two adjacent frames doesn't guarantee long-term consistency. This means enforcing consistency in longer intervals. To improve the stylization and maintain long-term temporal consistency, I am using the long-term temporal loss function(Lai et al., 2018) which will penalize the deviation from a more distant frame.

This will calculate the temporal loss between two frames with long intervals, say frame at time t and frame at time t-5. The long-term temporal loss function can be calculated as:

$$Loss_{longtemp} = \sum(1/D)M_t||(y_t - W(y_{t->5})) - (x_t - W(x_{t->5}))||^2)$$

4.5 Total loss and Weights

After calculating all the losses, tallying the total loss is a simple task. We choose weights for all the losses we are using here.

α and β are the weights for content loss and style loss respectively. A higher α value prioritizes the network for reducing $Loss_{content}$ and the same goes for style loss weight, if β is higher, it prioritizes the network for reducing $Loss_{style}$. To get a better stylization result which has style as well as content from the respective images, $Loss_{content}$ is likely to be greater in magnitude than $Loss_{style}$, so we increase the β value so that both losses are roughly equal.

γ, λ_{st} and λ_{lt} are the weights for total variation loss, temporal loss and long temporal loss respectively. More detail on these weights is given in Section 5.2.

The total loss is given as:

$$Loss_{total} = \alpha * Loss_{content} + \beta * Loss_{style} + \gamma * Loss_{tv} + \lambda_{st} * Loss_{temp} + \lambda_{lt} * Loss_{longtemp}$$

Chapter 5

Implementation and Testing

5.1 Data preparation

The dataset used for the experiment is MPI Sintel Dataset. This dataset has all the required files used for the method. More detail on this is given in Section 3.1. The content input image dataset used here is taken from the clean folder in the dataset. The occlusion mask of the content image is taken from the occlusions folder and the optical flow file is taken from the flow folder which gives the pixel values of the movement of objects in the matrix. The implementation of the code is done in PyTorch of version 1.12.1 with CUDA 11.3.

We take 2 adjacent images for the method which will help in maintaining the temporal consistency. *img1* is the image at time t and *img2* is the image at time t-1. The mask and flow files are of *img1*. The first frame is stylized independently as we don't have a previous motion file of it. Both the image array and mask array are resized using Bilinear interpolation with size 512x 512. We use bilinear interpolation as it uses 4 nearest neighbours to generate a smooth output surface. Style image is resized to the same size as the content images with bilinear interpolation.

After reading all the files from the dataset, we store all the data using dataloader will help train the model easily. The dataloader is iterable that keeps the data in mini-batches, reshuffles the data at every epoch and reduces model overfitting. Now, we have an iterable which has *img1* at time t, *img2* at time t-1, mask and flow of *img1*.

5.2 Stylization

To determine the number of epochs chosen for training the model, we train the model for 100 epochs by assuming the loss weights from Gao et al. (2018) with Van Gogh's "Starry Night" style image and check the stylized output with the increasing number of epochs. This shows that model is improving after a certain number of epochs. Figure 5.1 shows the output the model is giving at different epochs. At the first epoch, the model is giving a gibberish image for the first image, the model is giving a little bit better output at the third epoch but the content is still not clear and it can be seen that the output keeps improving with the increasing number of epochs. The model is giving good output at

epoch 60 and it gives similar output for later epochs. So I am training the model for 60 epochs.

Figure 5.1: Output at different epochs

We need to find the optimizer for the model to be trained. Optimizers are the method to minimize the loss function and increase the efficiency of the model. These are dependent on the parameters of the model like the weights and biases. The weights and the learning rate can be changed using the optimizer and this helps to reduce the loss in the neural network. The graph(Figure 5.2)shows the loss obtained from the two optimizers, that are Adam and AdaMax. We see that AdaMax gives a minimum loss function for the neural network compared to Adam. Also, we don't have to focus too much on choosing the learning rate for Adam and Adamax and it is updated on its own for each epoch.

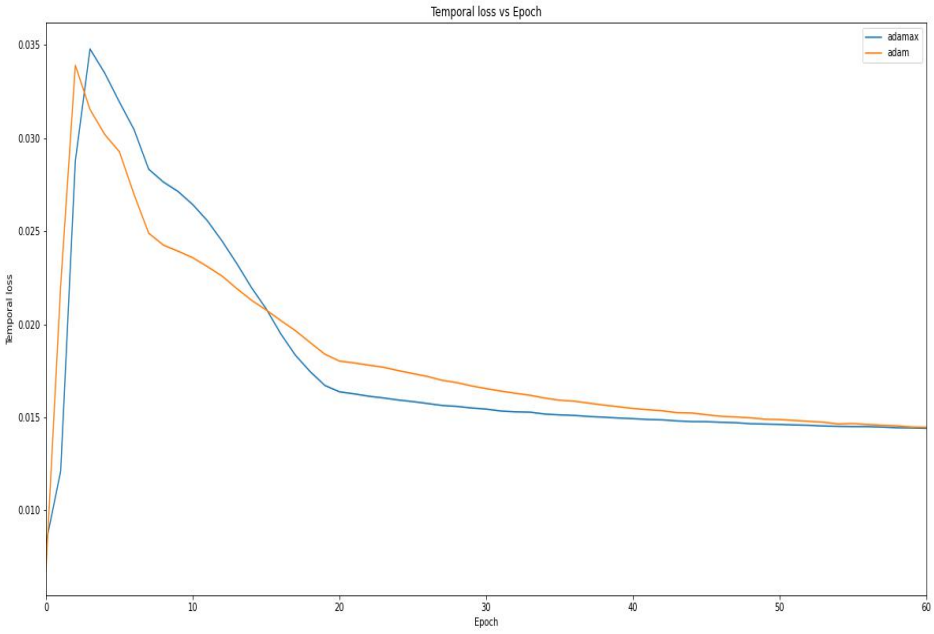

Figure 5.2: Adam vs AdaMax optimizers

The weight for content, style, total variation loss, temporal loss and long temporal loss collectively are responsible for the stylization of the image as well as maintaining the temporal stability of the optimized model. Total variation loss has a meaningful impact on the smoothness of the output, but it does not contribute to the stability of the output. So for now, we are not considering it for determining other weights. We need to consider

the case where we don't get the stable output but poor stylization or good stylization, we need to balance the temporal losses with the perceptual losses.

The evaluation of temporal losses, style and content loss is done by using a different set of parameters for 60 epochs. The temporal loss weights for both short and long temporal errors are ranging from 10 to 100. The content loss weight is chosen as 1 because it is bigger than the style loss weight and to balance them, the style loss weights are ranging from 10^4 to 10^5.

The below Table 5.1 shows the loss value for different combinations of loss weights.

Table 5.1: Loss Weight

λ_{st}	λ_{lt}	α	β	Temporal loss	Long temporal loss	Content Loss	Style Loss
10	10	1	10^4	0.0004767	0.0023654	13.15	0.0004993
10	100	1	10^4	0.0004494	0.0015406	14.85	0.0004313
100	10	1	10^4	0.0002522	0.0010092	18.15	0.0006853
100	100	1	10^4	0.0005467	0.0018495	10.82	0.0003616
10	10	1	10^5	0.0035184	0.0018208	32.27	0.0001166
10	100	1	10^5	0.0024016	0.0026946	28.37	0.0001167
100	**10**	**1**	**10^5**	0.0021918	0.0020594	25.35	0.0001065
100	100	1	10^5	0.0018925	0.0020170	27.14	0.0001323

From Table5.1, we can see that if the style loss is taken as 10^4, we get less temporal loss but the style loss is increased impeding the performance of style transfer. If we see values when the style weight is 10^5, the bold row in the table gives the minimum value for all the losses. So the weights used in the proposed method are $\alpha = 1$, $\beta = 10^5$, $\gamma = 10^{-6}$, $\lambda_{st} = 100$ and $\lambda_{lt} = 10$.

The model is trained here for 20 epochs, with a batch size of 2. The small batch size will help the model learn fast with better style transfer in the content image. The AdaMax optimizer is used in the method with a learning rate of 0.0001.

When an image is passed to the style network. The style network is used to transform the image. The encoder will increase the size of the image meanwhile the image filter will detect the edges of the content image and maps the residues of the style texture, then the decoder will reduce the image to its original size. At first, the network is randomly initialized, so it is like the first image given to the network is transformed into a gibberish image. The output image from the style network, the content image and the style image will then be passed to the loss network which will calculate the losses. With more data passing to the network with the increasing number of batches in an epoch, the style network will learn and be able to give better stylized output as the resulting tensor is concatenated with the edge-only image and style features. The style network being an optimization-based method helps to generate better output.

After deciding all the parameters for training a model. The two adjacent input frames are given to the style network, and then the output of both the image frames is given to the VGG loss network along with both the input image frames and the style image to calculate the losses(Section 5.4). The visual representation of the whole proposed design is given in Appendix A.

5.3 Calculating Temporal Loss

To find the temporally stable stylized video, we first need to warp the input frame at time t. The input frame is warped using the ground-truth optical flow of the image. We also find the mask boundary of the image from the optical flow file. The mask boundary is the boundary of the mask where the motion of any object is changed.

To retain the unchanged places from the current and previous image frames, the mask of the current frame should eliminate the changed places. Before this, first, we take the luminance change of traceable pixels into account and add a relative luminance constraint. We need to train two models separately for the real world and digital videos with different relative luminance values. The difference between two adjacent images will give the optical flow image. This optical flow image is compared with the mask of the current frame to remove the untraceable pixels. The value of mask(M) varies from 0 to 1. For traceable points/regions by the previous frame, like static background, the value in the mask tends to be 1. On the contrary, at occlusion or false flow points/regions, the value in the mask is 0.

We find the warped image from the stylized current frame. The mean squared error of the stylized warped image with the previous stylized image is taken to calculate the temporal loss function.

Figure 5.3 shows a visual representation of how the temporal loss function is calculated.

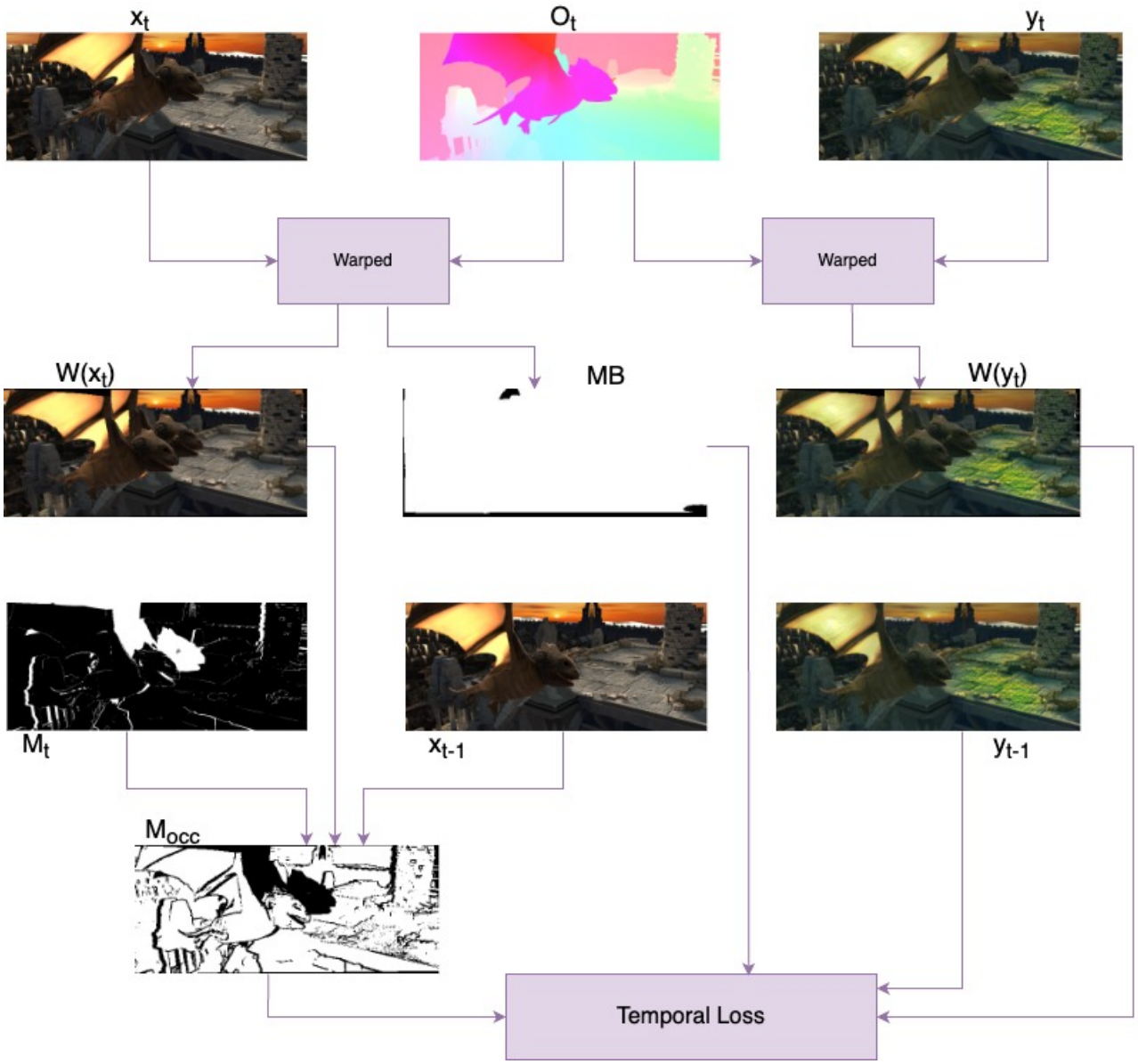

Figure 5.3: Temporal Loss Calculation. Taking an example from MPI Sintel dataset. x_t is the current content image, and O_t is the ground truth optical flow. y_t is the stylized output of image x_t. x_{t-1} is the previous content imagey_{t-1} is the stylized output of x_{t-1} at time t-1

One limitation of the temporal loss between two adjacent input frames can be that the stability cannot be maintained in a long interval of time for the video when some area in the frame is static. To keep the stability for long intervals in the frames of the video, we use the long temporal loss function.

The long temporal loss function follows the same process as the temporal loss function. The only difference is that instead of taking the previous frame for calculating the temporal loss, we take the previous frame at a longer interval. We save every 5^{th} warped image and then calculate the temporal loss function for every frame with that 5^{th} frame.

5.4 Total Loss calculation

When we add an input image to train in the neural network, the autograd system of the library(PyTorch) is used to compute all the gradients. All the losses are calculated and the weights are then updated so that we can minimize the total loss.

The stepwise gradient descent process can be summarized as:

1. Style Loss: Pass the output images of two adjacent frames in the VGG-16 model to the first convolutions of deep layers from 1 to 4 to extract feature maps and calculate style loss(code in Appendix D).
2. Create Gram matrices for style feature maps using (1) and calculate the mean squared error with the gram matrix of two target images separately.
3. Add the style loss of two adjacent frames. Multiply the style loss with the style loss weight and add to the total loss.
4. Content Loss: Pass the output images of the adjacent frames to the third convolution of deep layer 3 of VGG-16 which will extract the feature maps and calculate the content loss.
5. Calculate the mean squared error of the feature map from (4) for both the content image separately.
6. Add the content loss of two adjacent frames. Multiply the content loss with the content loss weight and add to the total loss.
7. Total Variation Loss: Calculate the total variation loss of both the output images using a standard regularization term on high-frequency artefacts.
8. Sum the total variation loss of the two frames and multiply the total variation loss with the total variation loss weight and add to the total loss.
9. Multiply the temporal loss(5.3) with the temporal loss weight and long temporal loss with the long temporal loss weight. Finally, add both losses to the total loss.
10. Backpropagation: Calculate the gradients of pixel values in the direction that decreases the total loss computed from (9) and update the pixel values.
11. Repeat steps (1) – (10) until convergence

5.5 Testing

To test the output from the trained model, I am taking a series of video image frames. Load_state_dict function from Pytorch is used to load the model parameters dictionary, like weights and biases. This dictionary object maps the parameter tensor to each layer in the model.

Once the model is trained, it will take the edges from the input image and the style features and combine them. Each of the frames is passed into the stylizing network which gives the stylized output frames of the images. These images are combined to form a video using the OpenCV function VideoWriter. The stylized output of two adjacent frames can be seen in Figure 5.4. The long-term consistency can be seen in Figure5.5.

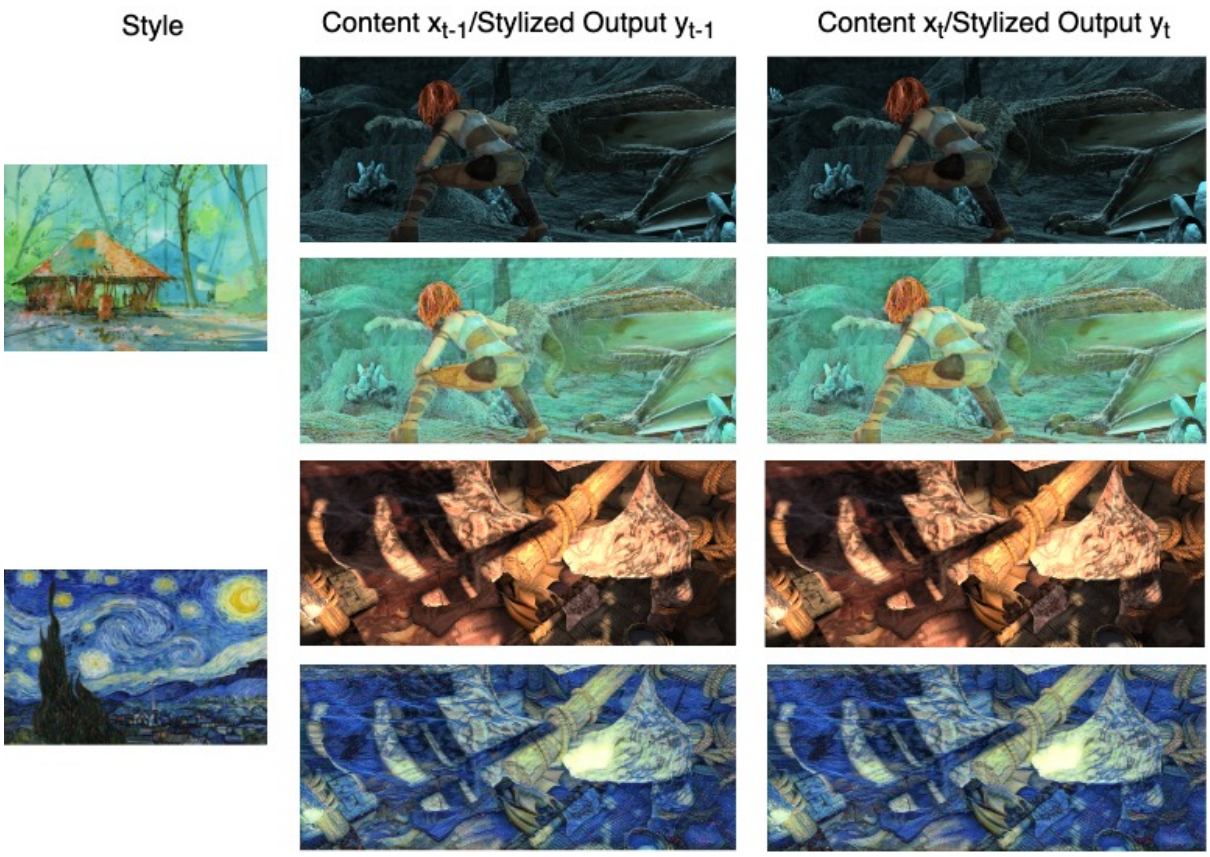

Figure 5.4: The input images are taken from MPI Sintel Dataset. The style images are Konkan Hut and VanGogh respectively.

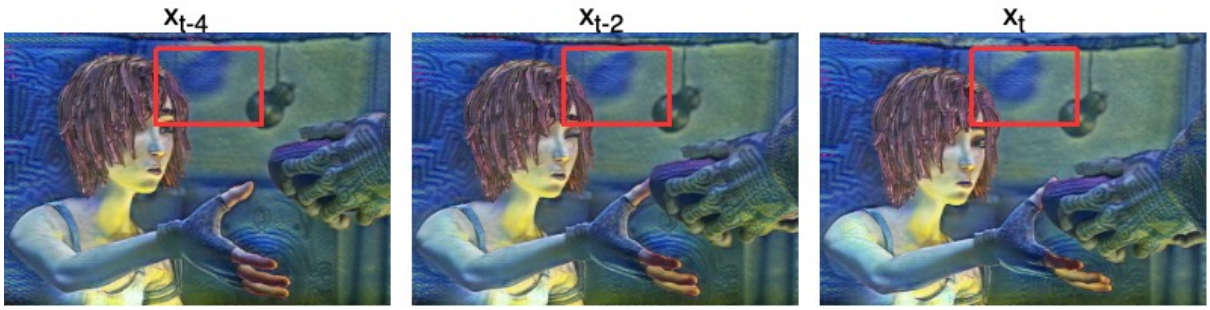

Figure 5.5: Long Temporal Consistency Demonstration

The code of this implementation is given in Appendix D and more outputs of the method are given in Appendix C.

Chapter 6

Results

From section 5.5, we can see the output we are getting. In this chapter, we will further analyse the result of the stylized video and lay out the path for future improvement (discussed in Section 7.1).

The content loss, style loss and temporal loss values are plotted for each epoch. Figure 6.1-6.3 shows the different graphs.

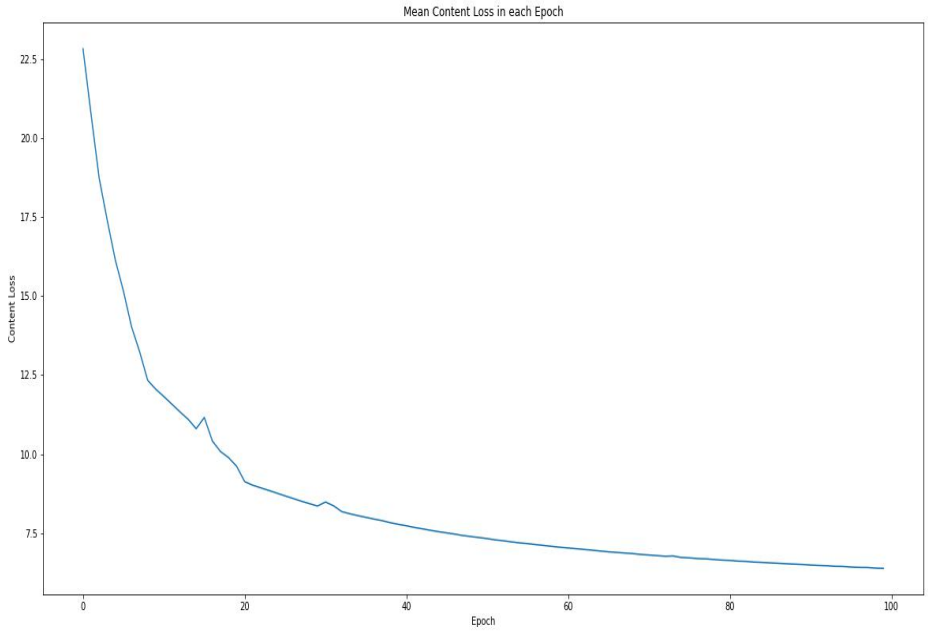

Figure 6.1: Content Loss

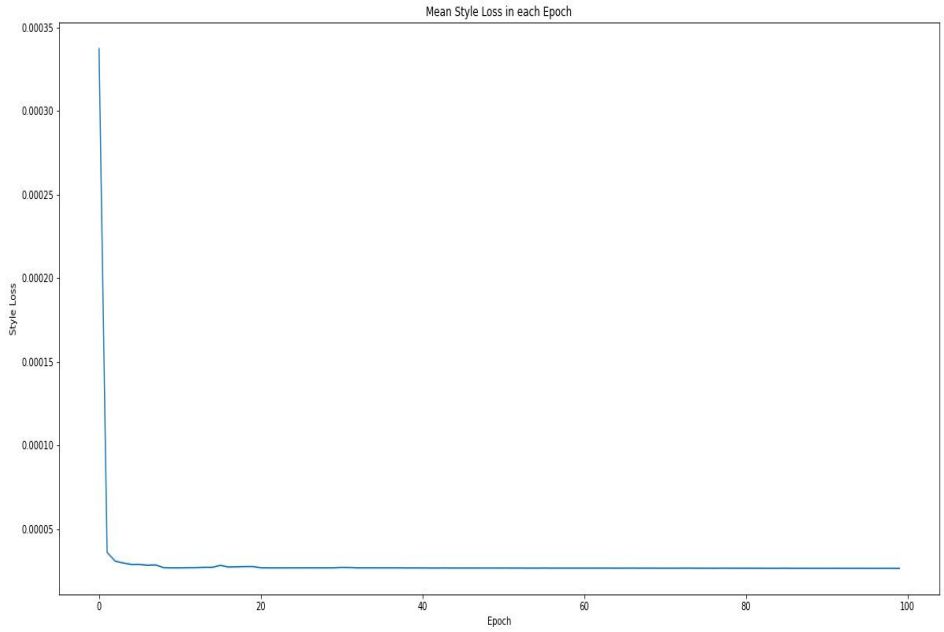

Figure 6.2: Style Loss

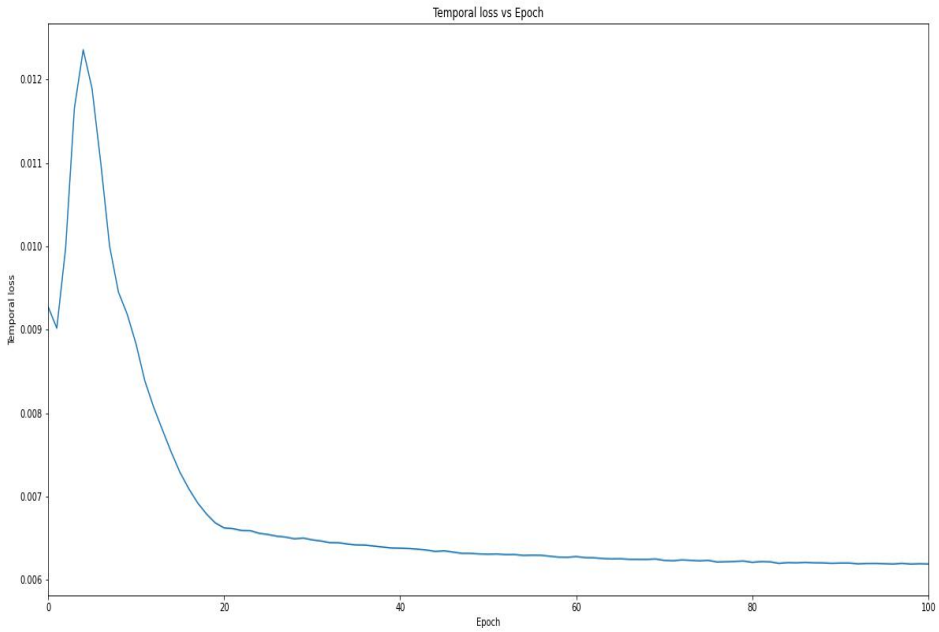

Figure 6.3: Temporal Loss

It can be seen that the content loss converges after 35 epochs, the style loss converges much before 20 epochs and temporal loss converges after 20 epochs. To get a better output, the model was trained will 60 epochs as content loss difference is reduced after

that.

I am comparing my method with two state-of-the-art approaches for neural style transfer of images and videos.

1. Baseline1: I am taking the method used proposed by Johnson, Alahi and Fei-Fei (2016). The feed-forward network is trained to stylize images and apply the result of the model to the video image frames separately. This gives the high-quality stylized output but leads to flickering in the video.
2. Baseline2: Next method, I am taking method by Dougherty (2020). This method uses an explicit function for stabilizing the video frames by adding noise to the video frames which gives stability to the video.

6.1 Quantitative Evaluation

6.1.1 Speed

The previous methods like Ruder, Dosovitskiy and Brox (2016); Huang et al. (2017); Chen et al. (2017) are very slow to stylize the video frames and can take a few hours to give a stylized output. The method proposed can produce a stable stylized video from the image frames faster. The stylized take a similar amount of time as the video produced from Baseline1 and Baseline2. The proposed method does not explicitly compute the optical flow while testing. It can be seen that the method can produce stylized video with the image frame size 256 X 256 with 47 frames per second(fps) and can produce an output from the image frame of size 512 X 512 at a speed of 10 fps in Nvidia Tesla P100 GPU.

6.1.2 Video Stability

The aim is to show that the stylization done by the method can produce a stable video. The instability of the baseline1 method for image stylization manifests strongly in the background region of the image with relatively little motion.

To check for stability in the real-world video, we use the SSIM score. It stands for structural similarity index measure. It is a perceptual metric used for measuring the similarity between two images. This will help us compute a score between two adjacent frames of the video on how well the style is rendered between them.

I am taking a video sequence from a real-time video. For each frame at time t, we find a patch of size 100 x 100 pixels and a corresponding patch of the same size from the frame at time t+1. We then calculate the SSIM score between the two frames and average the SSIM score for all the frames. The SSIM score of two styles across two video sequences are given in Table 6.1.

Table 6.1: SSIM Score

Video-Style	Baseline1	Baseline2	Proposed Method
Scenary-wave	0.40	0.59	0.66
Woodpecker-vangogh	0.68	0.85	0.88

The SSIM score for two adjacent video frames is shown in Fig 6.4. From this, we can see that Baseline1 (Johnson, Alahi and Fei-Fei, 2016) shows instability with the lowest score in all three methods. It can be seen that the proposed method gives the highest SSIM score among all three methods.

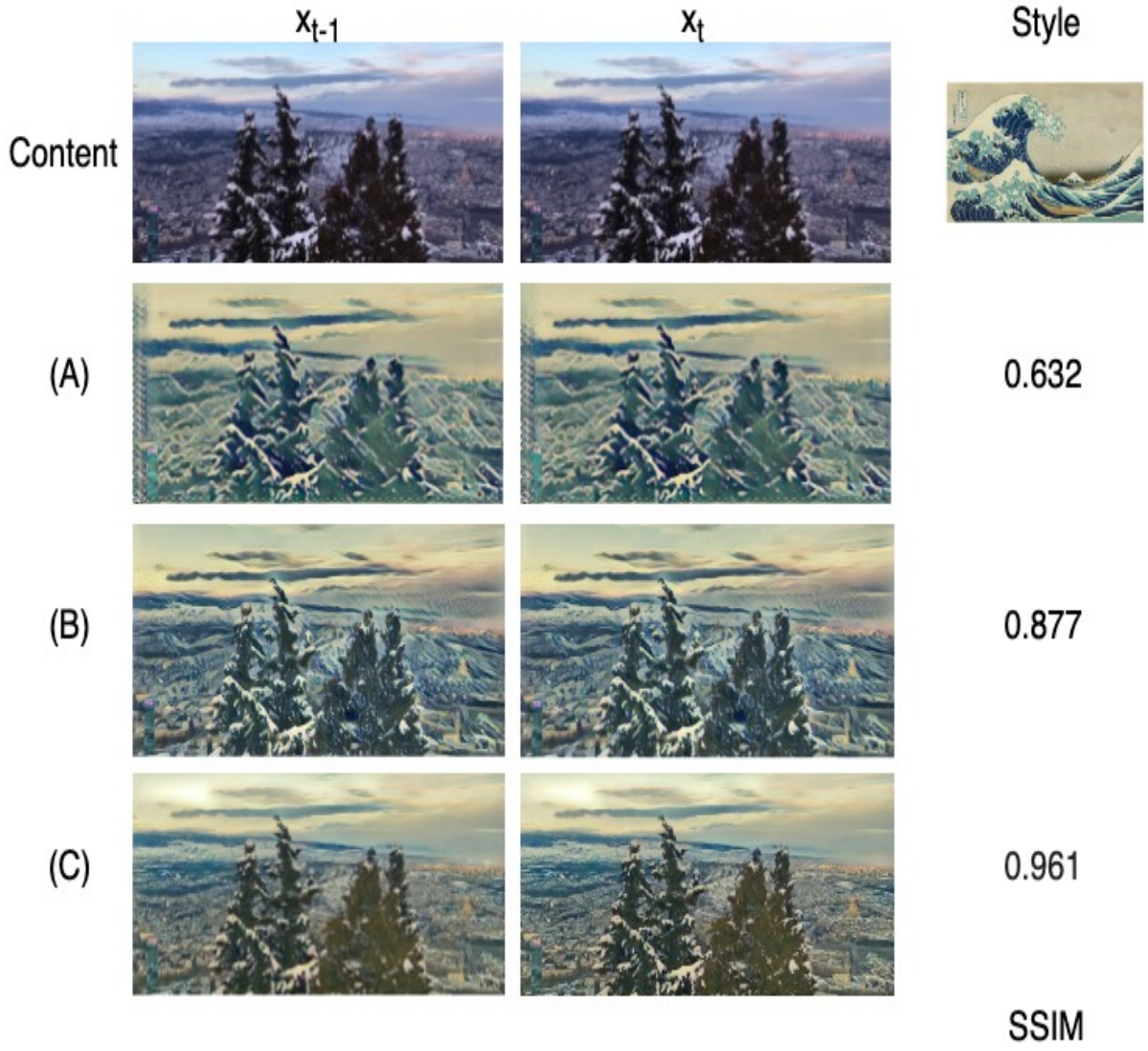

Figure 6.4: SSIM score for two adjacent frames. The content image is scenery and the style image is wave.

Another evaluation matrix used for digital videos was to calculate the temporal error using the formula:

$$E(V) = \sqrt{(1/T-1)\sum_{t=1}^{T}(1/D)M_t||y_t - W_t(y_{t-1})||^2}$$

where E(V) is the temporal error, T is the number of frames and D is the dimensions of an image frame.

The values obtained by using this evaluation matrix for style Candy are shown in Table6.2.

Table 6.2: Temporal Stability Error

Video	Chen et al. (2017)	Proposed Method
Alley-2	0.0934	0.0821
Market-6	0.1030	0.0801

Table 6.2 shows that the temporal error obtained by the proposed method is less than Chen et al. (2017). Therefore making the proposed method more stable.

6.1.3 Content Leak

It can be seen from Figure 6.4 that both the baseline methods make the video sequence corrupt in terms of content after stylization. Compared to the two baseline methods, there is a minimum content leak in the proposed method. To check this content leak, I used the structural similarity index(SSIM) to check the similarity between the content image frames and the stylized image frames and averaged the SSIM score. The method proposed gives the highest SSIM score with 52% similarity whereas the baseline2 method gives 40% SSIM score.

6.2 Qualitative Evaluation

6.2.1 Stability with Temporal Loss

When testing the video sequences after stylization with different styles, it can be seen that the proposed method gives similar stylization for the two adjacent frames, which eventually helps in producing a stable stylized video. The Baseline1 method is not able to produce a video sequence with the same stylization in the adjacent video frames. These changes in the stylization of the adjacent frames will result in inconsistent stylization in the video, causing a flickering effect in the video.

Figure 6.5 shows the consistency example in the proposed method compared with the baseline1 method.

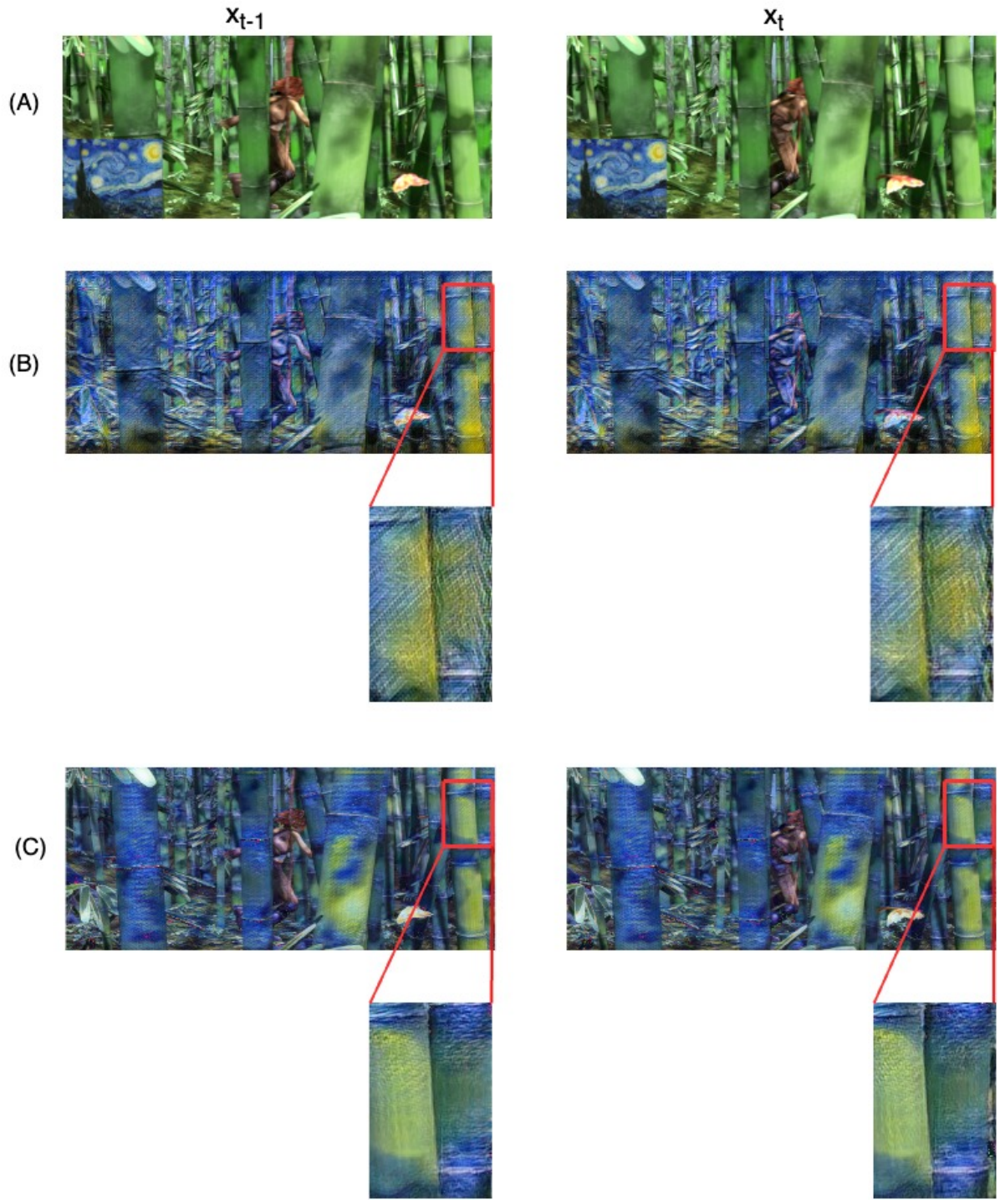

Figure 6.5: (A) shows the input images and style image VanGogh. (B) shows the inconsistency in style transfer in two consecutive images using Baseline1. (C) shows consistent style transfer by the proposed method.

6.2.2 Stability with Long Temporal Loss

Figure 5.5 shows an example of the long-term stability in the video where the object is stylized the same as it was done in the previous frame. The long temporal loss helped in maintaining the long-term loss. And it can be seen that when an object is occluded and reappears again in the later frames, the stylization of that object is similar. Figure 6.6

shows the image is stylized the same when the object is dis-occluded after being occluded in the video frame.

Figure 6.6: Three consecutive video frames are taken here, the man gets occluded in image x_{t-1} and dis-occluded in x_t. These images represent that the stylization is similar in x_{t-5} and x_t.

We can observe that the proposed model can successfully reproduce the texture of the artistic image and the stylized frames are visually coherent. The stylisation process is also fast, making it useful for real-time problems. Better texture of the artistic image can be transferred when more images are taken to train the model. Further discussed in Section 7.1.

Chapter 7

Conclusions

The initial hypothesis of the study was to use the optical flow method to produce a temporally stable stylized video. We present an improved feed-forward convolutional neural network called ReCoNet for video style transfer. The model is able to produce a stable stylized video sequence which is faster to process and can be used in real-time. From the qualitative and quantitative results from section 6, we can observe that the proposed method is able to achieve stability in the video. Long-term consistency is also introduced which ensures the stability of the object in long term. This long-term consistency maintains real-time performance.

The temporal loss function adds a luminance warping constraint to the adjacent frames which deal with the luminance difference in the colour space of the two frames. This helps in the better stylization of the video sequences and maintaining stability.

We also provide evaluation metrics used to check the stability of the video. The evaluation metrics proposed in this study use Structural Similarity Index which calculates a score indicating how similar two adjacent frames are in the video sequence. This shows if the style transferred is consistent in the sequence of video frames. The proposed method also gives a more stable stylized output compared to the baseline methods.

Neural Style Transfer is becoming popular in entertainment and social communication. The styling tools that enhance digital art from neural networks are a field of interest nowadays. Also, there are many applications for producing consistent videos. However, the real strength of such a technique can go beyond image and video creation. This technique can be used in medical, defence and different sectors where AI can learn the style of a person doing the task and try to copy it.

7.1 Limitations and Future Work

One of the limitations of the proposed method is that the model can train only one style at a time and the model had to be trained every time for different styles. Although training the model for one style reduced the training time even after including the temporal loss function which uses optical flow. The methods like AdaIN Huang and Belongie (2017), WCT Li et al. (2017a) need to train the model only once with different style images and can stylize the image sequence with any style image or a new style image which is not been trained but these methods take a lot of time for training the model and training can even be slower with optical flow. The Artflow method An et al. (2021) can be used to get the stylized video sequence from any style image which if using the temporal loss function gives a stable video with a high SSIM score.

Another limitation is that we need to train two models for real-world videos and animated videos separately with different relative luminance values to get a better result. The style transfer method can further be improved if we adjust the constraint of depth Liu et al. (2021) in the neural network. This will not improve the consistency of the video but can increase the texture synthesis in the videos.

To increase the performance of the model, we can use a larger and more varied dataset. Using the larger dataset can help the model learn to handle the varied type of input video sequences and make the model more robust. Despite the relative changes in the luminance channel of the colour space, the chromaticity also contributes to the inter-frame differences. The possibility of using the chromaticity and luminance difference in the warping of the images can improve the stability of the style transfer algorithm.

Bibliography

An, J., Huang, S., Song, Y., Dou, D., Liu, W. and Luo, J., 2021. Artflow: Unbiased image style transfer via reversible neural flows. *Proceedings of the ieee/cvf conference on computer vision and pattern recognition*. pp.862–871.

Baldi, P., 2012. Autoencoders, unsupervised learning, and deep architectures. *Proceedings of icml workshop on unsupervised and transfer learning*. JMLR Workshop and Conference Proceedings, pp.37–49.

Butler, D.J., Wulff, J., Stanley, G.B. and Black, M.J., 2012. A naturalistic open source movie for optical flow evaluation. In: A. Fitzgibbon et al. (Eds.), ed. *European conf. on computer vision (eccv)*. Springer-Verlag, Part IV, LNCS 7577, pp.611–625.

Chen, D., Liao, J., Yuan, L., Yu, N. and Hua, G., 2017. Coherent online video style transfer. *Proceedings of the ieee international conference on computer vision*. pp.1105–1114.

Collomosse, J.P., Rowntree, D. and Hall, P.M., 2005. Stroke surfaces: Temporally coherent artistic animations from video. *Ieee transactions on visualization and computer graphics*, 11(5), pp.540–549.

Davis, 2018. DAVIS: Densely Annotated VIdeo Segmentation. `https://davischallenge.org/`. [Online; accessed 18-August-2022].

Delanoy, J., Bousseau, A. and Hertzmann, A., 2019. Video motion stylization by 2d rigidification. *Expressive 2019-8th acm/eurographics proceedings of the symposium*.

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

Dosovitskiy, A., Fischer, P., Ilg, E., Hausser, P., Hazirbas, C., Golkov, V., Van Der Smagt, P., Cremers, D. and Brox, T., 2015. Flownet: Learning optical flow with convolutional networks. *Proceedings of the ieee international conference on computer vision*. pp.2758–2766.

Dougherty, T., 2020. Stabilizing neural style-transfer for videos with PyTorch. `https://thomasdougherty.ai/pytorch-video-style-transfer/`. [Online; accessed 25-August-2022].

Dumoulin, V., Shlens, J. and Kudlur, M., 2016. A learned representation for artistic style. *arxiv preprint arxiv:1610.07629*.

Fukushima, K. and Miyake, S., 1982. Neocognitron: A self-organizing neural network

model for a mechanism of visual pattern recognition. *Competition and cooperation in neural nets*. Springer, pp.267–285.

Ganegedara, T., 2019. Intuitive Guide to Neural Style Transfer. `https://towardsdatascience.com/light-on-math-machine-learning-intuitive-guide-to-neural-style-transfer-ef88e46697e` [Online; accessed 19-August-2022].

Gao, C., Gu, D., Zhang, F. and Yu, Y., 2018. Reconet: Real-time coherent video style transfer network. *Asian conference on computer vision*. Springer, pp.637–653.

Gatys, L., Ecker, A.S. and Bethge, M., 2015. Texture synthesis using convolutional neural networks. *Advances in neural information processing systems*, 28.

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

Gross, S. and Wilber, M., 2016. Training and investigating Residual Nets. `http://torch.ch/blog/2016/02/04/resnets.html`. [Online; accessed 18-August-2022].

Hertzmann, A., 1998. Painterly rendering with curved brush strokes of multiple sizes. *Proceedings of the 25th annual conference on computer graphics and interactive techniques*. pp.453–460.

Hertzmann, A., Jacobs, C.E., Oliver, N., Curless, B. and Salesin, D.H., 2001. Image analogies. *Proceedings of the 28th annual conference on computer graphics and interactive techniques*. pp.327–340.

Horn, B.K., 1974. Determining lightness from an image. *Computer graphics and image processing*, 3(4), pp.277–299.

Hosni, A., Rhemann, C., Bleyer, M. and Gelautz, M., 2011. Temporally consistent disparity and optical flow via efficient spatio-temporal filtering. *Pacific-rim symposium on image and video technology*. Springer, pp.165–177.

Huang, H., Wang, H., Luo, W., Ma, L., Jiang, W., Zhu, X., Li, Z. and Liu, W., 2017. Real-time neural style transfer for videos. *Proceedings of the ieee conference on computer vision and pattern recognition*. pp.783–791.

Huang, L., Wang, P., Yang, C.F. and Tseng, H.W., 2021. Rapid local image style transfer method based on residual convolutional neural network. *Sensors and materials*, 33(4), pp.1343–1352.

Huang, X. and Belongie, S., 2017. Arbitrary style transfer in real-time with adaptive instance normalization. *Proceedings of the ieee international conference on computer vision*. pp.1501–1510.

Ilg, E., Mayer, N., Saikia, T., Keuper, M., Dosovitskiy, A. and Brox, T., 2017. Flownet 2.0: Evolution of optical flow estimation with deep networks. *Proceedings of the ieee conference on computer vision and pattern recognition*. pp.2462–2470.

Johnson, J., Alahi, A. and Fei-Fei, L., 2016. Perceptual losses for real-time style transfer and super-resolution. *European conference on computer vision*. Springer, pp.694–711.

Kyprianidis, J.E., Collomosse, J., Wang, T. and Isenberg, T., 2012. State of the" art": A taxonomy of artistic stylization techniques for images and video. *Ieee transactions on visualization and computer graphics*, 19(5), pp.866–885.

Lai, W.S., Huang, J.B., Wang, O., Shechtman, E., Yumer, E. and Yang, M.H., 2018. Learning blind video temporal consistency. *Proceedings of the european conference on computer vision (eccv)*. pp.170–185.

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

Li, X., Liu, S., Kautz, J. and Yang, M.H., 2019. Learning linear transformations for fast image and video style transfer. *Proceedings of the ieee/cvf conference on computer vision and pattern recognition*. pp.3809–3817.

Li, Y., Fang, C., Yang, J., Wang, Z., Lu, X. and Yang, M.H., 2017a. Universal style transfer via feature transforms. *Advances in neural information processing systems*, 30.

Li, Y., Wang, N., Liu, J. and Hou, X., 2017b. Demystifying neural style transfer. *arxiv preprint arxiv:1701.01036*.

Litwinowicz, P., 1997. Processing images and video for an impressionist effect. *Proceedings of the 24th annual conference on computer graphics and interactive techniques*. pp.407–414.

Liu, X.C., Cheng, M.M., Lai, Y.K. and Rosin, P.L., 2017. Depth-aware neural style transfer. *Proceedings of the symposium on non-photorealistic animation and rendering*. pp.1–10.

Liu, Y., Jiang, A., Pan, J., Liu, J. and Ye, J., 2021. Deliberation on object-aware video style transfer network with long–short temporal and depth-consistent constraints. *Neural computing and applications*, 33(14), pp.8845–8856.

Neumann, A. and Neumann, L., 2005. Color Style Transfer Techniques using Hue, Lightness and Saturation Histogram Matching [Online]. In: L. Neumann, M. Sbert, B. Gooch and W. Purgathofer, eds. *Computational aesthetics in graphics, visualization and imaging*. The Eurographics Association. Available from: `https://doi.org/10.2312/COMPAESTH/COMPAESTH05/111-122`.

Odena, A., Dumoulin, V. and Olah, C., 2016. Deconvolution and checkerboard artifacts. *Distill* [Online]. Available from: `http://distill.pub/2016/deconv-checkerboard/`.

Ranjan, A. and Black, M.J., 2017. Optical flow estimation using a spatial pyramid network. *Proceedings of the ieee conference on computer vision and pattern recognition*. pp.4161–4170.

Ruder, M., Dosovitskiy, A. and Brox, T., 2016. Artistic style transfer for videos. *German conference on pattern recognition*. Springer, pp.26–36.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M. et al., 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3), pp.211–252.

Sheng, L., Lin, Z., Shao, J. and Wang, X., 2018. Avatar-net: Multi-scale zero-shot style transfer by feature decoration. *Proceedings of the ieee conference on computer vision and pattern recognition*. pp.8242–8250.

Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arxiv preprint arxiv:1409.1556*.

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

Ulyanov, D., Vedaldi, A. and Lempitsky, V., 2016. Instance normalization: The missing ingredient for fast stylization. *arxiv preprint arxiv:1607.08022*.

Vanderhaeghe, D. and Collomosse, J., 2013. Stroke based painterly rendering. *Image and video-based artistic stylisation*. Springer, pp.3–21.

Vaughan, B., 2021. How do i calculate luminance of an image in lux measure in matlab?

Videvo, 2018. Videvo: Videvo free footage. `https://www.videvo.net/`. [Online; accessed 18-August-2022].

Wang, W., Xu, J., Zhang, L., Wang, Y. and Liu, J., 2020. Consistent video style transfer via compound regularization. *Proceedings of the aaai conference on artificial intelligence*. vol. 34, pp.12233–12240.

Weinzaepfel, P., Revaud, J., Harchaoui, Z. and Schmid, C., 2013. Deepflow: Large displacement optical flow with deep matching. *Proceedings of the ieee international conference on computer vision*. pp.1385–1392.

Xiao, X. and Ma, L., 2009. Gradient-preserving color transfer. *Computer graphics forum*. Wiley Online Library, vol. 28, pp.1879–1886.

Xu, J., Xiong, Z. and Hu, X., 2021. Frame difference-based temporal loss for video stylization. *arxiv preprint arxiv:2102.05822*.

Appendix A

Design Diagrams

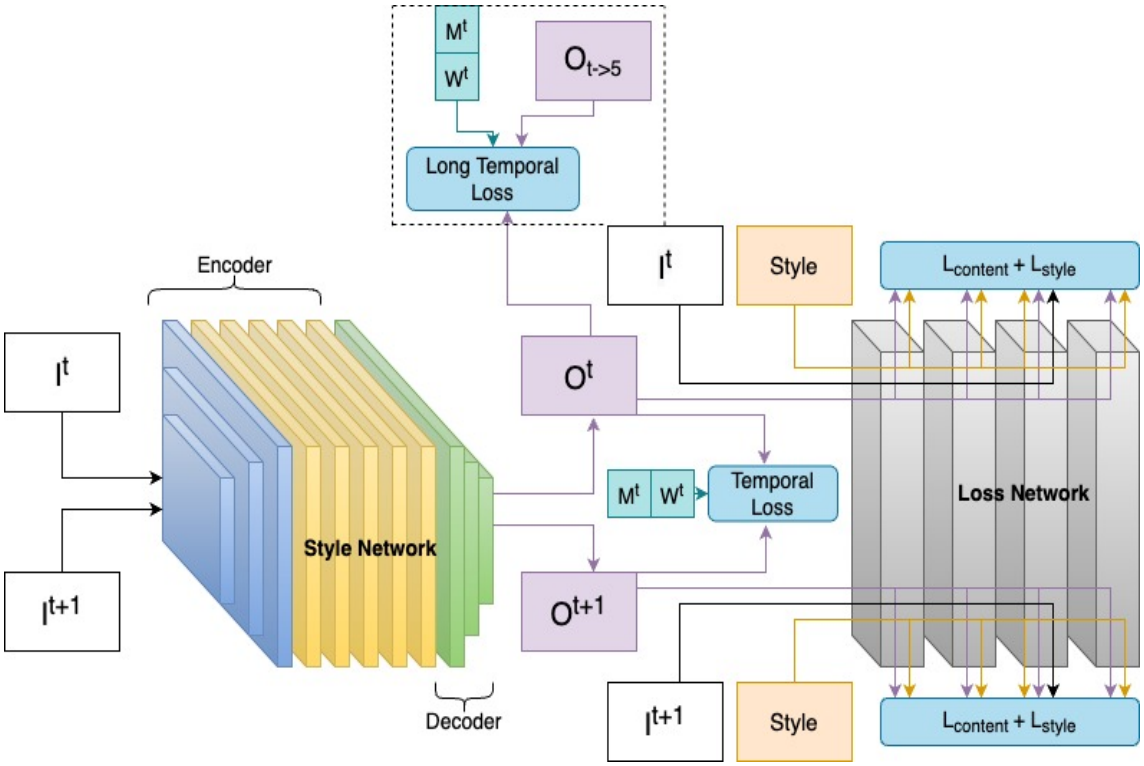

Figure A.1: Design

Appendix B

Glossary

Bilinear interpolation- It is a technique for calculating values of a grid location based on nearby grid cells in an image.

Colour space- Colour space is the organisation of the colours which can be interpreted and displayed on the plane. This interpretation is done using RGB(Red, Green and Blue) chromaticity diagram. When mixing the colours, it can be seen that we get negative plots for primary colours in the graph, to make the computation easy, we linearly transform the colour plot to form a new colour space XYZ where Y perceive luminance and X, Z perceived colours.

Feature map- Each layer in Convolutional Neural Network called a filter generates an output. The output has a mapping of a certain type of features present in the image called a feature map.

Feed Forward network- A Feed Forward Neural Network is an artificial neural network in which the connections between nodes go from one layer to another in a sequential manner and each layer is connected to the previous layer. The nodes do not form a cycle.

Feature space- Every image pixel values constitute the feature space V. One band of the image constitutes a one-dimensional feature space. k bands in an image construct a k-dimensional feature space V_k.

Image Space- Image space refers to the spatial coordinates of an image. Image denoted as I with m x n elements, where m is the number of rows and n is the number of columns in the image. The element in the image space I(i,j) (i = 1, 2,..., m; j = 1, 2,..., n) is called image pixel.

Non-Photorealistic Rendering- Non-photorealistic rendering (NPR) is a technique for animating and representing objects that take their inspiration from paintings, cartoons, and other non-photorealistic sources.

Sobel Gradient - Sobel is the filter used to detect the edges in the image.

Warp - Warping is the process of manipulating an image such that we can take any shape. When we use warping in optical flow, it means that we try to combine two adjacent image frames. A warped image example can be seen in Figure 5.3.

Appendix C

Raw Results Output

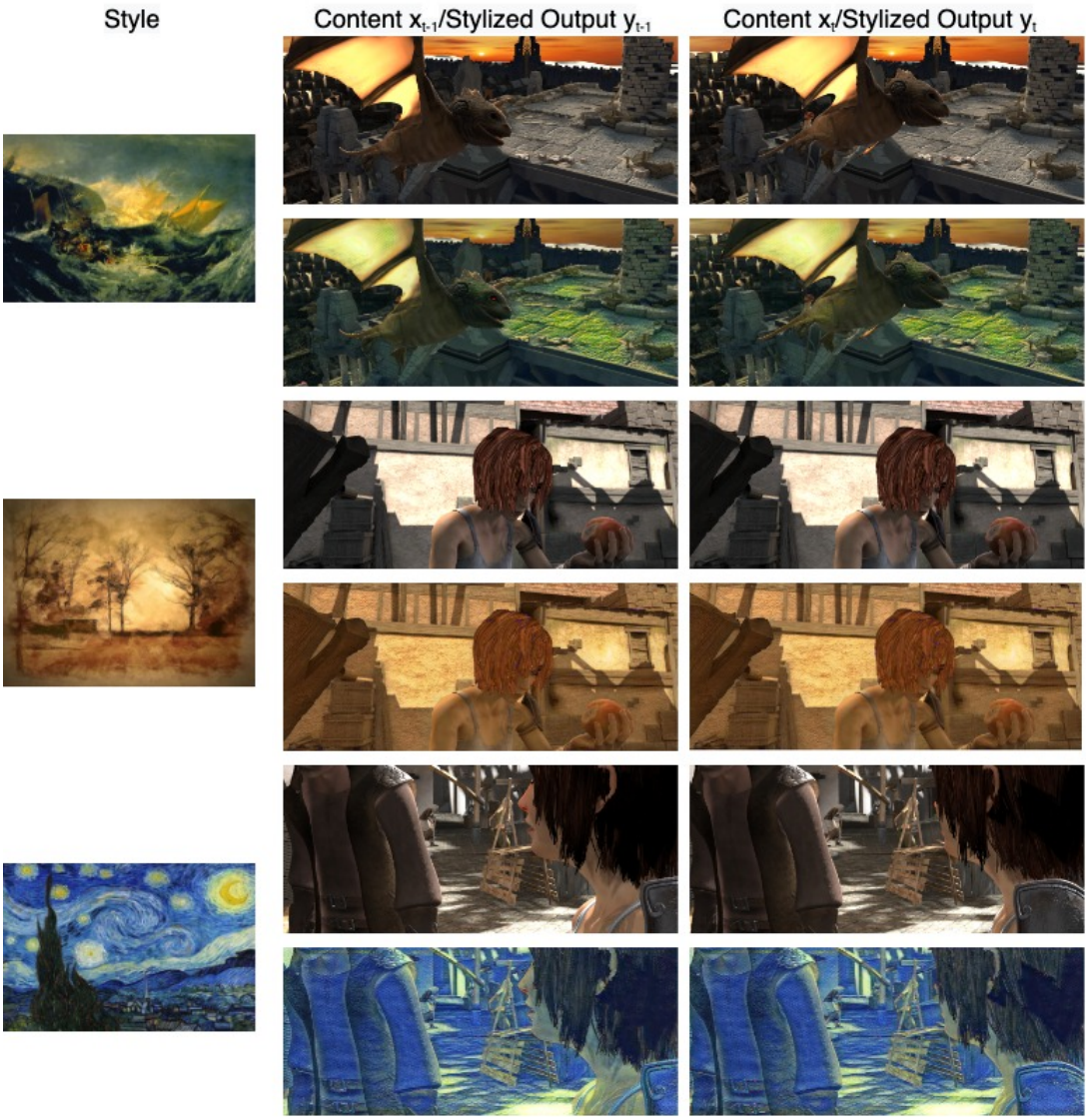

Figure C.1: Output: video frames are taken from MPI Sintel.

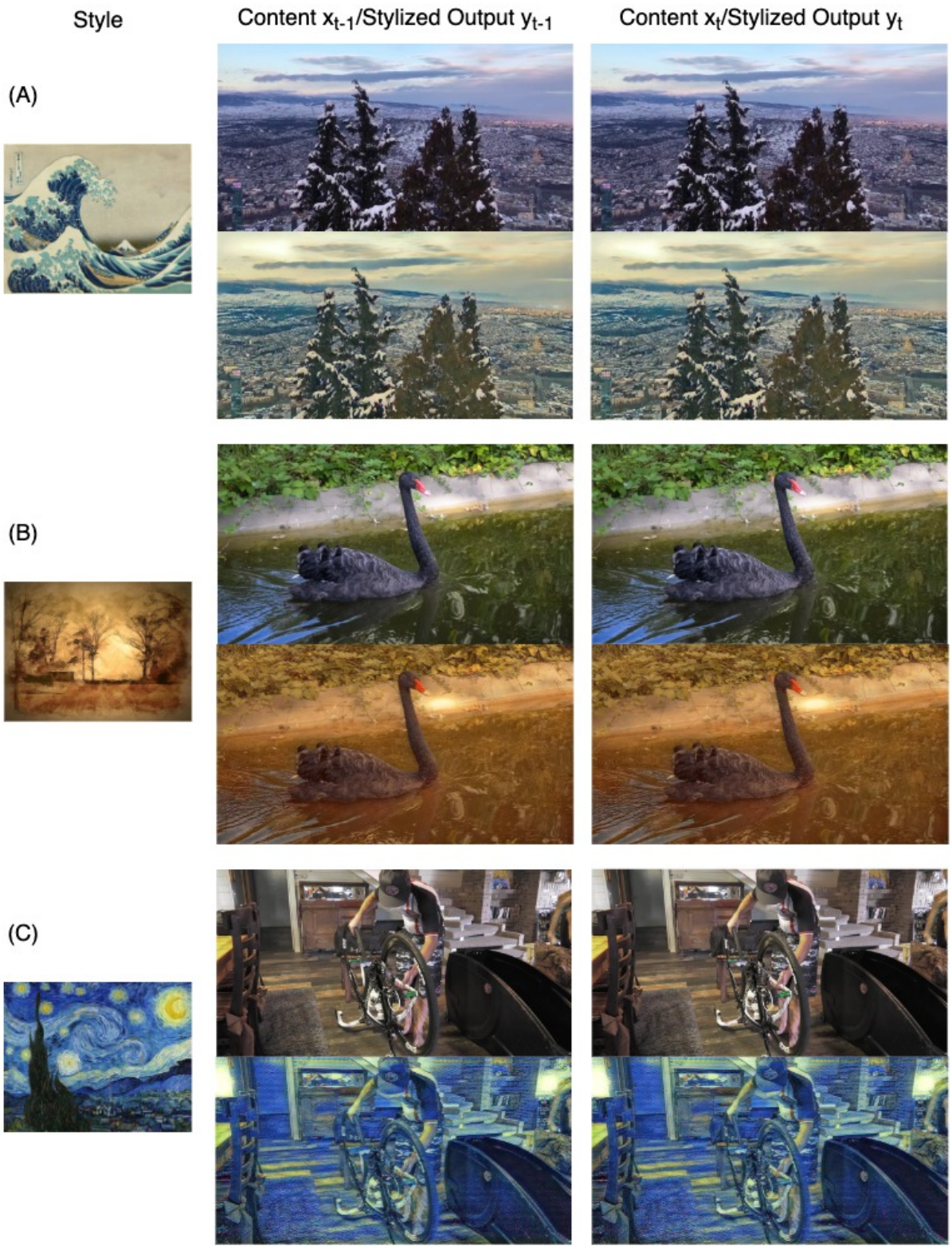

Figure C.2: Output: video frames are taken from Davis and videonet.

Appendix D

Code

The below files shows the code implemented for the proposed method. Below lines of code are the command line arguments given to train and test the model in Google Colab.

Listing D.1: Terminal.ipynb

```
##mount the drive
from google.colab import drive
drive.mount('/content/drive')

##add the code folder
cd /content/drive/My\ Drive/Dissertation/

##training the model
!python main.py train --dataset MPI-Sintel-complete --style_name
    style_images/vanGogh.jpg --epochs 100 --cuda 1

##testing the stylization on video sequence to make a video
!python main.py transfer --test_img_path
    MPI-Sintel-complete/test/clean/bamboo_3 --output_video
    output/vangogh_bamboo.avi --save_model trained_models
    --model-name train.pt --cuda 1
```

D.1 File: main.py

```
import argparse
from train import train
from test import stylize
from evaluate import estab

def get_args_parser():
    main_arg_parser = argparse.ArgumentParser(description="parser for video style transfer")
    subparsers = main_arg_parser.add_subparsers(title="subcommands", dest="subcommand")

    train_parser = subparsers.add_parser("train", help="train a model to do style transfer")
    train_parser.add_argument("--cuda", type=int, required=True, help="set it to 1 for running on GPU, 0 for CPU")
    train_parser.add_argument("--width", type=int, default=512, help="width of input image")
    train_parser.add_argument("--height", type=int, default=512, help="height of input image")
    train_parser.add_argument('--batch-size', type=int, default=2, metavar='N', help='input batch size for training (default: 2)')
    train_parser.add_argument('--epochs', type=int, default=100, help='epoch')
    train_parser.add_argument('--lr', type=float, default=0.0001, help='learning rate')
    train_parser.add_argument("--dataset", type=str, required=True, help="path to image dataset")
    train_parser.add_argument("--style_name", type=str, required=True, help="path to a style image to train with")
    train_parser.add_argument('--mean', type=float, default=[0.485, 0.456, 0.406], help='Mean for Normalization for VGG network')
    train_parser.add_argument('--std', type=float, default=[0.229, 0.224, 0.225], help='Standard Deviation for Normalization for VGG network')
    train_parser.add_argument('--alpha', type=float, default=1e0, help='Content Loss weight')
    train_parser.add_argument('--beta', type=float, default=1e5, help='Style Loss weight')
    train_parser.add_argument('--gamma', type=float, default=1e-6, help='Total Variation weight')
    train_parser.add_argument('--lambda_st', type=float, default=1e2, help='Temporal loss weight')
    train_parser.add_argument('--lambda_lt', type=float, default=1e1, help='Long Temporal loss weight')
    train_parser.add_argument('--schedular', type=bool, default=True, help='schedular')
    train_parser.add_argument('--save_model', type=str, default='trained_models', help='trained model is saved here')


    style_parser = subparsers.add_parser("transfer", help="style transfer with a trained model")
    style_parser.add_argument("--cuda", type=int, required=True, help="set it to 1 for running on GPU, 0 for CPU")
    style_parser.add_argument('--save_model', type=str, default='trained_models', help='trained model is saved here')
    style_parser.add_argument('--model-name', type=str, default='', help='model name')
    style_parser.add_argument('--test_img_path', type=str, default='', help='Path for test images')
    style_parser.add_argument('--output_video', type=str, default='output\output.avi', metavar='N', help='Output video name')
    style_parser.add_argument('--fps', type=int, default=15, metavar='N', help='input batch size for training')

    evaluate_parser = subparsers.add_parser("evaluate", help="Calculate temporal error")
    evaluate_parser.add_argument("--path", type=str, required=True, help="path to output images")
    evaluate_parser.add_argument("--cuda", type=int, required=True, help="set it to 1 for running on GPU, 0 for CPU")

    return main_arg_parser.parse_args()
```

```
def main():
    args = get_args_parser()
    # command
    if (args.subcommand == "train"):
        train(args)
    elif (args.subcommand == "transfer"):
        stylize(args)
    elif (args.subcommand == "evaluate"):
        estab(args)
    else:
        print("invalid_command")

if __name__ == '__main__':
    main()
```

D.2 File: train.py

```
import os
import torch
import torchvision
from tqdm import tqdm
from PIL import Image
from torch.optim import Adamax
from dataloader import data_load
import torch.optim.lr_scheduler as ls
from loss_network import Vgg16, Normal
from torch.utils.data import DataLoader
from style_network import ImageTransformer
from utils import gram_matrix, warped, get_mask
style_weight = [1, 1e0, 1e0, 1e0]

def train(args):
    #Training the model
    #code is using GPU if cuda is available else CPU
    device = torch.device("cuda" if args.cuda else "cpu")

    #Building Model
    style_model = ImageTransformer().to(device)
    loss_model = Vgg16().to(device)
    for param in loss_model.parameters():
        param.requires_grad = False

    loss_mse = torch.nn.MSELoss()
    loss_msesum = torch.nn.MSELoss(reduction='none')

    optimizer = Adamax(style_model.parameters(), lr=args.lr)
    schedular = ls.MultiStepLR(optimizer, milestones=[8, 20],
        gamma=0.2)

    #Loading Data
    train_dataset = data_load(os.path.join(args.dataset,
        'training'))
    data_train = DataLoader(dataset=train_dataset,
        batch_size=args.batch_size, shuffle=True)

    mean =torch.tensor([0.485, 0.456, 0.406]).to(device)
    std = torch.tensor([0.229, 0.224, 0.225]).to(device)

    #Loading Style
    style_img = Image.open(args.style_name)
    style_img = style_img.resize((args.width, args.height),
        Image.BILINEAR)
    style_img = torchvision.transforms.ToTensor()(style_img)
    style_batch = []
    style_batch.append(style_img)
    normalization = Normal(mean, std)
    count = 0
    while count < args.epochs:
        data = tqdm(data_train)
        count += 1

        for idx, x in enumerate(data):
            optimizer.zero_grad()

            img1, img2, mask, flow = x
            img1, img2 = img2, img1
            img1 = img1.to(device)
            img2 = img2.to(device)
            mask = mask.to(device)
            flow = flow.to(device)
            style_img = style_batch[0].to(device)

            #style network
            output_img1 = style_model(img1)
            output_img2 = style_model(img2)

            #temporal loss
            warp_img1, mask_boundary_img1 = warped(img1, flow,
                device)
            mask_occ = get_mask(warp_img1, img2, mask)
            warp_output_img1, _ = warped(output_img1, flow, device)
            temporal_loss = loss_msesum(output_img2,
                warp_output_img1)
            temporal_loss = torch.sum(temporal_loss * mask_occ *
                mask_boundary_img1) / (
                    img2.size(0) * img2.size(1) * img2.size(2) *
                        img2.size(3))
            temporal_loss *= args.lambda_lt

            #long temporal loss
            if (idx) % 5 == 0:
                frame0 = output_img2

            with torch.no_grad():
                frame0_mask = get_mask(warp_img1, frame0, mask)
                long_temporal_loss = torch.abs(loss_msesum(frame0,
                    warp_output_img1))
                long_temporal_loss = torch.sum(long_temporal_loss
                    * frame0_mask * mask_boundary_img1) / (
```

```
            frame0.size(0) * frame0.size(1) *
                frame0.size(2) * frame0.size(3))
long_temporal_loss *= args.lambda_st

#normalization to vgg16
img1 = normalization(img1)
img2 = normalization(img2)
style_img =
    normalization(style_img.repeat(output_img1.size(0),
    1, 1, 1))
output_img1 = normalization(output_img1)
output_img2 = normalization(output_img2)

#loss network
content1 = loss_model(img1)
content2 = loss_model(img2)
style_out = loss_model(style_img)
out1 = loss_model(output_img1)
out2 = loss_model(output_img2)

#content loss
loss_content = loss_mse(content1[2], out1[2]) +
    loss_mse(content2[2], out2[2])
loss_content *= args.alpha

#style loss
loss_style = 0.0
for i in range(len(style_out)):
    style_gram = gram_matrix(style_out[i])
    out1_gram = gram_matrix(out1[i])
    out2_gram = gram_matrix(out2[i])
    loss_style += style_weight[i] *
        (loss_mse(style_gram, out1_gram) +
        loss_mse(style_gram, out2_gram))
loss_style *= args.beta

        #total variation loss
        loss_tv_img1 = torch.sum(torch.abs(output_img1[:, :,
            :, :-1] - output_img1[:, :, :, 1:])) \
                + torch.sum(torch.abs(output_img1[:, :, :-1,
                    :] - output_img1[:, :, 1:, :]))
        loss_tv_img2 = torch.sum(torch.abs(output_img2[:, :,
            :, :-1] - output_img2[:, :, :, 1:]))\
                    + torch.sum(torch.abs(output_img2[:, :,
                        :-1, :] - output_img2[:, :, 1:, :]))
        loss_tv = (loss_tv_img1 + loss_tv_img2) /
            output_img1.size(0)
        loss_tv *= args.gamma

        data.set_description('Epoch:%d temp_loss:%.7f
            long_temp_loss:%.7f content_loss:%.2f '
                            'style_loss:%.7f tv_loss:%.1f'%
                            (count, temporal_loss.item(),
                             long_temporal_loss.item(),
                             loss_content.item(),
                             loss_style.item(),
                             loss_tv.item()))

        loss = loss_content + loss_style + loss_tv +
            temporal_loss + long_temporal_loss

        #backpropogation
        loss.backward()
        optimizer.step()

        if (args.schedular):
            schedular.step()
if (not os.path.exists(args.save_model)):
    os.mkdir(args.save_model)
model_path = os.path.join(args.save_model, 'train.pt')
torch.save(style_model.state_dict(), model_path)
```

D.3 File: dataloader.py

```
import os
import torch
import torchvision
import numpy as np
from PIL import Image
from skimage import transform
from torch.utils.data import Dataset
from torchvision.transforms import ToTensor, ToPILImage, Resize

def readFlowfile(name):
    f = open(name, 'rb')
    header = f.read(4)
    if header.decode("utf-8") != 'PIEH':
        raise Exception('Flow file header does not contain PIEH')
    width = np.fromfile(f, np.int32, 1).squeeze()
    height = np.fromfile(f, np.int32, 1).squeeze()
    flow = np.fromfile(f, np.float32, width * height *
        2).reshape((height, width, 2))
    return flow.astype(np.float32)


class data_load(Dataset):

    def __init__(self, path):
        #looking at the "clean" subfolder for images.
        #root_path is training folder inside the MPI Sintel
            dataset folder
        self.width = 512
        self.height = 512
        self.root_path = path
        self.folder_list = os.listdir(self.root_path + "/clean/")
        self.folder_list.sort()
        self.folder_list = [item for item in self.folder_list if
            item.find('bandage_1') < 0]
        self.imglist = []
        for folder in self.folder_list:
            self.imglist.append(len(os.listdir(self.root_path +
                "/clean/" + folder + "/")))

    def __len__(self):
        return sum(self.imglist) - len(self.imglist)

    def __getitem__(self, idx):

        for i in range(0, len(self.imglist)):
            folder = self.folder_list[i]
            imgpath = self.root_path + "/clean/" + folder + "/"
            occpath = self.root_path + "/occlusions/" + folder +
                "/"
            flowpath = self.root_path + "/flow/" + folder + "/"
            if (idx < (self.imglist[i] - 1)):
                n1 = self.converttoString(idx + 1)
                n2 = self.converttoString(idx + 2)
                img1 = Image.open(imgpath + "frame_" + n1 +
                    ".png").resize((self.width, self.height),
                    Image.BILINEAR)
                img2 = Image.open(imgpath + "frame_" + n2 +
                    ".png").resize((self.width, self.height),
                    Image.BILINEAR)
                mask = Image.open(occpath + "frame_" + n1 +
                    ".png").resize((self.width, self.height),
                    Image.BILINEAR)
                flow = readFlowfile(flowpath + "frame_" + n1 +
                    ".flo")
                img1 = ToTensor()(img1).float()
                img2 = ToTensor()(img2).float()
                mask = ToTensor()(mask).float()
                h, w, c = flow.shape
                flow = torch.from_numpy(transform.resize(flow,
                    (self.height, self.width))).permute(2, 0,
                    1).float()
                # flow[0] contains flow in x direction and flow[1]
                    contains flow in y direction
                flow[0, :, :] = flow[0, :, :] *
                    float(flow.shape[1] / h)
                flow[1, :, :] = flow[1, :, :] *
                    float(flow.shape[2] / w)

                #take no occluded regions to compute
                mask = 1 - mask
                mask[mask < 0.99] = 0
                mask[mask > 0] = 1
                break
            idx = idx - (self.imglist[i] - 1)
        # img2 should be at t in img1 is at t-1
        return (img1, img2, mask, flow)

    def converttoString(self, n):
        string = str(n)
        while (len(string) < 4):
            string = "0" + string
        return string
```

D.4 File: stylenetwork.py

```
import torch.nn as nn
from math import floor

class ConvLayer(nn.Module):
    def __init__(self, input_channel, output_channel, kernel_size,
        stride, bias=True):
        super(ConvLayer, self).__init__()
        padding = int(floor(kernel_size / 2))
        self.pad = nn.ReflectionPad2d(padding)
        self.conv_layer = nn.Conv2d(input_channel, output_channel,
            kernel_size, stride, bias=bias)

    def forward(self, x):
        x = self.pad(x)
        x = self.conv_layer(x)
        return x


class ConvInstRelu(ConvLayer):
    def __init__(self, input_channel, output_channel, kernel_size,
        stride):
        super(ConvInstRelu, self).__init__(input_channel,
            output_channel, kernel_size, stride)
        self.inst_norm = nn.InstanceNorm2d(output_channel,
            affine=True)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = super(ConvInstRelu, self).forward(x)
        x = self.inst_norm(x)
        x = self.relu(x)
        return x

class ResidualBlock(nn.Module):
    def __init__(self, input_channel, output_channel,
        kernel_size=3, stride=1, padding=1):
        super(ResidualBlock, self).__init__()
        self.conv_layer1 = nn.Conv2d(input_channel,
            output_channel, kernel_size, stride, padding=padding)
        self.inst_norm1 = nn.InstanceNorm2d(output_channel,
            affine=True)
        self.conv_layer2 = nn.Conv2d(input_channel,
            output_channel, kernel_size, stride, padding=padding)
        self.inst_norm2 = nn.InstanceNorm2d(output_channel,
            affine=True)
        self.relu = nn.ReLU()

    def forward(self, x):
        residual = x
        x = self.conv_layer1(x)
        x = self.inst_norm1(x)
        x = self.relu(x)
        x = self.conv_layer2(x)
        x = self.inst_norm2(x)
        x = residual + x
        return x

class ConvTanh(ConvLayer):
    def __init__(self, input_channel, output_channel, kernel_size,
        stride):
        super(ConvTanh, self).__init__(input_channel,
            output_channel, kernel_size, stride)
        self.tanh = nn.Tanh()

    def forward(self, x):
        x = super(ConvTanh, self).forward(x)
        x = self.tanh(x)
        return x


class ImageTransformer(nn.Module):
    def __init__(self):
        super(ImageTransformer, self).__init__()
        self.enc1 = ConvInstRelu(3, 32, 9, 1)
        self.enc2 = ConvInstRelu(32, 64, 3, 2)
        self.enc3 = ConvInstRelu(64, 128, 3, 2)

        self.res1 = ResidualBlock(128, 128)
        self.res2 = ResidualBlock(128, 128)
        self.res3 = ResidualBlock(128, 128)
        self.res4 = ResidualBlock(128, 128)
        self.res5 = ResidualBlock(128, 128)

        self.upsample1 = nn.Upsample(scale_factor=2,
            mode='bilinear', align_corners=True)
        self.deco1 = ConvInstRelu(128, 64, 3, 1)
        self.upsample2 = nn.Upsample(scale_factor=2,
            mode='bilinear', align_corners=True)
        self.deco2 = ConvInstRelu(64, 32, 3, 1)
        self.deco3 = ConvTanh(32, 3, 9, 1)

    def forward(self, x):
```

```
#encoder
x = self.enc1(x)
x = self.enc2(x)
x = self.enc3(x)
#residual
x = self.res1(x)
x = self.res2(x)
x = self.res3(x)
x = self.res4(x)
x = self.res5(x)
#decoder
x = self.upsample1(x)
x = self.deco1(x)
x = self.upsample2(x)
x = self.deco2(x)
x = self.deco3(x)
return x
```

D.5 File: lossnetwork.py

```
import torch.nn as nn
import torchvision


class Vgg16(nn.Module):
    def __init__(self):
        super(Vgg16, self).__init__()
        vgg = torchvision.models.vgg16(pretrained=True,
            progress=True).features.eval()
        self.s1 = nn.Sequential()
        self.s2 = nn.Sequential()
        self.s3 = nn.Sequential()
        self.s4 = nn.Sequential()

        for i in range(4):
            self.s1.add_module(str(i), vgg[i])
        for i in range(4, 9):
            self.s2.add_module(str(i), vgg[i])
        for i in range(9, 16):
            self.s3.add_module(str(i), vgg[i])
        for i in range(16, 23):
            self.s4.add_module(str(i), vgg[i])

    def forward(self, x):

        x = self.s1(x)
        relu1_2 = x
        x = self.s2(x)
        relu2_2 = x
        x = self.s3(x)
        relu3_3 = x
        x = self.s4(x)
        relu4_3 = x

        return relu1_2, relu2_2, relu3_3, relu4_3


class Normal(nn.Module):
    def __init__(self, mean, std):
        super(Normal, self).__init__()
        self.mean = mean.view(-1, 1, 1)
        self.std = std.view(-1, 1, 1)

    def forward(self, img):
        return (img - self.mean) / self.std
```

D.6 File: utils.py

```
import torch

def gram_matrix(image):
    b, c, h, w = image.size()
    lst = []
    for i in range(b):
        x = image[i]
        gram = x.view(c, h * w)
        gram = torch.mm(gram, gram.T)
        lst.append(gram.unsqueeze(0))
    return torch.cat(lst, dim=0) / (c * h * w)

def get_mask(warp_img, sample, mask):

    #relative luminance -  digtal video
    img_gray = 0.2989 * warp_img[:, 2, :, :] + 0.5870 *
        warp_img[:, 1, :, :] + 0.1140 * warp_img[:, 0, :, :]
    sample_gray = 0.2989 * sample[:, 2, :, :] + 0.5870 * sample[:,
        1, :, :] + 0.1140 * sample[:, 0, :, :]

    #relative luminance - real world video
    #img_gray = 0.2126 * warp_img[:, 2, :, :] + 0.7152 *
        warp_img[:, 1, :, :] + 0.0722 * warp_img[:, 0, :, :]
    #sample_gray = 0.2126 * sample[:, 2, :, :] + 0.7152 *
        sample[:, 1, :, :] + 0.0722 * sample[:, 0, :, :]

    img_gray = img_gray.unsqueeze(1)
    sample_gray = sample_gray.unsqueeze(1)
    mask_cont = torch.abs(img_gray - sample_gray)
    mask_cont[mask_cont < 0.05] = 0
    mask_cont[mask_cont > 0] = 1
    mask_cont = mask - mask_cont
    mask_cont[mask_cont < 0] = 0
    mask_cont[mask_cont > 0] = 1
    return mask_cont


def warped(img, flow, device):
    b, c, h, w = img.size()
    x = torch.arange(0, w)
    y = torch.arange(0, h)
    y_grid, x_grid = torch.meshgrid(y, x)
    grid = torch.cat((x_grid.unsqueeze(0),
        y_grid.unsqueeze(0))).repeat(b, 1, 1, 1).float().to(device)
    grid = grid + flow
    grid[:, 0, :, :] = 2.0 * grid[:, 0, :, :] / (w - 1) - 1.0
    grid[:, 1, :, :] = 2.0 * grid[:, 1, :, :] / (h - 1) - 1.0
    grid = grid.permute(0, 2, 3, 1)

    output = torch.nn.functional.grid_sample(img, grid)
    mask_boundary =
        torch.nn.functional.grid_sample(torch.ones(img.size(),
        device=device), grid, mode='bilinear')
    mask_boundary[mask_boundary < 0.9999] = 0
    mask_boundary[mask_boundary > 0] = 1
    output = output * mask_boundary
    return output, mask_boundary
```