UNIVERSITY OF PATRAS

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Department of Computer Engineering & Informatics - Division of Hardware and Computer Architecture

Deep Learning Video Stabilization

DIPLOMA THESIS

Konstantinos Sardelis

SUPERVISOR: Evagelos Dermatas

PATRAS – February 2024

University of Patras, Department of Electrical and Computer Engineering.

Konstantinos Sardelis

CERTIFICATION

It is certified that the Diploma Thesis titled

Deep Learning Video Stabilization

of the Department of Electrical and Computer Engineering student

Konstantinos Sardelis

Registration Number: 1046971

was presented publicly at the Department of Electrical and Computer Engineering at

……../……../……

and was examined by the following examining committee:

Evangelos Dermatas, Professor, Department of Computer Engineering & Informatics (supervisor)

Kostas Berberidis, Professor, Department of Computer Engineering & Informatics (committee member)

Emmanouil Psarakis, Professor, Department of Computer Engineering & Informatics (committee member)

The Supervisor	The Director of the Division
Evangelos Dermatas Professor	Name Surname Title

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ

Τμήμα Μηχανικών Η/Υ & Πληροφορικής - Τομέας Υλικού και Αρχιτεκτονικής των Υπολογιστών

Deep Learning Σταθεροποίηση Βίντεο

Δ Ι Π Λ Ω Μ Α Τ Ι Κ Η Ε Ρ Γ Α Σ Ι Α

Σαρδέλης Κωνσταντίνος

ΕΠΙΒΛΕΠΩΝ: Ευάγγελος Δερματάς

ΠΑΤΡΑ – Φεβρουάριος 2024

Πανεπιστήμιο Πατρών, Τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών.

Κωνσταντίνος Σαρδέλης

ΠΙΣΤΟΠΟΙΗΣΗ

Πιστοποιείται ότι η Διπλωματική Εργασία με τίτλο

Deep Learning Σταθεροποίηση Βίντεο

του φοιτητή του Τμήματος Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών

ΚΩΝΣΤΑΝΤΙΝΟΣ ΣΑΡΔΕΛΗΣ ΤΟΥ **ΠΑΝΑΓΙΩΤΗ**

Αριθμός Μητρώου: 1046971

Παρουσιάστηκε δημόσια στο Τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών στις

……../……../……

και εξετάστηκε από την ακόλουθη εξεταστική επιτροπή:

Ευάγγελος Δερματάς, Καθηγητής, Τμήμα Μηχανικών Η/Υ & Πληροφορικής (επιβλέπων)

Κωνσταντίνος Μπερμπερίδης, Καθηγητής, Τμήμα Μηχανικών Η/Υ & Πληροφορικής (μέλος επιτροπής)

Εμμανουήλ Ψαράκης, Καθηγητής, Τμήμα Μηχανικών Η/Υ & Πληροφορικής (μέλος επιτροπής)

Ο Επιβλέπων	Ο/Η Διευθυντής/τρια του Τομέα
Ευάγγελος Δερματάς Καθηγητής	Ονοματεπώνυμο Βαθμίδα

Thesis Abstract

With the rapid advancement of technology in portable devices (such as mobile phones, smartwatches, etc.), it is possible to capture high-quality videos without the use of heavy equipment like tripods and stabilizers. However, these videos often contain unwanted camera movements and vibrations. For this reason, digital video stabilization is a rapidly evolving and intensively studied field. The purpose of this work is to contribute to video smoothing through deep learning.

Initially, the reader is provided with an introduction to digital video stabilization through optimization and low-pass filtering, while simultaneously introducing basic concepts of computer vision. Then, a brief overview of deep learning and how convolutional networks learn and operate is presented. This is followed by detailed studies and implementations of video stabilization algorithms using artificial intelligence. These methods can be categorized into 1) camera path reconstruction/optimization without unwanted noise, 2) creation of stable frames through supervised learning, and 3) generation of new intermediate frames in the video. Through these methods, modern deep learning technologies such as transfer learning, perceptual cost functions, and adversarial learning are introduced. All algorithms are compared using a set of metrics, and their effectiveness is discussed. Finally, the issue of lack of training data is addressed by creating synthetic video data. I compare existing methods with two of my own, which use artificial intelligence and statistical analysis to create realistically shaky videos from existing stable videos.

Thesis Abstract (Greek Version)

Με τη ραγδαία εξέλιξη της τεχνολογίας στις φορητές συσκευές (κινητά, έξυπνα ρολόγια κλπ.), είναι δυνατή η λήψη βίντεο υψηλής ποιότητας χωρίς τη χρήση βαρύ εξοπλισμού όπως τρίποδα και σταθεροποιητές. Τα βίντεο αυτά ωστόσο, συχνά εμπεριέχουν ανεπιθύμητη κίνηση της κάμερας και ταλαντώσεις. Για το λόγο αυτό, η ψηφιακή σταθεροποίηση βίντεο είναι ένας τομές που μελετάται έντονα και εξελίσσεται ραγδαία. Σκοπός της εργασίας αυτής είναι η συμβολή της βαθιάς μάθησης στην εξομάλυνση βίντεο.

Αρχικά παρέχεται στο αναγνώστη μια εισαγωγή στην ψηφιακή σταθεροποίηση βίντεο μέσω βελτιστοποίησης και χαμηλοπερατών φίλτρων, εισάγοντας ταυτόχρονα βασικές έννοιες της υπολογιστικής όρασης. Στη συνέχεια παρέχω μια σύντομη επισκόπηση στην βαθιά μάθηση και στον τρόπο που μαθαίνουν και λειτουργούν τα συνελικτικά δίκτυα. Έπειτα ακολουθούν αναλυτικές μελέτες και υλοποιήσεις αλγορίθμων σταθεροποίησης βίντεο μέσω τεχνητής νοημοσύνης. Οι μέθοδοι αυτές μπορούν να κατηγοριοποιηθούν σε 1) ανακατασκευή του μονοπατιού της κάμερας

χωρίς ανεπιθύμητο θόρυβο, 2) δημιουργία σταθερών καρέ μέσω επιβλεπόμενης μάθησης και 3) δημιουργία καινούριων ενδιάμεσων καρέ του βίντεο. Στις υλοποιήσεις αυτές παρέχω σχόλια και βελτιώσεις είτε σε ενδιάμεσα στάδια είτε στην αρχιτεκτονική των δικτύων. Μέσω αυτών εισάγω σύγχρονες τεχνολογίες βαθιάς μάθησης όπως μεταφορά μάθησης, αντιληπτικές συναρτήσεις κόστους και μάθηση μέσω αντιπαλότητας. Όλοι οι αλγόριθμοι συγκρίνονται με ένα σύνολο μετρικών και σχολιάζεται η αποτελεσματικότητα τους. Τέλος αντιμετωπίζω την έλλειψη δεδομένων εκπαίδευσης, μέσω της δημιουργίας συνθετικών δεδομένων βίντεο. Συγκρίνω υπάρχουσες μεθόδους με δύο δικές μου που κάνουν χρήση τεχνητής νοημοσύνης και στατικής ανάλυσης για τη δημιουργία ρεαλιστικά κουνημένων βίντεο από υπάρχοντα σταθερά βίντεο.

Table of Contents

Thesis Abstract vii
Thesis Abstract (Greek Version) vii
Introduction 1
Chapter 1: Problem Formulation and Traditional Approaches 5
1.1 Introduction 5
1.1.1 Motion Estimation 5
1.1.2 Motion Compensation 7
1.1.3 Frame Warping 9
1.2 Global Transformation Motion Representation 10
1.3 Sparse Vertex Motion Representation 11
1.4 Dense Flow representation and Pixel Profile Optimization 16
Chapter Summary 20
Chapter 2: Machine & Deep Learning Principles 21
Introduction 21
2.1 Generalization 21
2.2 Introduction to Deep Learning 23
2.3 A brief overview of backpropagation, gradient descent, and the importance of learning rate 26
2.4 Building Blocks of Neural Networks 30
2.4.1 Fully Connected Layers (Linear or Dense Layers) 30
2.4.2 Convolutional Layers 31
2.4.3 Regularization and Pooling Layers 35
2.5 Advanced Convolutional Network Architectures 36
Chapter Summary 39
Chapter 3: Deep Learning Video Stabilization Methods Overview 40
3.1 Unsupervised Trajectory Optimization 40
3.2 Supervised Learning Approaches 41
3.3 Frame interpolation 42
Chapter 4: Datasets and Evaluation Metrics 44
4.1 Datasets 44
4.2 Method Evaluation and Quality Assessment 47
Chapter 5: Deep Trajectory Optimization 49
Introduction 49
5.1 Implementation of Deep Flow [23] 49
5.1.1 Input Preprocessing 50
5.1.2 Network: Architecture, Input & Loss functions 55
5.1.3 Implementation Details 57
5.1.4 Discussion and Modifications 59

5.1.5 Results and Discussion 62
5.2 Video Stabilization through Learning Motion Refinement and smoothing parameters. 63
5.2.1 Motion Initialization 63
5.2.2 Motion Refinement 65
5.2.3 Trajectory Smoothing 66
5.2.4 Training and Implementation Details 68
5.2.5 PCA-Flow based Image Warping 68
5.2.6 Results and Discussion 69
5.3 Optimization in CNN weight space 70
Chapter Summary 72
Chapter 6: Supervised Video Stabilization 73
Introduction 73
6.1 Deep Video Stabilization with Transformation Learning 73
6.1.1 Network Input 74
6.1.2 Network Architecture 74
6.1.3 Training and Loss functions 75
6.1.4 Implementation Details 76
6.1.5 Modifications 77
6.1.6 Results and Discussion 78
6.2 Original (Vanilla) GANs 80
6.2.1 Improved Adversarial Training 82
6.3 Purely Generative Video Stabilization Approach 86
6.3.1 Dataset Generation Pipeline 86
6.3.2 Network Architecture and Input 87
6.3.3 Training Strategy 87
6.3.4 Modifications 96
6.3.5 Results and Discussion 97
Chapter Summary 98
Chapter 7: Video Stabilization through Frame Interpolation 99
Introduction 99
7.1 DIFRINT Implementation 100
7.1.1 Training Scheme 100
7.1.2 Loss Functions 101
7.1.3 Testing Scheme 101
7.1.4 Implementation Details 102
7.1.5 Discussion and Modifications 105
7.1.6 Evaluation Results 107
7.2 Frame Interpolation With CAIN 108

7.2.1 Network Architecture and Loss function 108
7.2.2 Inference Scheme 109
7.2.3 Evaluation Results 111
Chapter Summary 113
Chapter 8: Synthetic Dataset Generation 114
Introduction 114
8.1 Random Affine Transformations 115
8.2 Sampling Transformations from Noise Dataset 116
8.3 Unstable Video Synthesis with Supervised Transformation Learning 117
8.4 Transformation Learning with Adversarial Training 118
8.5 Unwanted motion generation through PCA 121
8.6 Method Evaluation 123
Chapter Summary 124
Chapter 9: Conclusions, Challenges & Future Directions 125
9.1 Conclusions 125
9.2 Challenges 126
9.3 Future Directions 127
References 129

Introduction

The prevalent integration of high-quality cameras in hand-held devices, such as smartphones and action cameras, has transformed the way the general population records the memorable moments of their lives. These devices have made it incredibly accessible for individuals to capture high-resolution videos in various settings and conditions. However, despite the widespread availability of these devices, one critical challenge remains - the stability of the recorded videos.

While technology has enabled anyone to become a videographer, ensuring the professional-quality stability of videos still typically requires specialized and often expensive equipment such as SteadyCam, Gyrostick, gimbal (for drones), or specially designed lenses and matrices similar to those available in professional cameras. The result is that a significant portion of user-generated content remains plagued by issues of shakiness, jitter, and motion-induced artifacts. These issues not only detract from the overall viewing experience but also limit the usability of these videos in professional contexts.

Consequently, a substantial body of literature and research efforts has been dedicated to solving the video stabilization problem. Video stabilization aims to rectify the instability and unwanted motion present in videos, making them smoother, more professional-looking, and visually appealing. This problem is of paramount importance in various domains, including but not limited to:

1. Entertainment and Media Production: In the entertainment industry, video stabilization is crucial for creating high-quality movies, television shows, and online content. Stable videos enhance storytelling and viewer engagement.
2. Surveillance and Security: Video surveillance systems rely on stable footage to accurately monitor and analyze events. Stabilization ensures clear and actionable footage in security applications.
3. Medical Imaging: In medical imaging, stable video footage is vital for accurate diagnosis and surgical procedures where even minor camera movements can affect the outcome.
4. Virtual Reality (VR) and Augmented Reality (AR): Stable video is essential for creating immersive VR and AR experiences, reducing motion sickness, and enhancing user engagement.

5. Consumer Electronics: Improving video stabilization algorithms directly benefits consumer devices like action cameras, drones, and smartphones, enhancing the overall user experience.

In light of these considerations, the need for effective video stabilization techniques has never been more pressing. This thesis aims to contribute to this evolving field by leveraging deep learning techniques to develop advanced video stabilization models that can provide stable and visually pleasing results, even in challenging scenarios.

By addressing the video stabilization problem through the lens of deep learning, we seek to bridge the gap between professional-grade stability and the convenience of consumer-grade devices, making stable video recording and production accessible to a wider audience.

All Digital Video Stabilization methods, however, suffer from the following phenomena:

1. Crop and Quality Loss: To stabilize video, frames are often cropped or scaled, which can lead to a reduction in the video's resolution and quality. This loss of content can be undesirable, especially when maintaining the original field of view is crucial.
2. Limited Compensation: While stabilization can mitigate shakiness to a certain extent, it cannot completely eliminate all forms of motion, especially when dealing with severe and erratic camera movements.
3. Artifact Introduction: Some stabilization techniques may introduce artifacts, such as unnatural warping or distortion, if not applied carefully. These artifacts can negatively impact the visual quality of the stabilized video.
4. Computational Complexity: Complex video stabilization algorithms can be computationally intensive, requiring significant processing power and time, making real-time stabilization challenging on some hardware.
5. Handling Complex Scenes: In scenes with fast and complex motion, occlusions, or rapid changes in perspective, traditional stabilization methods may struggle to produce satisfactory results.

Deep Learning

In response to the challenges posed in the field, researchers and engineers are continually pushing the boundaries, exploring innovative techniques that harness the power of deep learning to enhance the efficacy and efficiency of digital video stabilization. The primary objective is to reconcile the convenience offered by modern

video recording devices with the growing demand for professional-quality stability in recorded content.

Deep learning is a subset of machine learning and a key component of artificial intelligence, that distinguishes itself by utilizing deep neural networks with multiple layers to automatically learn hierarchical representations of data. In the realm of computer vision, deep learning has undergone a transformative evolution, empowering machines to comprehend and interpret images and videos. This technological advancement has become a cornerstone in various applications, including image classification, object detection, and image generation. The advance of Convolutional Neural Networks (CNNs) has particularly revolutionized computer vision, offering specialized neural networks designed to efficiently process grid-like data such as images and videos. CNNs leverage convolutional layers to autonomously learn and extract features from input data, beginning with basic features like edges and textures and progressing to more intricate features. This enables precise recognition, classification, segmentation and many more tasks.

Deep Learning Video Stabilization

Leveraging the capabilities of deep learning for the task of video stabilization can be implemented using a variety of techniques. These approaches can be broadly categorized into the following three main groups:

1. **Motion Refinement**: These methods involve utilizing a neural network to compensate for the unwanted camera motion. The networks infer transformations for the unstable frames, with the aim of producing an overall stable and more pleasant video.
2. **Generative Methods**: These methods focus on generating stable intermediate frames from unstable sequences using supervised learning. Complex architectures, including Generative Adversarial Networks (GANs), are often employed, with transfer learning playing a significant role. Carefully crafted loss functions are utilized to ensure stable and visually pleasing results.
3. **Frame Interpolation**: Frame interpolation techniques aim to mitigate unwanted motion by generating intermediate frames, effectively applying a low-pass filter to the video. These methods contribute to smoother and visually appealing videos, bridging the gap between consecutive frames and enhancing overall stability.

Data Availability

In the domain of video stabilization, the availability of labeled data presents a unique challenge. Ideally, to train effective video stabilization models, one would require a comprehensive dataset consisting of pairs of stable and unstable video sequences. These pairs would serve as the foundation for supervised learning, allowing the model to learn the intricate patterns and transformations needed to stabilize video recordings effectively.

However, in practice, such a dataset is often scarce or entirely absent. While some video datasets may include stabilized and unstabilized versions of the same content, there is often a slight perspective mismatch between them. This perspective mismatch can confuse generative models and pose challenges for training.

To address this data scarcity issue and generate authentic training data, innovative approaches have been explored. For instance, Wang (citation) developed a hardware setup using two portable GoPro Hero 4 Black cameras and a hand-held stabilizer, where the cameras were positioned horizontally next to each other with a small disparity. This setup allowed the capture of synchronized pairs of stable and unstable video footage.

Additionally, there have been attempts to create synthetic stable/unstable video pairs. The generation of synthetic data involves simulating camera motion and instability, paired with the corresponding stable version. We will delve deeper into these efforts in the last chapter of this thesis, discussing the challenges and insights gained from synthetic dataset creation.

Summary

Throughout the course of this thesis, we will explain the traditional formulation of the digital video stabilization problem. We will introduce basic deep-learning blocks and principles and then proceed to use them for the task at hand. We will address the task through the techniques mentioned above. Detailed reviews of papers will be offered along with my proposed modifications. Each method will be evaluated with common metrics used throughout the literature. Finally, we will address the data availability issue, by comparing existing synthetic dataset generation techniques along with two new methods I propose. Code implementations for most methods described will be made available on my GitHub profile.

Chapter 1: Problem Formulation and Traditional Approaches

1.1 Introduction

For us to understand how Deep Learning can offer solutions to the video stabilization problem, we must first understand traditional optimization approaches and how the problem is formulated.

Video stabilization at its core is three step process illustrated in Figure 1.0. The first step is motion estimation, which means establishing a motion model and estimating global or local motion vectors. The second step, motion compensation involves finetuning the motion model, in order to remove unwanted camera motion. Lastly, in the final step each frame is warped based on the stable camera path.

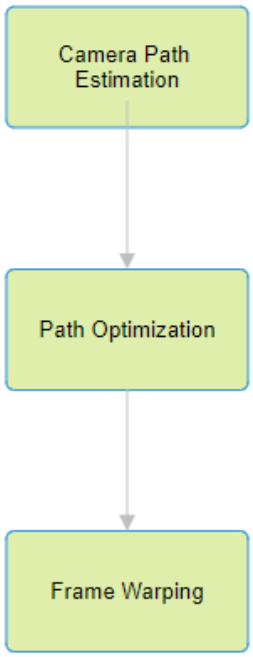


Figure 1.0 Video Stabilization Algorithm Overview

1.1.1 Motion Estimation

This step is crucial to video stabilization, as it involves estimating the camera trajectory of the video over time. A key concept that we have to introduce is optical flow. Optical flow refers to the pattern of apparent motion of objects in a visual scene caused by the relative motion between the observer (camera) and the scene. It is a fundamental technique used to analyze the motion of objects within a sequence of images or video frames. It can be categorized into two main types: sparse optical flow, dense optical flow.

1) **Sparse Optical Flow**: This type of flow computes motion vectors only at specific points in the image, typically at selected key features or interest points. These key points are critical because they represent distinct structures in the scene, such as corners or edges, where significant visual information and motion changes often occur. The gradient of an image at a particular point describes the direction and magnitude of the intensity change. In the context of feature detection, high gradients often indicate the presence of interesting structures like corners or edges. Based on this notion there is a plethora of feature detection algorithms such as: Shi-Tomasi Corner Detector [1], SURF (Speeded-Up Robust Features) [2], SIFT (Scale-Invariant Feature Transform) [3]. The motion is estimated based on matched such features between consecutive frames. In 1981, [4] introduced a widely employed differential method for optical flow. The method assumes that the optical flow in the vicinity of pixels remains constant, and the least-squares method is applied to solve the optical flow equation for all pixels within that neighborhood. It's worth noting that since the original proposal, several improvements have been suggested, and these enhanced methods are now more commonly utilized. For instance, [5] have adopted the pyramid-based iterative Lucas-Kanade method for optical flow. In their work, they not only utilize this method but also introduce enhancements by proposing an estimation of camera motion based on the histogram of local motion. These advancements contribute to a more refined and robust approach to optical flow analysis compared to the original Lucas-Kanade method.

   Using this method, we could infer a global transformation as we will see in 1.2 for the whole frame or divide the image into regions, each with its spatially distinct motion as demonstrated in 1.3.

   However, sparse optical flow estimation algorithms heavily rely on the quality of the detected keypoints. Outlier filtering is a crucial step that must be implemented. Also, large moving objects in the scene or regions with uniform color may cause such algorithms to fail.

   After all inter- frames motions have been calculated, we can sum the cumulatively to get the camera path over the whole video.

2) **Dense Optical Flow**: This type of flow computes motion vectors for every pixel in the image, providing a more detailed representation of the motion across the entire frame. Dense optical flow is beneficial when a detailed understanding of the motion in the entire scene is required, such as in applications like video stabilization, where the smoothness of the entire frame is crucial. Perhaps, the most famous algorithm is the Farnerback Method [6]. The process involves calculating spatial derivatives (gradients) of an image, employing second-order polynomials to approximate local patches and

capture intensity variations and incorporating temporal derivatives to estimate motion between consecutive frames in a video sequence. However, there are other methods like [7] and Deep-Learning frameworks such as [8] which are very accurate and commonly used for better inference speed.

This type of flow also has its drawbacks. First of all, since it involves estimating the motion vectors of each pixel it can be computationally intense. Apart from that, Handling occlusions and regions with abrupt motion changes can be challenging for traditional dense optical flow algorithms.

Again, the final camera trajectory for each pixel is estimated through cumulatively adding all inter frame flows.

After the camera path is estimated, using either approach, we can move on to the next step, which is motion compensation, to refine the camera motion and remove unwanted noise and jerkiness.

1.1.2 Motion Compensation

The motion compensation step in a video stabilization algorithm involves adjusting each frame to counteract the effects of camera motion. In figure 1.1 we illustrate the x-trajectory of an unstable video where the person holding the camera is walking. It is clear that there is high frequency noise introduced by handheld jerkiness and low frequency bounces caused by the person walking.

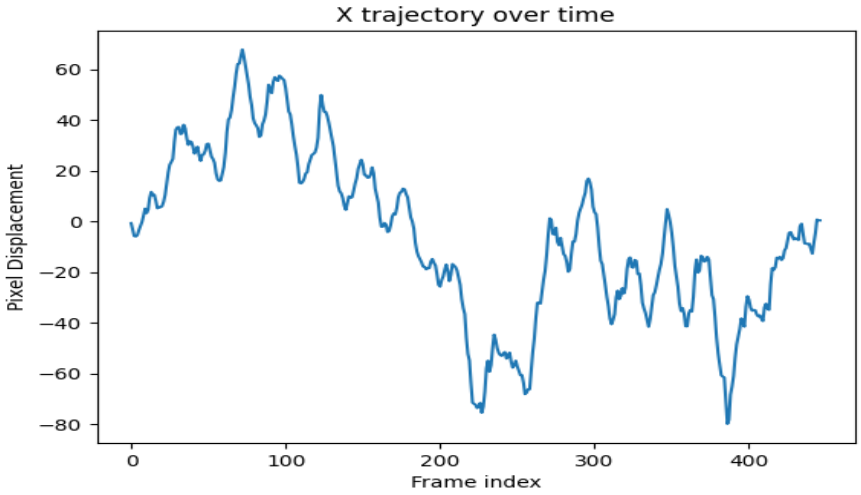
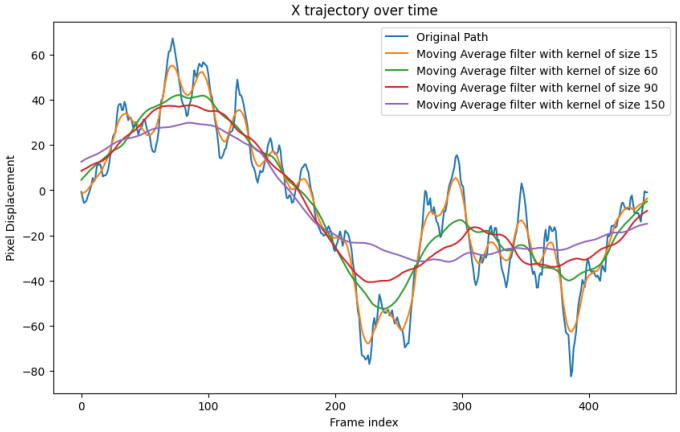


Figure 1.1 Camera Path in the x direction of an unstable video where the person holding the camera was walking.

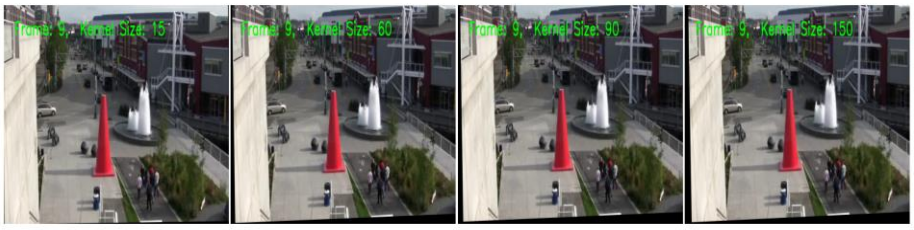
We want to estimate a smoother version of this trajectory to generate a more visually pleasing result. This step is crucial as it directly addresses unwanted jitter or shake in the video, providing a smoother and more visually pleasing result. There are two approaches to compensate motion: Parametric filtering, Trajectory Smoothing.

1) **Parametric filtering:** This approach involves applying a low-pass filter to the extracted camera path, in order to discard unwanted abrupt motion. There is

a plethora of low-pass filters to choose from such as moving average, median, or gaussian. However, the choice of the filter and its intrinsic parameters is nontrivial, because over-smoothing the path can introduce additional distortions and a heavy amount of cropping. This is illustrated in figure 1.2, where a moving average filter with a kernel of size 150 achieves the smoothest camera path at the cost of cropping. The goal is to estimate an optimal camera path that is smooth over time but stay close to the original trajectory.



a)



b)

Figure 1.2 a) Comparison of different kernel sizes for a moving average filter b) Illustration of larger kernels leading to more cropping.

The choice of the filter and its parameters heavily depends on the type of noise that is present in the video and there is no 'one size fits all' approach. For example, we want to mitigate the effects of high frequency jerkiness present in handheld videos alongside the low frequency bounces of a person walking. This would require empirically creating a band pass filter specific for each case.

2) **Trajectory Smoothing:** These approaches formulate video stabilization as a minimization problem, where the objective function is a weighted sum of the distance from the original path and a carefully crafted smoothing term (adaptive in most cases) which aims to not distort the video. In 1.2 we demonstrate how [9] generates a smooth path while adhering to some imposed constraints through linear programming. Another way to smoothen the camera motion is to write the objective function consisting of the similarity

and smoothness term in the form $A * x = b$ and solve it iteratively, using a sparse linear solver. The last method is demonstrated in 1.3, 1.4.

Now that we have the optimal camera path, we can move on to video synthesis by warping the original frames with the necessary transformations.

1.1.3 Frame Warping

After estimating the optimal path, we can obtain the transformations needed to stabilize each frame by:

$$transformations = optimal\ path - original\ path$$

The type of transformation is clearly dependent on the motion model we have chosen. When the camera path is modeled in a global sparse flow representation for each frame, then warping is achieved through a 2D geometric transformation such as an affine or a homography transformation. Using this method can cause blank areas to appear due to the loss of image pixels in the motion compensation when the video is output.

On the other hand, when motion is modeled in a sparse grid representation, we either have to interpolate the values for all pixels or use an image stitching approach known as Mosaic. This is the approach used in 1.3. However, using Mosaic stitching may cause artifacts in non-planar scenes and around moving objects.

Finally, when motion is estimated through dense optical flow, the transformations are also a dense warp field. This is the simplest warping method as it only involves remapping through standard interpolation methods.

The rest of this chapter will briefly cover three different approaches to video stabilization through path optimization. Each one uses different motion estimation techniques and different optimization methods. These concepts are of crucial importance to us, so that we can later leverage Deep Learning capabilities to enhance them.

1.2 Global Transformation Motion Representation

In [9] the aim is to construct an optimal path consisting of constant linear and parabolic segments, through minimizing the first, second and third derivative of the resulting camera path in an attempt to resemble camera motions employed by cinematographers. The goal is to be able to recreate:

1. A static camera i.e. a constant camera path by setting the derivative to zero, $DP(t) = 0$
2. Motions of constant velocity like panning or a dolly shot by setting the second derivative of the path to zero, $D^2P(t) = 0$
3. Smooth transitions between a static and panning camera with constant acceleration by setting the third derivative of the path to zero, $D^3P(t) = 0$

The objective function is formulated as:

$$O(P) = w1 * |D(P)|_1 + w2 * |D^2(P)|_1 + w3 * |D^3(P)|)$$

Where D is the differential operator, P is the camera path and $w1, w2, w3$ are carefully selected weights to balance the minimization of each derivative.

The goal is to minimize this objective while adhering to the following constraints that are imposed through a linear programming framework.

Inclusion constraint: The path transformation of the crop window, denoted as $P(t)$, must always fit within the transformed frame rectangle, $C(t)$, representing the original camera path. When we treat this as a strict rule, it ensures that during video stabilization and retargeting, all pixels within the crop window contain valid information.

Proximity constraint: The new camera path, $P(t)$, should maintain the original cinematic style of the movie. For instance, if the original camera path included segments with zooming, the optimal path should also follow this motion but in a smooth and coherent manner.

Saliency constraint: Salient points, such as those identified by a face detector or through a saliency map, should be present within the crop window after it's transformed by P(t). This constraint is best considered as a flexible guideline to avoid excessive tracking of salient points, which can lead to abrupt and non-smooth motion in non-salient regions.

The motion is modeled as a series of global frame affine transforms F with six degrees of freedom that can express rotation, scaling, shear, and translation. The original camera path can be expressed as: $C_{t+1} = C_t F_{t+1} \Rightarrow C_t = F_1 F_2 \dots F_t$. The aim is to compute B_t for each frame, to obtain the optimal path P_t: $P_t = C_t B_t$, where $B_t = C_t^{-1} P_t$. This way of modeling motion in combination with linear programming, makes

it straightforward to impose constraints on different parts of the transformation, allowing us to have strict bounds on the amount of zoom, rotation etc. This algorithm works well in many cases and was previously deployed by YouTube as its stabilization standard. However, a global frame transformation cannot handle cases with variable scene depth or the presence of dynamic objects which do not match the frame motion.

This issue could be addressed by calculating multiple camera paths spatially distributed over the frame.

1.3 Sparse Vertex Motion Representation

In MeshFlow [10] each frame is divided uniformly into a grid mesh as shown in Figure 1.4. The motion is modeled as the displacements between frames of the vertices that define the grid mesh. This highly flexible model lies between a global linear transformation and dense per-pixel optical flow and is able to handle parallax, while at the same time being much more computationally efficient than dense optical flow. However, its high degree of freedom prohibits linear programming and also calls for strong regularization.

Motion Model

Their algorithm initializes vertex motion through calculating a global homography between matched features. Based on the principle that each grid vertex should have similar motion to its nearby key-point features, the motions of all nearby detected features within a search radius around the vertex are passed through a median filter F1, ordered by the distance to the vertex. Then the average of the filtered motions gets accumulated to the vertices. After all the vertices have been updated, a second median filter F2 gets passed over the whole frame, as a way to ensure further spatial coherence. The median filter is commonly used in optical flow estimation as illustrated in [11] and a similar idea is implemented here.

With a higher number of grid cells the algorithm is more robust but it comes at cost over time efficiency. Since this approach involves detecting key-point features, its robustness also depends heavily on the quality of the detected points and their distribution over the frame. It's also noteworthy that this approach is sensitive to outliers, so the authors reject them using a homography fitting Ransac method at a 4x4 sub – image level, instead of a global level as we need to retain motion that does not lie in the global linear space. As this motion model itself enforces spatial smoothness no additional spatial constraints need to be added to the objective function.

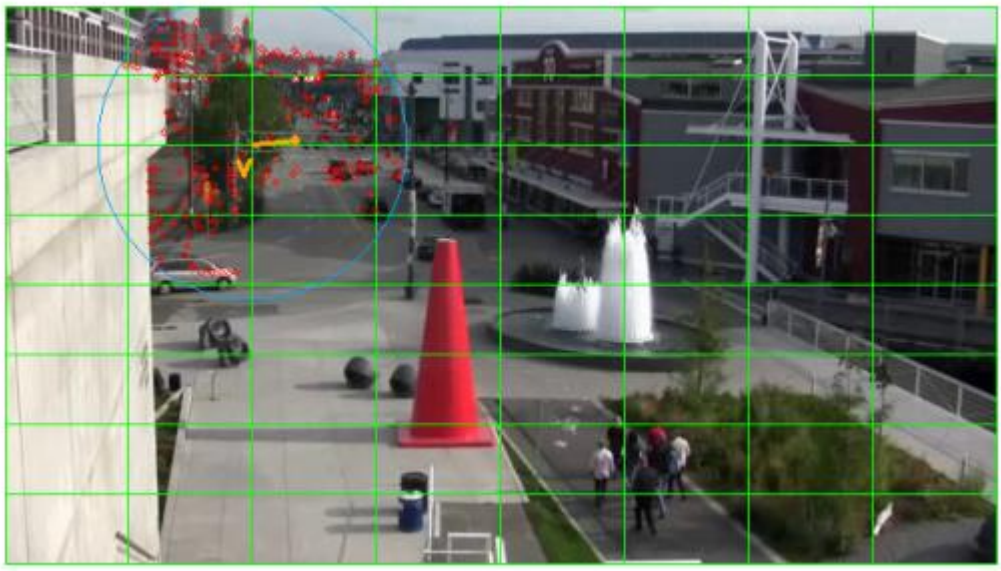


Figure 1.4: a) A uniform mesh grid of 16x16 cells is used in the algorithm but 8x8 is used for visualization b) A single vertex denoted as V is drawn along with its spatial search radius and the keypoints within it which will affect the vertex motion.

Similarly to [10], [12], also utilizes a sparse grid motion representation with different estimation and regularization techniques. They, however, compute the optimal path in a very similar manner.

Path Optimization

Both methods define an objective function that balances two terms, deviation from the original path and a smoothness term. The smoothness term in [12] is a bilateral weight composed of two gaussian functions while in [10] it's a simple gaussian filter, which assigns larger weights to nearby frames.

The optimization function for each vertex path is:

$$O\big(P(t)\big) = \sum_{t} (\|P(t) - C(t)\|^2 + \lambda_t * \text{smoothness term})$$

Specifically, for MeshFlow the function is:

$$O\big(P(t)\big) = \sum_{t} (\ \|P(t) - C(t)\|^2 + \lambda_t\ * \sum_{r \in \Omega_t} w_{t,r} \|P(t) - P(r)\|^2)$$

Here Ω_t denotes the temporal smoothing radius which is set to 20 frames in the algorithm and $w_{t,r}$ is a gaussian weight which is set to $exp\ (-\|r\ -\ t\ \|\text{^}2\ /\ (\Omega\_t\ /\ 3)\text{^}2)$.The most important part of both objective functions is the coefficient of the smoothness term. If frames with large velocity are oversmoothed this can lead to large black borders. Additionally, scenes with large depth deviation also require specially determined amounts of smoothing, so that that they do not introduce distortion.

The way [12] solve this problem is initializing λ to a relatively high value, for example λ = 5 for every frame. The user imposes constraints regarding the amount of cropping and distortion that are acceptable and then λ is iteratively decreased until the constraints are satisfied. Although this method is effective, it clearly is not efficient.

In MeshFlow they propose using indicators for λ that can be computed in real time from a global homography transformation between neighboring frames, during the motion estimation stage. The first indicator T_u, measures the velocity of the frame and is computed as the magnitude of the translational elements of the transformation. The second indicator Fa, is the ratio of the two largest eigenvalues of the affine part of the transformation. The first step is obtaining the optimal value for λ, with the iterative method described above on a dataset of videos containing large motions, parallax and scene depth variation. Then two linear models were trained to map each indicator to the optimal value of λ.

$$\lambda_t' = -1.93 * T_u + 0.95$$

$$\lambda_t'' = 5.83 * Fa + 4.88$$

The final value of λ_t is choses as $\max(\min(\lambda_t', \lambda_t''), 0)$. After the lambda values and gaussian weights for each frame have been computed the objective function can be written in the form $A * x = b$, which is quadratic and can be minimized with a sparse linear solver. Here we solve it iteratively with a Jacobi based solver. What we need is a good initial guess for x. Here we will choose the original camera path. The solution is obtained iteratively by:

$\boldsymbol{x_{i+1}} = (\boldsymbol{diag}[\boldsymbol{A}])^{-1} * \{\boldsymbol{b} - (\boldsymbol{A} - \boldsymbol{diag}[\boldsymbol{A}]) * \boldsymbol{x_i}\}$, where i is the iteration index.

We keep updating x for an empirically determined number of iterations or until the absolute change of the value is smaller than a tolerance we have defined. In the MeshFlow algorithm the process is repeated for 100 iterations.

Shown in Figure 1.6 are the optimized camera paths for a single vertex, using the adaptive λ weights computed during motion estimation and paths computed with constant lambda values. The yellow path, obtained with the adaptive λ weights, is smooth while maintaining the original path's properties. Contrary the red path was

optimized using a constant value of λ = 100 and resembles more of a straight line and is bound to introduce distortion.

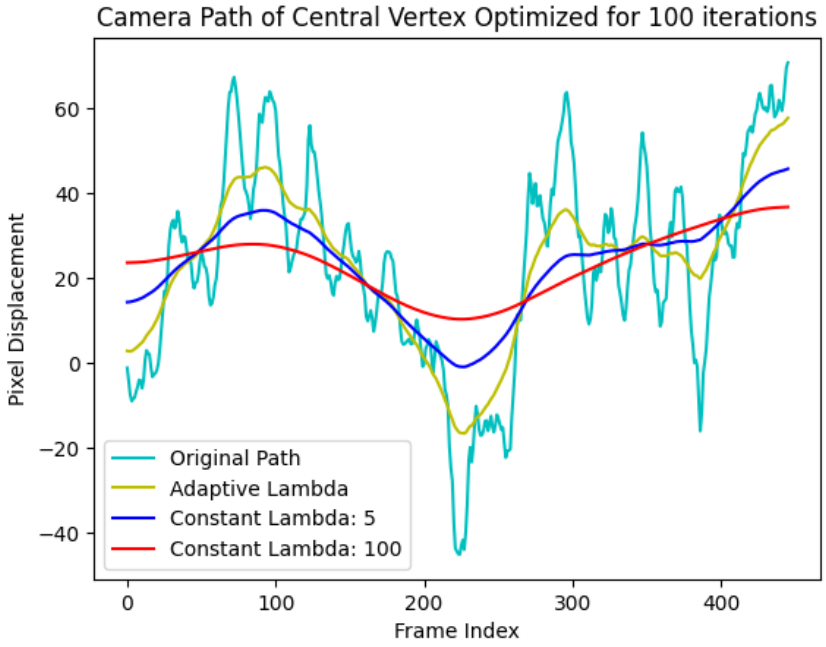


Figure 1.6: Τροχιά κεντρικής κορυφής με εξομαλυμένη με διαφορετικά βάρη λ.

Sparse to Dense Flow

After estimating the optimal path for all grid vertices, we can obtain the transformations needed to stabilize each frame by:

$$transformations = optimal\ path - original\ path$$

These transformations are sparse, of shape $M\ x\ N$ (where M, N is the number of grid vertices in the vertical direction and horizontal direction) and need to be interpolated to the shape $H\ x\ W$ in order to be applied to all pixels.

We will compare the simplest way to achieve that, bilinear interpolation with the multi-homography Mosaic method employed in MeshFlow. Bilinear interpolation fills the values of missing pixels based on the weighted sum of the values of the four vertices of its enclosing grid. More specifically, for a pixel p with image coordinates (x, y) which lies in the cell defined by the vertices $[v_1, v_2, v_3, v_4]$ with coordinates $[(x1, y1), (x2, y1), (x2, y2), (x1, y2)]$, the formula for getting its value is:

$$p_{\text{value}} = \left(1 - \frac{x - x_1}{x_2 - x_1}\right)\left(1 - \frac{y - y_1}{y_2 - y_1}\right)v_1 + \left(1 - \frac{x - x_1}{x_2 - x_1}\right)\frac{y - y_1}{y_2 - y_1}v_2 + \frac{x - x_1}{x_2 - x_1}\left(1 - \frac{y - y_1}{y_2 - y_1}\right)v_3 + \frac{x - x_1}{x_2 - x_1}\frac{y - y_1}{y_2 - y_1}v_4$$

On the other hand, the Multi-Homography approach, initializes a grid for all pixels, where each element's value is its coordinates. All the sparse flow cells are iterated and

a homography transformation H is fitted for the cell's vertices $[v_{1,t}, v_{2,t}, v_{3,t}, v_{4,t}]$ and their positions in the next frame $[v_{1,t+1}, v_{2,t+1,}, v_{3,t+1,}, v_{4,t+1}]$. The values for the missing pixels are obtained by warping their position with H. We are left with a dense warp field, where each block was calculated through independent homography fittings.

Before we proceed to compare the two methods through visualization, a fundamental question arises: How can we effectively visualize optical flow data? The difficulty lies in the fact that raw optical flow data, doesn't conform to a standard intensity range, such as the 0 to 255 range commonly used for traditional images. Instead, optical flow data is represented by channels that denote horizontal and vertical pixel displacements. So, the challenge is to translate this data into pixel intensity in a meaningful manner. Fortunately, we can turn to the HSV (Hue, Saturation, Value) color format as a solution.

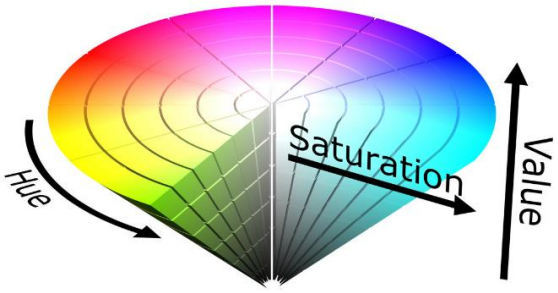


Figure 1.7 HSV Color Format.

A commonly employed technique involves transforming the Cartesian representation of motion, characterized by the horizontal dx and vertical dy components, into its polar equivalent, comprising magnitude and angle. Specifically:

$$\text{magnitude} = \sqrt{dx^2 + dy^2}$$

$$angle = arctan\left(\frac{dy}{dx}\right)$$

Next, we create an empty HSV image and set the saturation channel to its maximum value of 255. We adjust the hue according to the angle of motion and the value based on the normalized magnitude. Converting this image back to the RGB color space yields a meaningful visualization of optical flow. You can observe the results of this method in Figure 1.8.

Figure 1.8 Most pixels are depicted with a light blue color meaning that their motion was to the left with a varying magnitude, which translates to different color intensities. At the same time many pixels are depicted as black. This does not mean that they don't have optical flows, but the estimation algorithm failed on those texture-less regions. The algorithm used to generate the flow in the figure is Gunnar Farnerback's algorithm [6].

The loss of information when interpolating the sparse flow representation contrary to the multi-homography approach, is apparent in figure 1.9. So, when dealing with videos containing variable depth scenes and dynamically moving objects, the second technique is heavily favored.

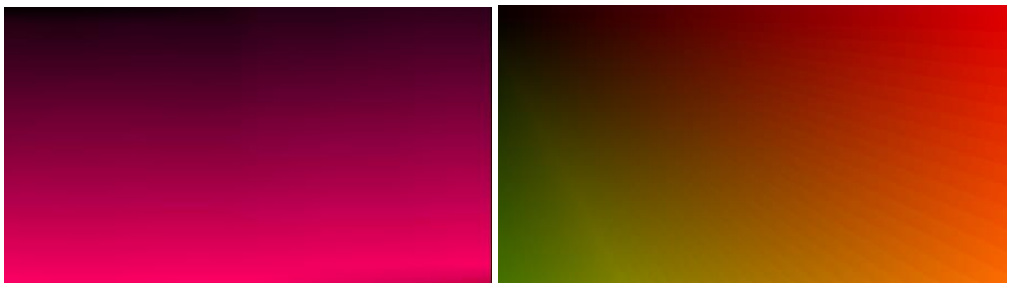


Figure 1.9 On the right side the warp field was interpolated to this dense representation through bilinear interpolation. The left warp field on the other hand was estimated through the multi-homography scheme explained above.

Another powerful technique for obtaining dense warp fields from sparse grids and for inpainting regions of inaccurate flow is offered through Principal Component Analysis as explained in PCA FLOW [13]. An in-depth analysis of this approach is offered in Chapter 3.2.1.

1.4 Dense Flow representation and Pixel Profile Optimization

SteadyFlow [14] suggests that a video could be stabilized by smoothing pixel camera paths, instead of feature or grid vertex trajectories as we have seen so far. They call these pixel trajectories pixel profiles and their difference to feature trajectories is depicted visually in Figure 1.10.

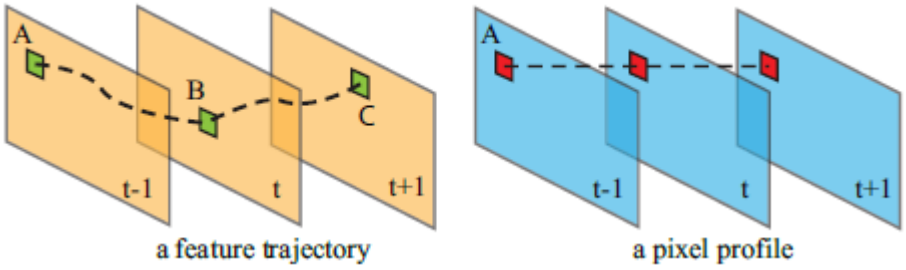


Figure 1.10. [14] Feature trajectory vs pixel profile. A feature trajectory tracks a scene point while a pixel profile collects motion vectors at the same pixel location across the whole video.

The motion is modeled in a dense field manner by accumulating the displacement vectors at each pixel location. We can stabilize a video with gradual depth changes by directly smoothing the raw pixel profiles. However, for videos with discontinuous motion regions, optimizing the raw optical flow will cause severe image distortions. This calls for regularization, that is achieved by inpainting such regions in a similar manner to the one described in [15]. They call the inpainted spatially smooth flow, SteadyFlow. More details on how the inaccurate motion regions are identified and how the motion completion is achieved will be provided after we discuss the advantages of pixel profiles over feature trajectories.

Advantages over Feature Trajectories

Pixel profiles excel in comparison to feature trajectories for several reasons. Pixel profiles are densely distributed both in space and time, whereas feature trajectories are sparse and can extend beyond the video frame, making it difficult to apply effective smoothing techniques. Furthermore, achieving accurate, long feature trajectories is challenging and can result in drifting errors. Smoothing feature trajectories independently can introduce distortions, necessitating additional constraints. In contrast, pixel profiles can be individually smoothed as long as the flow field is spatially smooth.

The quality of optical flows plays a vital role in pixel profiles, but optical flow estimation may lack precision in texture-less regions and object boundaries. However, these inaccuracies tend to have a smaller impact on regions without significant structure. To address issues at object boundaries, techniques like discontinuity abolition and motion completion are employed for mitigation.

Algorithm

Firstly, the raw optical flow must be initialized. There is a plethora of algorithms for robust dense flow estimation, however the authors first align all frames using matched KLT features [4], [1] and they follow the algorithm described in [16]. Now to demonstrate the need for motion inpainting we will compare two pixel profiles, one residing in static background and the other on a dynamically moving object as shown

in Figure 1.11. The pixel marked by the red circle always lies on static background, therefore its trajectory is smooth over time. On the other hand, the trajectory of the pixel marked by a green rectangle has high frequency components, since a lot of moving objects (cars) pass through it along the video.

If we attempt to perform a uniform smoothness optimization on both kinds of pixel profiles, we will introduce heavy amounts of distortion on moving objects and on depth changes. For this reason, regions containing such pixels need to be masked and inpainted.

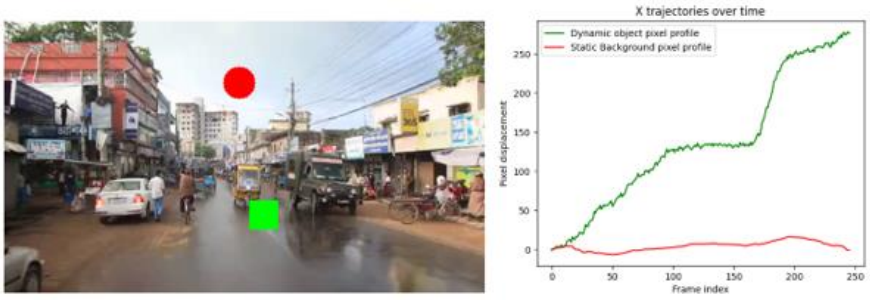


Figure 1.11 Pixel profiles which lie in different regions of the video. The pixel denoted by a red circle always lies on static background, while the pixel denoted by green square lies in the center of the frame and its accumulated motion vectors have high frequency components.

As for the motion completion task, the following scheme is followed. Firstly, the outlier mask must be created. A pixel p is identified as an outlier if its trajectory up to t differs more than a threshold ϵ, from the gaussian smoothed version of the trajectory. The gaussian filter has standard deviation $\sigma = 3$. The formula for outlier detection is :

$$M_t(p) = \begin{cases} 0, & (\|c_t(p) - G \otimes c_t(p)\|) > \epsilon \\ 1, & otherwise \end{cases}$$

Where $c_t(p)$ is the pixel profile up to t , G is the gaussian filter and ϵ, is a threshold that is decreased with each iteration the algorithm is run. The missing regions are filled based on the concept of 'as-similar-as-possible' introduced in [17]. In essence the mask boundaries are used as control points and the motion is filled by warping 2D meshes of size 40x40 pixels. Mathematically it means minimizing the energy function:

$$E(V) = E_d(V) + E_s(V)$$

Where Ed is the data term and Es is the smoothness term. Specific details of their definition and importance can be found in [17]. Very broadly, the data term encourages the bilinear interpolation of the optical flow of the control points to stay close to its original value. The smoothness term penalizes the deviation of each output grid cell from a similarity transformation of its corresponding input grid cell.

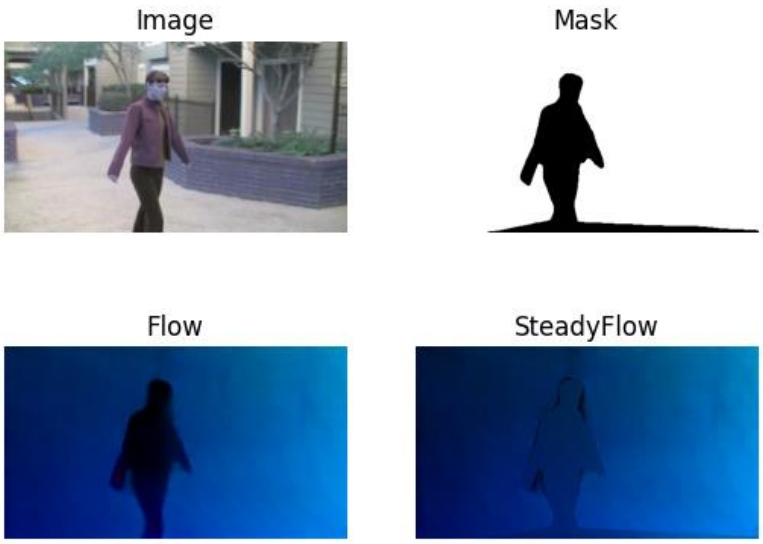


Figure 1.12 a) A frame t b) Its motion completion mask, generated for the first iteration of the algorithm. c) The original flow d) Its inpainted SteadyFlow counterpart.

By optimizing the combined energy function for every frame, we can obtain SteadyFlow as depicted in Figure 1.12. Then through the cumulative sum of SteadyFlow, we get the pixel profiles. Once this process is completed, we can move onto the optimization stage. The objective function to be minimized is the same as the one used in MeshFlow [10]:

$$O\big(P(t)\big) = \sum_{t} \big(\|P(t) - C(t)\|^2 + \lambda_t * \sum_{r \in \Omega_t} w_{t,r} \|P(t) - P(r)\|^2 \big)$$

However, there is one important difference. Since we aim to replicate feature trajectory smoothing through pixel profiles, the technique requires a feature trajectory to be similar to its corresponding pixel profile within the temporal window Ω_t. For this reason, an adaptive temporal smoothing window is utilized as opposed to MeshFlow, where the temporal smoothing radius was fixed to 20 frames. The requirement is that a feature trajectory stays inside a spatial window centered around its assigned pixel profile A. Once the trajectory drifts outside the window it would introduce errors to the approximation. So Ω_t is estimated for each pixel to ensure that a feature point that started at pixel A, stays within a valid range around it. The concept of the adaptive temporal window is illustrated in Figure 1.13.

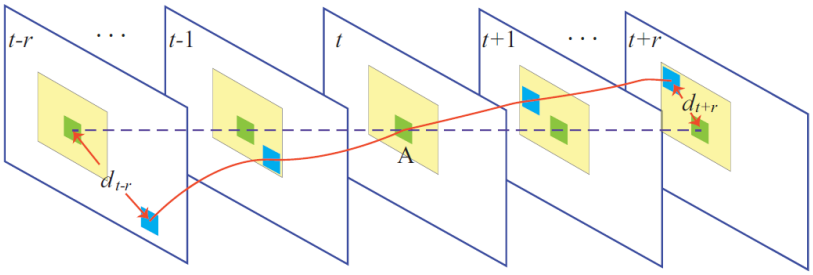


Figure 1.13: [14] The adaptive temporal window is selected so that a pixel profile stays close to its neighboring feature trajectories.

Then the objective function can be solved using a sparse linear solver like the iterative Jacobi method we illustrated in 1.3.2. The whole method is run for 5 iterations, with ϵ decreasing per each iteration according to the formula:

$$\epsilon = \left(1 + \alpha^{(1/\xi)}\right) * \beta$$

Where ξ is the iteration index and $\alpha = 20, \beta = 0.2$ are empirically determined values. Essentially, this means masking out bigger parts of the initial flow, thus guaranteeing further spatial coherence. Once the optimal path is computed, the per-frame transformations we need to stabilize the video can be obtained like so:

$$transformations = optimal\ path - original\ path$$

These transformations are dense, and the warping can be achieved in a standard remapping manner. First a coordinate grid is initialized that gets accumulated with $transformations$. Now we must remap all image pixels with this new warp field, using a standard interpolation method, bilinear, nearest-neighbor, or bicubic. Applying this to all frames yields the stabilized version of the video.

Chapter Summary

In this chapter, we delved into the formulation and resolution of the Digital Video Stabilization problem through traditional techniques. The fundamental concepts and terminology of computer vision were introduced to provide a solid foundation. I provide my implementation of traditional Digital Video Stabilization methods in this repository https://github.com/btxviny/Trajectory-Optimization-and-Parametric-filtering-based-Video-Stabilization. I provide six algorithms with different motion models, path smoothing techniques and frame warping methods. Moving forward in this thesis, our focus will shift towards exploring the realm of Deep Learning and how it enhances traditional approaches and also extends the spectrum of techniques, particularly through the generation of frames. But first, we must introduce the basic principles and fundamental ideas of Deep Learning.

Chapter 2: Machine & Deep Learning Principles

Introduction

Machine learning, a subset of artificial intelligence, revolves around developing algorithms that enable computers to discern patterns and forecast outcomes without explicit programming. The training process entails harnessing data, refining it through preprocessing and feature engineering, selecting an apt model, and iteratively refining its parameters for optimal performance. Fundamentally, these models are designed for diverse applications such as classification, regression, clustering, and decision-making, representing core aspects of their functionality. In supervised learning, they learn from labeled data to predict target outputs, while in unsupervised learning, they discern inherent patterns within unlabeled data. Reinforcement learning involves predicting actions that yield the most favorable outcomes in a given environment. The applications of machine learning, spanning image and speech recognition to recommendation systems and healthcare, showcase the versatility of these predictive capabilities.

The primary objective of machine learning models is to generalize their acquired knowledge, providing accurate predictions for new, unseen data. Assessing a model's generalization ability occurs during training, employing a distinct dataset known as the validation set. This set serves as feedback for iterative refinement of the model. Following multiple rounds of training and tuning, the ultimate model undergoes evaluation on a test set. This evaluation simulates the model's performance when encountering new and unseen data, ensuring its readiness for real-world applications.

2.1 Generalization

In the realm of machine learning, the concept of generalization refers to a model’s ability to perform well on data it has not been trained on. Two key concepts regarding a model’s performance on new data from the problem domain are overfitting and underfitting.

Overfitting in machine learning occurs when a model learns the training data too closely, capturing noise and random fluctuations rather than the underlying patterns,

leading to excellent performance on the training set but poor generalization to new data. The concept of variance, measuring the model's sensitivity to changes in the training data, is closely tied to overfitting. High variance often results from excessive model complexity, making it overly responsive to training data specifics. This phenomenon occurs when training a complex model, like Decision Trees on a noisy dataset. Managing variance is crucial for finding the right balance between model complexity and generalization ability. Overcoming overfitting involves employing regularization techniques, simplifying the model, or increasing the training dataset size to help the model focus on essential patterns rather than noise.

Underfitting, on the other hand, is a phenomenon where a model is too simplistic to capture the underlying patterns in the training data, resulting in poor performance both on the training set and new, unseen data. It arises when the model is insufficiently complex to represent the inherent complexities of the problem. The term "bias" is closely associated with underfitting, representing the error introduced by overly simplistic assumptions about the real-world problem. High bias leads to underfitting, as the model oversimplifies and fails to grasp the nuances in the data. Simple models, often prone to underfitting are Linear and Logistic regression models. Effectively addressing underfitting involves increasing the model's complexity, employing more sophisticated algorithms, or enhancing the quality of the training data.

Striking the right balance between bias and variance is essential for optimal machine learning model performance, ensuring both effective learning from the training data and adept generalization to new, unseen data. An illustration of overfitting, underfitting and a well-balanced model is offered in Figure 2.2.

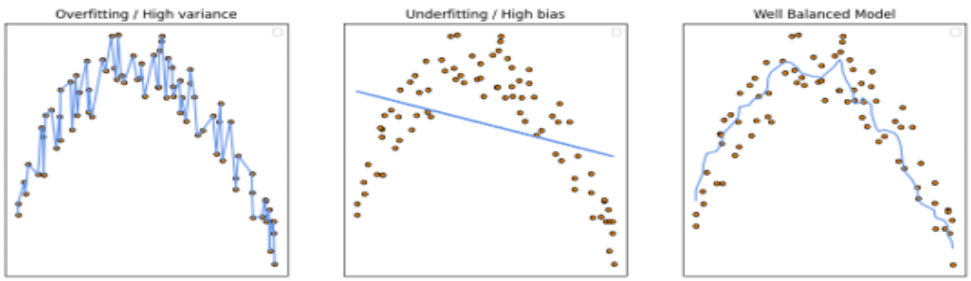


Figure 2.1: Illustration of overfitting a), underfitting b) as opposed to a well-balanced model c).

2.2 Introduction to Deep Learning

Deep Learning is a subset of machine learning, that utilizes neural networks in an attempt to mimic human-like perceptional abilities. Their founding block is the Perceptron module [18], first introduced 1958 which aims to replicate the functionality of a neuron cell. In figure 2.3 a) is an illustration of a neuron cell and in b) the fundamental architecture of a perceptron.

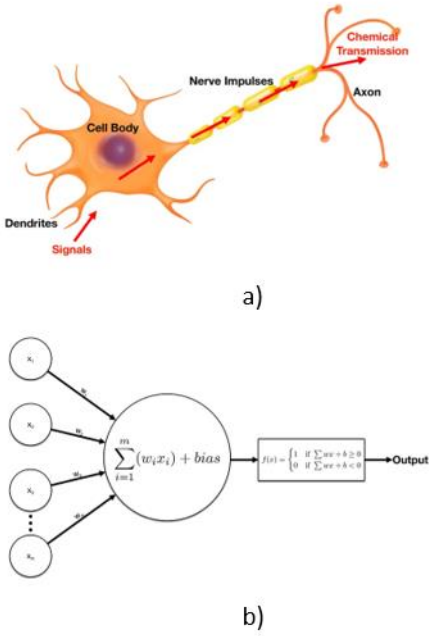


Figure 2.2

The procedure of a perceptron processing data is as follows:

1. On the left side we have neurons of $\boldsymbol{x} \in R^m$ carrying **data** input.

2. We multiply each of the input by a **weight** $w \in R^m$, along the arrow (also called a synapse) to the big circle in the middle. So, we have $w \cdot x = \sum_{i=1}^{m} w_i \cdot x_i$

3. Once all the inputs are multiplied by the corresponding weight, we add another pre-determined number called **bias**.

4. Then, we push the result further to the right through the **activation function**. In this case, we have the step function in the rectangle. What it means is that

if the result from step 3 is any number equal or larger than 0, then we get 1 as output, otherwise if the result is smaller than 0, we get 0 as output.

5. Finally, the output is either 1 or 0.

This is a simple architecture that enables a machine to be able to make predictions / decisions. However, we must be very careful, regarding the initialization of our weights and bias for the model to learn effectively. Researchers soon realized that this perceptron architecture could not learn the XOR (exclusive or) operation, which is fundamental for most decision-making tasks, so we moved onto multi-layer perceptron architectures. These more complicated models, however, now need the so-called concept of backpropagation, which is the mechanism that enables them to learn through trial and error. If we have a model and its actual output is different from the desired output, we need a way to back-propagate the error information along the neural network to tell weights to adjust and correct themselves by a certain value so that gradually the actual output from the model gets closer to the desired output after rounds and rounds of training.

As it turns out, for more complicated tasks that involve outputs which cannot be expressed as linear combination of the inputs, the step function demonstrated above won't work, as the backpropagation mechanism requires the activation function to have a meaningful derivative. In the case of the step function, its derivative is zero for all points except for point zero, where it is undefined since the function is discontinuous there. The significance of non-linearity is also underscored by the prevalence of non-linear patterns in many real-world problems. In scenarios where linear transformations alone are employed, the model's ability to capture the complexity inherent in these patterns is severely reduced.

For this reason, all intermediate layers integrate non-linear activation functions. The most commonly used ones, are the following:

1. **Sigmoid**: The sigmoid function, also known as the logistic function, is a widely used activation function in deep learning. It maps any real-valued number to the range between 0 and 1, making it useful for binary classification problems. The formula for the sigmoid function is given by:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

2. **ReLU(Rectified Linear Unit):** The Rectified Linear Unit (ReLU) is another commonly used activation function in deep learning. It is simple yet effective

and is widely used in various neural network architectures. The ReLU function is defined as follows:

$$\mathrm{ReLU}(x) = \max(0, x)$$

3. **Leaky-ReLU**: The Leaky Rectified Linear Unit (Leaky ReLU) is a variation of the ReLU activation function. It allows a small, non-zero gradient when the input is negative, addressing the "dying ReLU" problem where neurons can become inactive and stop learning during training. The Leaky ReLU function is defined as follows:

$$LeakyReLU(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{otherwise} \end{cases}$$

Here α is a small positive constant, often set to a small value like 0.01. Often Leaky-ReLU is preferred over ReLU as it prevents the phenomenon of vanishing gradients.

4. **Tanh**: The hyperbolic tangent function (tanh) is another commonly used activation function in deep learning. It squashes input values to the range between -1 and 1, making it useful for scenarios where the data distribution has negative as well as positive values. The tanh function is defined as follows:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

All activations mentioned above are depicted in Figure 2.3 in their respective order.

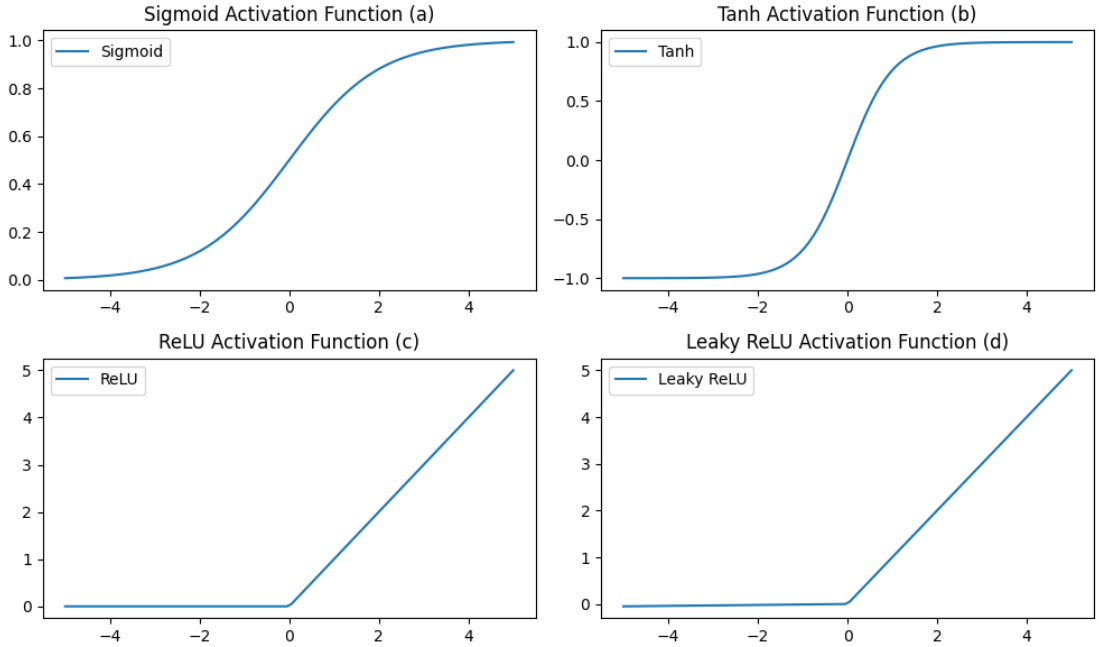


Figure 2.3: Illustration of most commonly used activation functions in neural networks.

2.3 A brief overview of backpropagation, gradient descent, and the importance of learning rate

In this section we will cover how a deep learning network essentially learns through the use of a loss function that indicates how wrong the model was in its current prediction. Then how the gradient descent algorithm determines in which direction each weight needs to be adjusted where the amount of adjustment is dictated by a predetermined parameter called the learning rate. And finally, how these adjustments are applied using backpropagation. The model is fully trained, or in other words it has converged, when we have reached the global minimum of the loss function.

To tie all this concepts together for a better understanding of how a neural network learns, we will break down the training process into steps:

Step 1: Forward Pass

- During the forward pass, input data is passed through the neural network, and predictions are made.
- The output is compared to the expected values using a loss function, which quantifies the error.

Step 2: Gradient Calculation & Backpropagation

- The gradient of the loss with respect to a parameter is calculated by multiplying the gradient of the loss with respect to the layer's output by the gradient of the layer's output with respect to the parameter.
- The chain rule is applied to calculate the gradients layer by layer, starting from the output layer and moving backward through the network. Which essentially is the process of backpropagation.
- This process is repeated for each parameter in the network.

Step 3: Weight Update

- The weights are then updated in the opposite direction of the gradient to minimize the loss.
- The update rule is typically of the form:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_{\theta} \mathrm{Loss}^{(t)}$$

Where θ is the vector of model parameters (weights and biases), with $\theta^{(t)}, \theta^{(t+1)}$ representing the parameters at the current iteration and next

iteration respectively. And finally, η is the learning rate which is the amount of the adjustments made to the weights.

Step 4: Iterative Process

- Steps 1-3 are repeated iteratively until the loss converges to a minimum.

The weight update algorithm (also referred to as an optimizer) used above is known as stochastic gradient descent (SGD). SGD is a fundamental optimization algorithm used for training machine learning models, including neural networks. In each iteration, it updates the model parameters by moving in the opposite direction of the gradient of the loss function with respect to those parameters. However, many improvements have been proposed to improve convergence.

The first major improvement was the introduction of the **momentum** concept. Momentum is an enhancement to the basic SGD algorithm that helps accelerate convergence, especially in the presence of high curvature, small but consistent gradients, or noisy gradients. It introduces a moving average of past gradients to smooth the updates and prevent oscillations. The update rule for the parameter θ_i with momentum is:

$$v_i^{(t+1)} = \beta v_i^{(t)} + (1-\beta)\nabla_{\theta_i}\text{Loss}^{(t)}$$

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \eta v_i^{(t+1)}$$

Where β is the momentum term. Another modification to SGD is the Root Mean Square Propagation (**RMSprop**) algorithm [19]. It essentially adapts the learning rates for each parameter individually based on the root mean square of recent gradients. It helps handle issues with different scales of gradients for different parameters. The update rule for the parameter θ_i in this case is:

$$v_i^{(t+1)} = \beta v_i^{(t)} + (1-\beta)\left(\nabla_{\theta_i}\text{Loss}^{(t)}\right)^2$$

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \eta \frac{\nabla_{\theta_i}\text{Loss}^{(t)}}{\sqrt{v_i^{(t+1)}} + \epsilon}$$

where β is a decay term and ϵ is a small constant to prevent division by zero. Finally, an optimization algorithm that combines ideas from both momentum and RMSprop is Adaptive Moment Estimation (**ADAM**) [20]. It uses moving averages of the gradients and the squared gradients to adaptively adjust the learning rates for each parameter.

The update rule for θ_i with ADAM is:

$$m_i^{(t+1)} = \beta_1 m_i^{(t)} + (1-\beta_1)\nabla_{\theta_i}\text{Loss}^{(t)}$$

$$v_i^{(t+1)} = \beta_2 v_i^{(t)} + (1-\beta_2)\left(\nabla_{\theta_i}\text{Loss}^{(t)}\right)^2$$

$$\widehat{m_i} = \frac{m_i^{(t+1)}}{1-\beta_1^{t+1}}$$

$$\widehat{v_i} = \frac{v_i^{(t+1)}}{1-\beta_2^{t+1}}$$

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \eta\frac{\widehat{m_i}}{\sqrt{\widehat{v_i}}+\epsilon}$$

Where $m_i^{(t+1)}$, $v_i^{(t+1)}$ are the first and second moment estimations respectively. The terms $\widehat{m_i}, \widehat{v_i}$ correct the bias introduced by the initialization of m_i, v_i in the early iterations.

Nowadays, ADAM is the most commonly used optimization algorithm used in Deep Learning, as it has shown great convergence results in numerous applications. However, the choice of optimizer still heavily depends on the specific characteristics of the problem and the dataset.

Previously we mentioned the importance of selecting an appropriate learning rate, as it affects the ability of our model to learn effectively. Here we will attempt to demonstrate this with two cases, where the learning rate's value is too large or too small as shown in Figure 2.4. Two examples of loss functions are plotted with respect to the model's parameters.

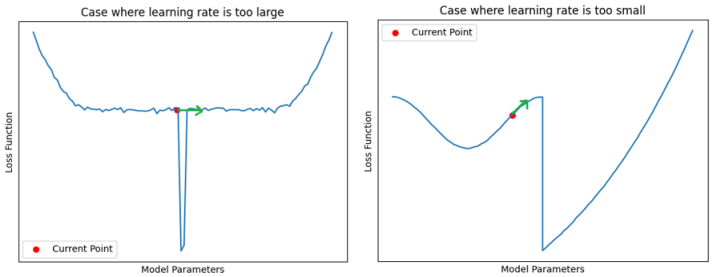


Figure 2.4: Illustration for sub-optimal values for the learning rate

In the first scenario depicted in the upper part of the figure, the learning rate chosen is excessively large. This implies that during weight updates, the step size is too substantial, causing the optimization process to overshoot the optimal solution – the global minimum located at the bottom of the graph. Conversely, in the second scenario where the learning rate is too small, the optimal parameters cannot be

attained. In this case, the model is unable to escape its current region as the gradient points in the positive direction when it attempts to move towards the optimal solution. Consequently, the model remains trapped in its current state, preventing it from converging to the desired global minimum.

One could argue that the learning rate holds paramount importance among hyperparameters and demands careful tuning. Numerous strategies exist for determining an appropriate learning rate. One common practice involves selecting a subset of the dataset and training the model for a brief period with different learning rate values. The resulting loss values are then plotted against the corresponding learning rates. The optimal learning rate is typically identified at the point where the loss value exhibits the steepest rate of change or gradient.

However, this process can be cumbersome and is dependent on the specific model architecture. As a practical alternative, many implementations resort to empirical methods by initially using a standard learning rate, such as 1e-3. If the loss suddenly diverges during training, the learning rate is iteratively decreased until a stable reduction in the loss is observed. This empirical approach provides a quicker and often effective way to set an appropriate learning rate, especially when extensive hyperparameter tuning might be impractical.

There are approaches which dynamically tune the learning rate during training. Essentially, decreasing it as we approach the optimal set of parameters, so as not to overshoot them. One such approach is learning rate scheduling, which entails reducing the learning rate by a certain amount, after a specified number of iterations. Another technique is reducing the learning rate, when no significant improvement in the loss value has been observed after a certain number of iterations.

2.4 Building Blocks of Neural Networks

In this section, we will delve into the foundational layers that comprise a neural network. To begin, we introduce Fully Connected Layers, constructed from stacked perceptrons [18]. These layers form the backbone of Deep Neural Networks (DNNs), enabling the modeling of intricate relationships within data. The interconnected nature of fully connected layers facilitates the learning of complex patterns, making them integral components in various applications.

Following the exploration of fully connected layers, we will shift our focus to Convolutional Layers and their variations, a revolutionary development in the deep learning field. These layers have transformed the landscape of machine learning with diverse applications, spanning from computer vision to natural language processing and beyond. The unique architecture of convolutional layers allows the network to automatically learn spatial hierarchies and invariant features, significantly enhancing performance in tasks such as image recognition.

Finally, we will introduce Regularization Layers, crucial for achieving a balanced bias-variance trade-off, a cornerstone that all machine learning models must navigate. These layers play a pivotal role in preventing overfitting and ensuring the generalization of the model. Together, these fundamental building blocks constitute the intricate framework of neural networks, each contributing to the model's ability to understand and learn from complex data.

2.4.1 Fully Connected Layers (Linear or Dense Layers)

Linear layers consist of stacked perceptron layers. It involves an input vector **x**, trainable connection weights **w**, a bias scalar **b**, and an output unit **y**. The output is determined through the application of a nonlinear activation function **f(**·) to the weighted sum of inputs, achieved via the inner product and is defined as:

$$y(x, w, b) = f(w^T x + b) = f\left(\sum_{i=1}^{D} x_i w_i + b\right)$$

These layers play a pivotal role in constructing Deep Neural Networks (DNNs). By stacking multiple layers of these fundamental units, DNNs gain significant expressive power, enabling them to model intricate patterns within data. This depth and complexity empower DNNs to automatically extract hierarchical features, facilitating their ability to learn and understand nuanced relationships, showcasing their

effectiveness in various applications. A very simple architecture of a DNN consisting of one Fully connected layer is illustrated in Figure 2.5.

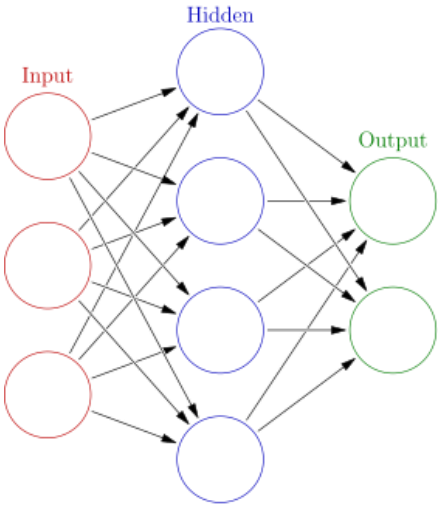


Figure 2.5: A single hidden layer DNN.

2.4.2 Convolutional Layers

This layer performs an operation called convolution. Convolution is a widely used technique in any form of signal processing, including image processing. The Deep Learning field was revolutionized with the introduction of Convolutional Neural Networks (CNNs), with convolutional layers as their building blocks. In the context of a convolutional neural network, a convolution is a linear operation that involves the multiplication of a set of weights with the input, much like a traditional neural network. Given that the technique was designed for two-dimensional input, a matrix multiplication is performed between an array of input data and a two-dimensional array of weights, called a filter or a kernel.

The filter is smaller than the input data and the type of multiplication applied between a filter-sized patch of the input and the filter is a dot product. A dot product is the element-wise multiplication between the filter-sized patch of the input and filter, which is then summed, always resulting in a single value. Some may argue that this operation better describes cross- correlation. Consequently, signal-processing convolution the filter is flipped before the dot product with the input, however in deep learning the flip is not implemented, since it will be eventually learned through training. The process of 2D convolution is illustrated in Figure 2.6.

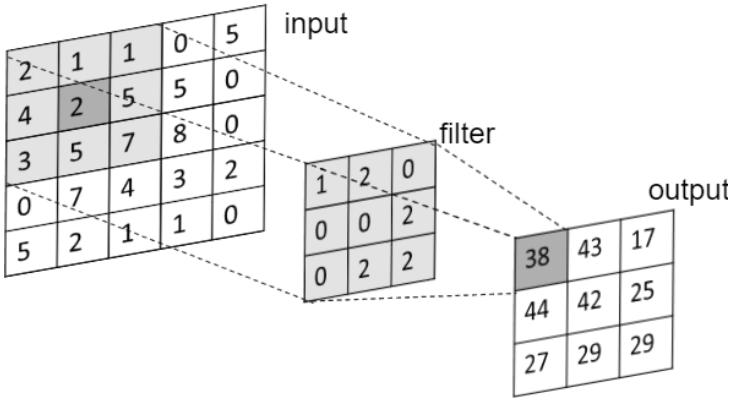


Figure 2.6: Convolution over a 2D grid with a 3 by 3 kernel

The intentional use of a filter smaller than the input in convolutional neural networks (CNNs) serves a specific purpose: it allows the repeated application of the same set of weights across the input array at different positions. Specifically, the filter is systematically applied to each overlapping segment or patch of the input data, progressing in a left-to-right and top-to-bottom fashion. This systematic application of a single filter across the entire image introduces a powerful concept. If the filter is designed to identify a particular feature in the input, applying it systematically across the entire image provides the filter with the opportunity to detect that feature anywhere in the image. This capability is commonly referred to as translation invariance, emphasizing a general interest in the presence of the feature rather than its specific location.

A convolutional layer is defined by many parameters such as:

- **Number of channels**: It signifies the number of channels produced by the convolution. Each channel represents a learned filter.
- **Kenel size**: The spatial dimensions of the convolving kernel/filter.
- **Stride**: The stride determines the step size the convolutional filter takes when sliding over the input. It influences the downsampling of the spatial dimensions of the output feature map.
- **Padding**: Padding adds values around the input data, preventing loss of spatial information as the convolutional filter moves across the input. It helps maintain the spatial dimensions of the input.
- **Dilation**: Dilation introduces spacing between kernel elements, influencing the receptive field. It can help capture information from a broader area while using fewer parameters.

In the context of CNNs, the outcome of applying a filter to the input is known as a feature map. A feature map highlights the presence of certain features or patterns in

the input data. As the filter is moved across the input, it detects relevant features, and the resulting feature map represents the spatial distribution of these features. This process allows CNNs to learn hierarchical representations of features, capturing increasingly complex patterns in deeper layers. The concept of translation invariance, as described in your passage, underscores the network's ability to recognize features irrespective of their exact location in the input image.

In an attempt to better understand the inner workings of CNNs and their hierarchical feature learning, in Figure 2.7 we're visualizing the pretrained kernels of the well-known ResNet [21] at the first and last convolutional layer. This analysis will offer insights into how the network acquires knowledge of patterns and features across different layers.

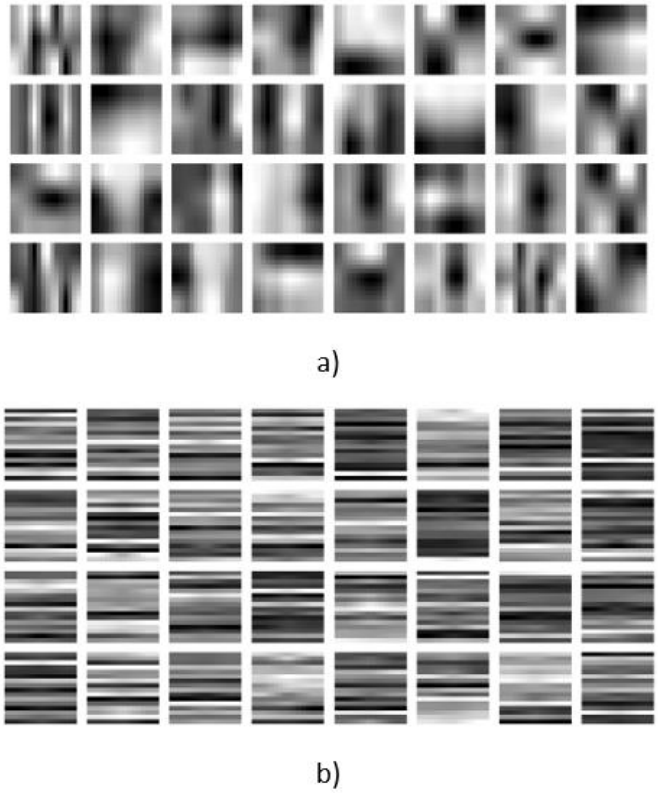


Figure 2.7: a) Visualization of pretrained filters of layer 1 b) Visualization of pretrained filters of the convolutional last layer

The first layer, Figure 2.7.a showcases kernels that predominantly capture low-level details and high-frequency information, such as textures and basic patterns. As we progress through the model, the kernels demonstrate a shift towards more complex and abstract features as shown in Figure 2.5.b, reflecting the network's ability to learn

hierarchical representations. Perhaps this can be better showcased by visualizing the intermediate feature maps of an image passing through the network, as shown in Figure 2.8.

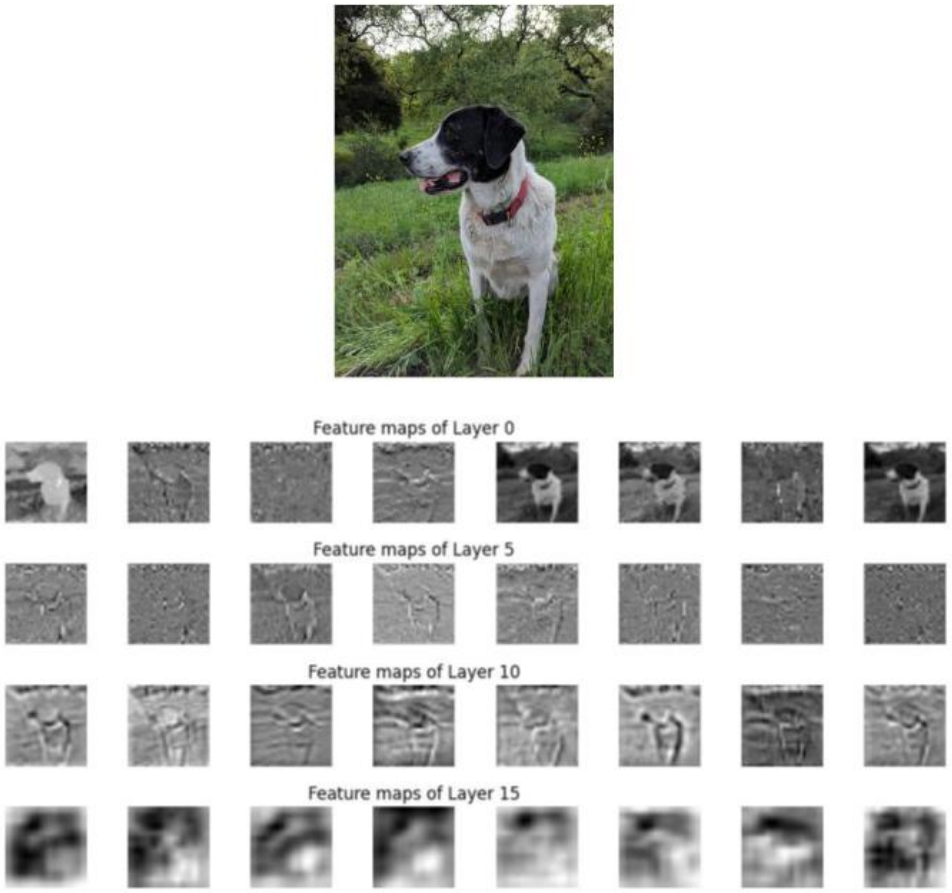


Figure 2.8: Visualization of intermediate feature maps

Starting with layer 0, the feature map illustrates the network's focus on capturing fundamental details and high-frequency information, showcasing a coarse representation of the input image. As we progress to layers 5, 10, and 15, the feature maps become increasingly refined, highlighting more complex patterns and abstract representations. In layer 5, the network begins to discern edges, corners, and basic structures, reflecting the emergence of mid-level features. Layer 10 delves deeper into the image, capturing intricate details and emphasizing specific components that contribute to the overall object recognition. By layer 15, the feature map demonstrates a high-level abstraction, revealing the network's ability to recognize and represent complex semantic information.

In this section we only talked about 2D convolution, however there are also 1D and 3D convolutional layers with many applications in timeseries and image sequence processing respectively.

2.4.3 Regularization and Pooling Layers

As we mentioned, like all machine learning models, neural networks also require regularization. As models become increasingly complex, there is a higher risk of overfitting, where the network memorizes the training data rather than learning the underlying patterns. Regularization techniques aim to mitigate this issue by imposing constraints on the network's parameters, preventing them from reaching extreme values. This helps generalize the model to unseen data, enhancing its performance through achieving a bias-variance trade off.

One common form of regularization is weight regularization, which penalizes large weights in the network. This encourages the model to prioritize smaller, more distributed weights, promoting simplicity and reducing the risk of overfitting. Another widely used technique is dropout, which randomly deactivates a fraction of neurons during training. This introduces a form of ensemble learning within the network, forcing it to be more robust and preventing reliance on specific neurons. In addition to these techniques, normalization layers play a pivotal role in stabilizing the learning process. Batch Normalization, Layer Normalization, and Instance Normalization are three prominent methods. Batch Normalization normalizes the inputs of a layer over a mini batch, addressing issues like internal covariate shift. Layer Normalization, on the other hand, normalizes across features for each individual data point. Instance Normalization extends this concept to normalize each instance independently, making it useful in style transfer and certain image generation tasks. Understanding the nuances of these normalization techniques is vital for constructing stable and effective deep learning architectures. Additionally, larger batches can act as a regularization technique, introducing a form of noise during training that aids in generalization to unseen data.

Pooling layers are essential components in convolutional neural networks (CNNs) that play a crucial role in downsampling the spatial dimensions of the input data. The primary purpose of pooling is to reduce the computational complexity of the network while retaining the essential features that contribute to the overall understanding of the data. There are different types of pooling layers, such as Max Pooling, Average Pooling, and Global Average Pooling, each serving specific purposes.

- **Max Pooling**: Max pooling is a popular choice in CNN architectures. It operates by selecting the maximum value from a set of values within a predefined window or filter. This process helps retain the most prominent features in a given region, allowing the network to focus on the most critical information while discarding less relevant details. Max pooling is particularly effective in capturing the presence of features within an image.

- **Average Pooling**: Unlike max pooling, average pooling computes the average value of the elements within the filter window. This type of pooling provides

a more smoothed representation of the input, making it less sensitive to small variations and noise. Average pooling can be beneficial in scenarios where preserving the general trend of features is more critical than capturing precise details.

- **Global Average Pooling (GAP)**: Global Average Pooling is a variation of average pooling where the spatial dimensions are reduced to a single value by computing the average of all values in each feature map. This results in a more compact representation of the data, reducing the number of parameters in the subsequent layers. GAP has been shown to prevent overfitting and improve model generalization.

The importance of pooling layers lies in their ability to achieve translation invariance, reduce spatial dimensions, and control the model's computational complexity. By summarizing the information in local regions, pooling layers enable the network to focus on the most relevant features while discarding redundant information. This down-sampling also helps in creating a hierarchical representation of the input, allowing the network to capture increasingly abstract and complex features as it progresses through the layers. Overall, pooling layers are vital for effective feature extraction, model efficiency, and improved generalization in convolutional neural networks.

2.5 Advanced Convolutional Network Architectures

In this section we will provide a brief examination of two cutting edge network architectures, namely ResNet and U-Net. ResNet, with its ingenious use of residual learning, challenges conventional limitations in network depth and addresses the vanishing gradient problem. As we delve deeper, we will unravel the architecture's inner workings, exploring the significance of skip connections and how they contribute to the model's remarkable success. On the other hand, U-Net, tailored for semantic segmentation tasks, introduces a unique encoder-decoder structure with skip connections that facilitate precise pixel-wise predictions. Our exploration will encompass an in-depth analysis of U-Net's architecture, emphasizing its suitability for tasks such as image segmentation. By dissecting these advanced architectures, we aim to gain a comprehensive understanding of their design principles, strengths, and applications in the realm of computer vision and deep learning.

ResNet

ResNet, proposed in [21], marked a significant advancement in deep learning architectures, particularly for image recognition tasks. The fundamental concept

behind ResNet is residual learning, where the network learns residual functions instead of directly learning the underlying mapping.
In traditional deep networks, adding more layers could lead to diminishing performance gains or even degradation due to the vanishing gradient problem. This problem occurs when gradients diminish exponentially during backpropagation, hindering the training of early layers as they receive negligible updates. ResNet addresses this issue by introducing skip connections or shortcut connections, enabling the network to bypass one or more layers. These shortcuts create residual blocks, allowing the model to learn residual functions that approximate the identity mapping. This not only facilitates the training of deeper networks but also eases the optimization process.

The residual blocks in ResNet consist of two main paths: the identity path and the residual path. The identity path represents the original mapping that the network is trying to learn, while the residual path captures the difference between the current and target mappings. The addition of these paths results in a more efficient and stable training process.

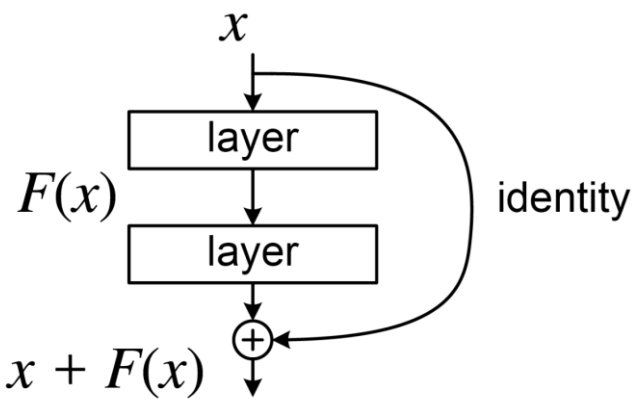


Figure 2.9: Residual Block

ResNet architectures come in various depths, such as ResNet-18, ResNet-50, ResNet-101, and ResNet-152, each with a different number of residual blocks. These architectures have demonstrated exceptional performance on benchmark datasets like ImageNet, showcasing their effectiveness in image classification tasks. Furthermore, ResNet's influence extends beyond image recognition, with applications in object detection, segmentation, and even video analysis.

U-Net

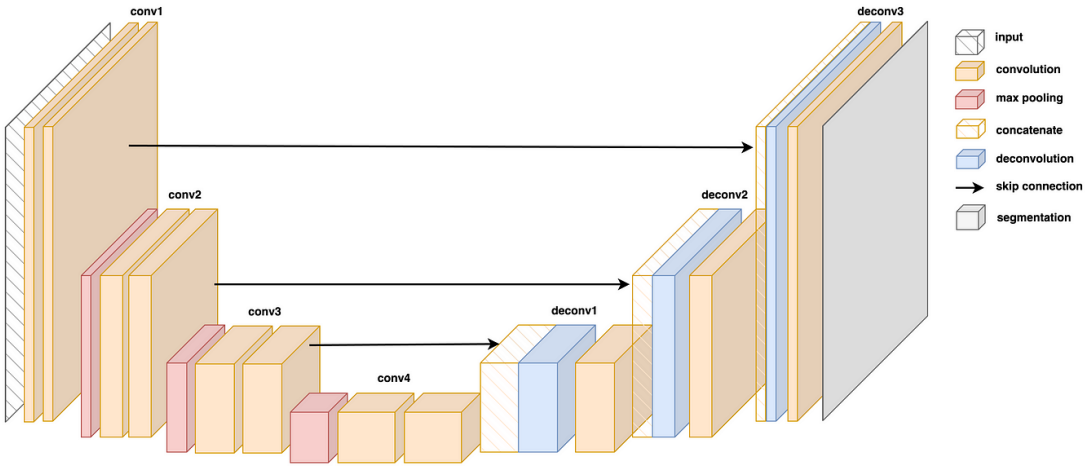


Figure 2.10: U-Net architecture

U-Net, introduced in [22], represents a groundbreaking architecture tailored for semantic segmentation tasks, making it highly influential in computer vision, particularly in medical imaging. The core innovation of U-Net lies in its unique encoder-decoder structure, which includes skip connections between corresponding layers of the encoder and decoder. These skip connections facilitate the incorporation of both local and global contextual information, enabling precise pixel-wise predictions.

In traditional convolutional neural networks, the downsampling in the encoding phase may result in the loss of fine-grained spatial information. U-Net addresses this challenge by introducing skip connections that concatenate feature maps from the encoder to the corresponding layers in the decoder. This allows the network to maintain detailed spatial information during upsampling, enhancing the accuracy of segmentation maps.

The architecture of U-Net resembles the letter "U," with the encoder on one side and the decoder on the other. The encoder captures hierarchical features and reduces the spatial dimensions, while the decoder recovers the spatial information and generates the final segmentation output. This design makes U-Net particularly effective for tasks where precise localization is crucial.

U-Net's success extends across various applications, including its effectiveness in generative models, where it excels in capturing and reproducing intricate details. Its versatility and robust performance have positioned U-Net as a cornerstone architecture in the field of semantic segmentation, proving valuable in tasks that require precise localization and fine-grained feature extraction. Researchers are actively exploring adaptations and enhancements of the U-Net architecture to cater to diverse computer vision tasks, showcasing its continued relevance and impact in the ever-evolving landscape of deep learning.

Chapter Summary

In summary, this chapter has provided a comprehensive overview of deep learning principles, covering key topics such as generalization, neural networks, backpropagation, and advanced convolutional architectures. In the rest of this thesis, we will tackle video stabilization by leveraging the capabilities of deep learning. These include generalization and inference speed, which can enhance preexisting optimization techniques and also expand the problem domain in more generative approaches.

Chapter 3: Deep Learning Video Stabilization Methods Overview

In this section, before we proceed with implementations, modifications, or reviews of previous work we will declutter the domain of Deep Learning Video Stabilization, by breaking it down into three main categories.

3.1 Unsupervised Trajectory Optimization

First, we have trajectory optimization methods, that aim to reconstruct a more stable camera trajectory path and then warp the frames appropriately to generate the stable video. These methods directly build upon pre-existing digital video stabilization techniques. The conditional input to these networks is the unstable motion of the camera and the output is the necessary transformations to stabilize the unsteady frames. This type of learning is unsupervised as we do not utilize any ground truth transformations. The camera path can be modeled in many ways as we saw in Chapter 1. In this thesis I will implement networks that take in both dense per-pixel motion fields as well as sparse grid vertex motion and infer the respective transformation in the same motion model.

The first implementation I will offer is that of **Deep Flow** [23], including some modifications to make it more robust. This algorithm given dense unstable pixel profile trajectories predicts the **dense per-pixel warp field** to stabilize the video. A key issue with this type of trajectory optimization approach is that the input pixel profiles have to spatially smooth in order to not introduce artifacts. We examined the necessity of this in 1.4 were we reviewed SteadyFlow [14] and showcased two-pixel profiles side by side, where one was on constant static background and the other had dynamically moving objects pass through it. The way [14] handled this was an iterative motion inpainting scheme with adaptive thresholds, which led to long inference times. Deep Flow can be viewed as direct improvement on SteadyFlow, that incorporates deep learning both in the motion inpainting and optimization parts of the algorithm. Dynamically moving objects are identified using a semantic segmentation network, and then the authors train a CNN to infer warping fields for each pixel. In my implementation, I replace the semantic segmentation model with the monocular depth estimator MiDaS [24]. I then train the proposed CNN by also including a regularization term to the loss function, to prohibit the network from over-smoothing the original camera trajectories. During testing, I altered the sliding window scheme the authors suggest, by making successive windows overlap. As a final refinement step, I further apply a moving average filter on the resulting trajectory, using PyTorch's [25] average pooling layer to speed up the process.

Then we will review **DUT** [26], an algorithm that tackles the task of video stabilization using motion modeled in **sparse grid vertices.** The choice of sparse grid motion is to make the algorithm more lightweight while at the same time mitigating the effects of

moving objects. The algorithm can be thought of as the deep learning counterpart to MeshFlow [10]. Two pretrained networks are utilized in the motion estimation stage; PWC-Net [8] that infers dense optical flow between adjacent frames and RF-Net that detects feature keypoints. The motion at the detect keypoints gets propagated to the nearby grid vertices using a multi-plane homography strategy that utilizes K-means [27] to segment the motion in two or more planes. Then the authors train two modules, one for motion refinement and one for trajectory smoothing. I will not try to replicate the algorithm, but I will offer an in-depth review and evaluation. One contribution I will provide is an alternative warping method to produce dense warp fields from the sparse grid vertices, that aims to suppress black border artifacts.

Regarding path optimization approaches, we will also examine [28] **CNN weight space** which is the predecessor of Deep Flow [23]. This algorithm does not train a neural network to generalize on a large dataset for video stabilization. Instead, the CNN is trained on each specific case, and is used to reduce the number of variables that would have to be optimized otherwise. This method is included as it showcases a totally different application of neural networks.

All the aforementioned methods optimize the camera trajectory in 2D space, there are however novel approaches that implement 3D geometry optimization. [29] utilize pretrained deep learning frameworks to reconstruct the 3D camera trajectory, which is then optimized in a CNN weight space. [30] combines optical flow with gyroscope and OIS data to train an LSTM-FN network to infer virtual camera poses.

3.2 Supervised Learning Approaches

These methods utilize labeled data for training and follow a more conventional deep-learning approach. The labeled data in the domain of video stabilization can have two formats, ground truth warping transformations and ground truth stable frames. A more in-depth discussion on the available datasets and their limitations will be offered in the upcoming chapter. I will implement two different supervised algorithms.

Firstly, I implement **StabNet** [31], a network that given a sequence of previous stable frames accompanied with the incoming unsteady frame, infers a multi-grid transformation to stabilize the video. The training data used are pairs of stable/unstable video pairs. The ground truth transformations are not provided, instead the inferred transformation is utilized to warp the incoming frame and directly compare it with the ground truth frame. The network is trained in a Siamese scheme, meaning that two branches of the model are instantiated, each receiving temporally successive inputs and their results are utilized in a temporal smoothness loss. The authors intend the algorithm for online applications and only utilize previous frames in grayscale format as input. Instead, I will train a model that takes in 5 unsteady frames, both previous and future unsteady frames in RGB format and use the inferred transformation to stabilize the intermediate frame. The original model utilizes a

pretrained ResNet-50 [21] as a backbone encoder for the input sequence which I replace with VGG-19 [32] which is widely used for perceptual losses and neural style transfer.

A second supervised method I implement is **Deep Motion Blind Video Stabilization** [33]. In contrast to StabNet, this network is trained in a supervised manner to directly produce stable frames given an unsteady frame sequence. This approach might seem more straightforward but has its own difficulties regarding the quality of the available training datasets, which are addressed though synthetic dataset generation. The network will be trained in a three-stage process, two of which implement adversarial training. The last stage also incorporates a pretrained video action recognition network that will be used to enforce further stability. For the model architecture the authors propose using a deep ResNet based architecture that takes in 5 consecutive unsteady frames $\{I_{t-2}, I_{t-1}, I_t, I_{t+1}, I_{t+2}\}$ and infers the stabilized version of the intermediate one. Alongside the proposed network I also train a U-Net model that incorporates gated convolutions [34] and has a modified input of 5 frames sampled uniformly over a second $\{I_{t-16}, I_{t-8}, I_t, I_{t+8}, I_{t+16}\}$. Modifying the network architecture was an attempt to improve on the method's inference speed, while modifying the conditional input was to see if further stability could be achieved.

The choice to implement the above methods was due to the unique approach each one takes to address video stabilization. However, there is plethora of other methods such as [35] which utilizes a U-Net with embedded STN [36] modules that is trained in an adversarial manner with an image and image sequence discriminator along with a VGG-19 [32] based perceptual loss. [37] addresses video stabilization specifically for selfie video scenes, utilizing a pretrained foreground detection CNN and learn a 1D auto-encoder to infer warp nodes, used in a moving least squares warping method. [38] train a U-Net architecture to directly infer multi-grid warp fields from previous frame flow, utilizing ground truth unstable to stable transforms from a synthetic dataset

3.3 Frame interpolation

Video stabilization through deep frame interpolation is a technique that addresses the challenges of shaky or jittery videos by generating intermediate interpolated frames. This idea was firstly introduced in [39] and states that the interpolated middle frame serves as the representation of the frame that would have been recorded between two consecutive frames. Consequently, the interpolated frame is indicative of the temporal midpoint, presumed to align precisely with the halfway point of inter-frame motion. When this process is run in an iterative manner, we can achieve visually pleasing, noise free

results. To demonstrate the capabilities of this technique I will offer my own implementations of **DIFRINT** [40] and Video Stabilization using **CAIN** [41] for frame interpolation.

DIFRINT is comprised of the pretrained PWC-Net [8] motion estimation network, followed by a U-Net frame interpolator and a ResNet refinement module. The training in the original paper takes place in a self-supervised manner. The inputs to the U-Net are the frames f_{i-1}, f_{i+1} warped towards the intermediate frame f_i and the output is f_{int}. However, the authors use a randomly translated version of the middle frame as the ground truth frame. During inference, when the process is run iteratively, blur is bound to accumulate. For this reason, the ResNet module is utilized to re-introduce lost information from the original middle frame onto the interpolated result. I tried to replicate the results of the original paper using their model architecture and training scheme, but the resulting interpolated frames were very blurry. I then proceed to alter the network's architecture regarding the hidden channel size, and also modify the input to include the computed optical flows. Finally, I was able to drastically improve the quality of the generated images by fine-tuning the framework on a new dataset specially crafted for frame interpolation.

I then proceeded to use the state-of-the-art frame interpolation network **CAIN** for the task of video stabilization. A major advantage is that the framework no longer relies on optical flow estimation, and instead tackles frame interpolation using its novel architecture that incorporates the channel attention module. The stabilization process is very similar to the inference scheme used in DIFRINT, and it was actually used for the synthetic dataset generation in [33]. My contribution is training the refinement ResNet module explained in [33] to reintroduce high-level details to the interpolated results which are lost after a number of iterations.

Chapter 4: Datasets and Evaluation Metrics

4.1 Datasets

Both for unsupervised and supervised approaches, the choice of training data is of crucial importance. The quality of the training data will directly influence the capabilities of our models. For example, if our dataset contains a disproportional amount of a certain type of camera motion such as panning, the models won't be able to perform well in other types of motion such as zooming. The ideal dataset should contain different types of unwanted camera motion, across a variety of scene layouts, so that the trained models will be able to generalize well on real world scenarios.

Figure 4.1: The setup on the left was used for capturing DeepStab [31] comprising of two GoPro Hero 4 Black cameras, whereas the setup on the right was used in [42] and uses two different cameras.

For unsupervised deep learning methods, the requirements are a large number of image sequences with a variety of motion patterns. Collecting such data becomes a much harder task in the case of supervised approaches. These datasets typically consist of stable/unstable video pairs. This kind of training data allows for the use of simpler loss functions, where the output of the network can be directly compared to a ground truth frame or transformation. The availability of such datasets is limited, as they must be captured through specialized gear or be synthetically created. Two datasets comprising of real-world video pairs are [31] and [42]. In both datasets the hardware configuration was a pair of cameras laid horizontally next to one another, with one mounted on a handheld stabilizer and the other left to move freely with the motion of the person capturing the video. In Figure 4.1 both hardware configurations can be seen.

Though the distance between the two cameras is small, they introduce a slight non negligible perspective mismatch as illustrated in Figure 4.1. We want to mitigate this

effect, so that a network can focus on the stability of the sequences. The inconsistency in the perspective makes it difficult for a model to learn the direct pixel-level, spatio-temporal relations of unstable videos to their stable counterparts. [42] addresses this use through image alignment. They warp the frames according to 'as-similar-as-possible' warping to align the corresponding frames. However as two different camera sensors are used, the corresponding frames differ slightly in terms of brightness and color representation. Another limitation of the aforementioned video pair datasets is the lack of motion pattern and color variation.

Figure 4.2: A case where perspective mismatch is apparent between a pair of corresponding frames, included in DeepStab [31].

Many methods address these limitations by generating a synthetic dataset. In [23] the authors utilize the RealEstate10K [43] dataset which contains stable videos with a large number of color variations. They generate the unstable counterparts by perturbing the stable frames with random affine transformations. The parameters of these transformations are sampled from uniform distributions. However, they augment each frame individually and don't treat the camera noise as a time series. I will offer an in-depth review of synthetic dataset generation techniques in the last chapter of this thesis.

With all these considerations in mind, I provide a list of the most representative datasets used in video stabilization.

- The **HUJ** dataset proposed in [44], consisting of 42 video sequences that lie in the following categories: walking, zooming, driving and dynamic scenes.
- The **NUS** dataset [12] which contains 144 video sequences which include simple, quick rotation, zooming, parallax, driving, crowd and running
- The **MCL** dataset introduced in [45], comprising of 162 videos classified in the following categories: simple, rolling shutter, depth, crowd, driving, running and object scenes.

- The **BIT** dataset [42] which contains 45 stable/unstable video pairs belonging to 10 categories: walking, climbing, running, riding, driving, large parallax, crowd, near-range object, and dark scenes.
- The **DeepStab** dataset [31] which contains 61 stable/unstable videos of various scenes such as: indoor scenes with large parallax, and common outdoor scenes with buildings, vegetation, crowed. Varying camera motion is present such as walking, panning, quick rotations and any combination of the above.
- The **Selfie** dataset [37] containing 1000+ scenes of selfie videos. The frame sequences are accompanied by foreground-background segmentation masks.
- The dataset proposed in [30] which contains 50 unstable videos accompanied by the respective gyroscope and OIS (Optical Image Stabilization) data.

In Chapter 7 we will review and implement video stabilization through deep learning frame interpolation. This approach no longer needs strictly unstable or stable-unstable video pairs. The goal is to train a model to produce intermediate frames. For this reason, an ideal dataset should contain a variety of scenes with color and optical flow variation with no restrictions regarding the camera motion. Commonly used datasets include:

- The **UCF101** dataset [46] featuring 101 human action classes, captured in diverse environments "in the wild," serving as a valuable resource for training and evaluating models in action recognition tasks.
- The **GoPro** dataset [47] deep video deblurring for handheld cameras, offering video sequences captured with GoPro cameras in various challenging conditions. The ground truth images can be used for frame interpolation as they include a variety of scenes and dynamic motion.
- The **KINETICS-400** dataset [48] contains videos from 400 action classes specially crafted for video action recognition. The variety of dynamic motion makes it a great option for frame interpolation as well.
- The **DAVIS** dataset [49] which contains more than 8000 images with challenging scenarios such dynamic motion and occlusion.
- The **Vimeo-Triplet** [50] which contains 80.000 triplets of images, specifically crafted for frame interpolation.

4.2 Method Evaluation and Quality Assessment

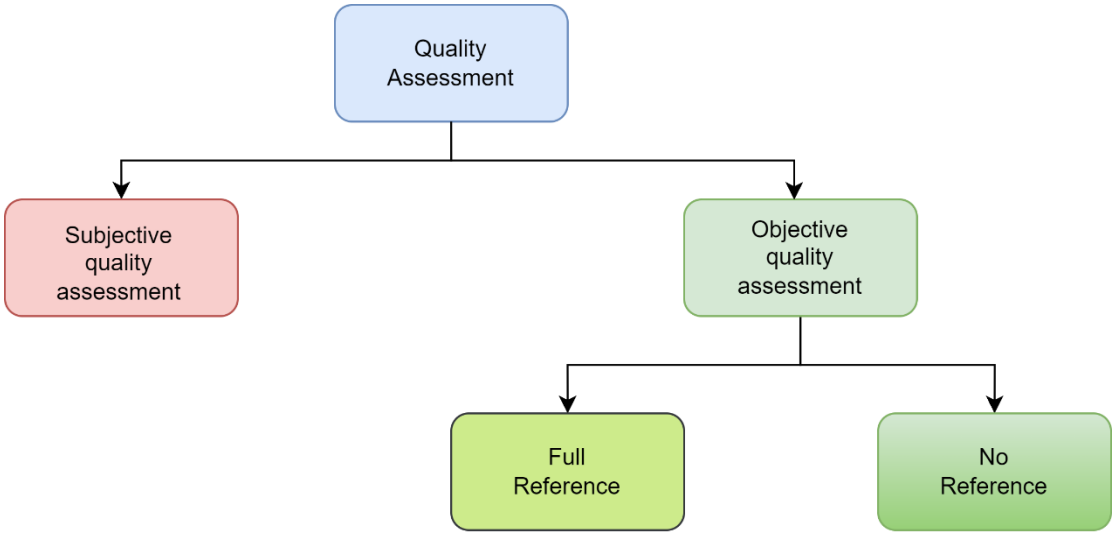


Figure 4.3: Quality Assessment

Video stabilization quality evaluation is divided into two categories: subjective and objective assessment. Subjective assessment is based on people's opinions based on the human visual system. User studies are executed based on the Mean Opinion Score method [51]. This process, however, is time consuming especially when run for large-scale datasets. An alternative are objective assessment methods, which use metrics that reflect the quality of the generated results. According to the availability of the ground truth stable video these methods are further divided into full-reference and no-reference.

Common metrics used in full-reference methods are Peak-Signal-to-Noise-Ratio, Mean Squared Error, Structural Similarity Index (SSIM) and more. In [42] the authors propose an algorithm based on a Riemann metric to assess the quality of the generated video opposed to the ground truth stable one. However, capturing a synchronized video pair needed for full-reference methods can prove challenging for all the reasons mentioned previously. As a result, no-reference methods are more commonly used by researchers. These methods do not require the ground truth stable video and rely on a statistical model to access quality. Through the course of this thesis I will use four objective metrics to quantitatively evaluate each method's performance. These metrics include: 1) **cropping ratio** 2) **global distortion** 3) **pixel loss** and 4) **stability**. We will interpret them as scores, and a good result should have a value close to 1.

These metrics except the pixel loss were defined in [12]. We begin by fitting a global homography H between the input unstable video and the generated result. The cropping ratio between a pair of frames is obtained by the scale component of the homography transformation, and the final cropping score is the mean cropping

between all frames. The distortion is calculated from the anisotropic scaling of H which is estimated as the ratio of the second largest eigenvalue of the affine part of H over the largest eigenvalue. From all distortion scores we chose the lowest one as the final score, which can be interpreted as determining whether the whole result or not. In chapter 6 ,7 I introduce generative networks that infer practically new stable frames. We need a way to measure the photometric loss between the original and generated frames. For this purpose, I normalize the pixel intensities in the range [0,1] and estimate the mean squared error for each image pair. The final score is obtained by subtracting the average error of all pairs from 1, to bring in accordance with the other metrics. As for the stability score, we construct the camera path of the generated video through global homography transformations between successive frames. We split the path into horizontal, vertical, and angular trajectories. Each one of these trajectories is transformed into the frequency domain using FFT [52]. We exclude the DC component and take the energy from the second to sixth lowest frequency. Then the percentage of this energy over the overall energy is the stability score of the trajectory. We take the average of the three trajectory scores as the final stability score.

For my evaluation dataset we will use the one provided in [12]. We take 5 videos from different categories of camera motion and scene layouts. These categories include parallax, quick rotation, zooming, crowd and regular unstable videos. The average score in each category will be used to compare methods. As a demonstration I provide the evaluation scores of my implementation of MeshFlow [10] which we reviewed in Chapter 1.

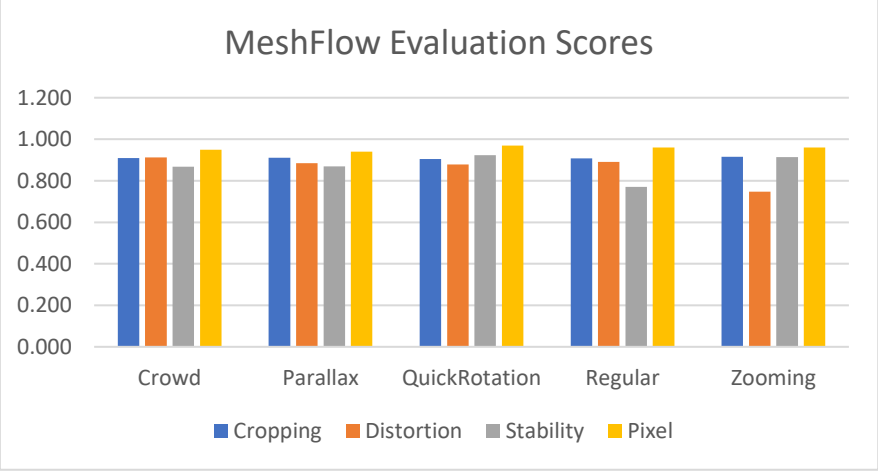


Figure 4.4: Evaluation Scores for the MeshFlow [10] stabilization algorithm.

The evaluation dataset and metrics code can be found in this repository https://github.com/btxviny/Trajectory-Optimization-and-Parametric-filtering-based-Video-Stabilization

Chapter 5: Deep Trajectory Optimization

Introduction

In this chapter, we will review learning-based path optimization methods. These algorithms utilize deep learning to generate a stable camera trajectory and generate the stabilized video using a variety of frame warping techniques. In these approaches, the input to the networks is the unstable camera motion, and the output comprises the required transformations for stabilizing the unstable frames. This form of learning is unsupervised, as it doesn't rely on any ground truth transformations. We will review three such methods and I also offer some improvements and modifications for the first two.

5.1 Implementation of Deep Flow [23]

In [23] the authors propose a model that takes in dense optical flow from a 20-frame window and predicts dense per-pixel warps stabilizing each individual pixel profile. This type of motion modelling is identical to SteadyFlow which we examined in 1.4. There we discussed that directly smoothing pixel profiles can introduce severe image distortions around dynamically moving objects. The same limitation applies here, but instead of inpainting discontinuous motion regions through optimization, we will utilize a technique based on Principal Component Analysis. Finally, the unstable video is obtained by smoothing the original camera path, using a sliding window scheme. We will delve into specific details of the motion estimation stage, training scheme and the inference algorithm. We will discuss some modifications in the motion inpainting process, the objective function, as well as the inference scheme, that aim to improve overall performance. The pipeline of the proposed algorithm is shown in Figure 5.1

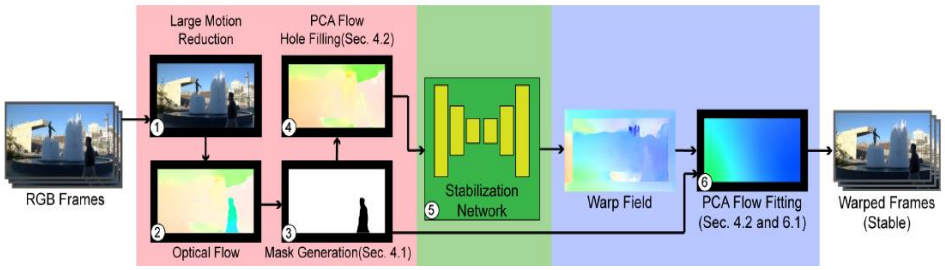


Figure 5.1: [23] The three stages of the Optical Flow based stabilization algorithm.

5.1.1 Input Preprocessing

Large Motion Reduction

Before we extract the dense optical flows, of an input unsteady video we must remove large motions. This first preprocessing part is achieved through a simple feature trajectory smoothing algorithm. The authors detect matched SURF features [2], between consecutive frames and estimate the affine transformations. By extracting the translation and rotation components, of the transformations they construct the unstable camera path. This trajectory is smoothed using moving average filter with a window of size 40. From the difference of the smooth and unstable trajectories they obtain the necessary affine transformations to alleviate the video from large abrupt motion. In my implementation of this algorithm, I used SIFT features [3] since SURF is still under patent. I matched the detected features between two frames using a brute force matcher with KNN classification [53] and then applied a ratio test to filter out bad matches. The brute force matcher is employed to compare each feature in one frame to every feature in the other. It computes the distances of the feature descriptors between all pairs and then selects the one with the smallest distance as the best match. After the brute force matching, the ratio test is applied to filter out unreliable matches. The basic idea is to consider the ratio of the distances of the two nearest neighbors for each feature. If the ratio is below a certain threshold, 0.75 in my case, the match is kept; otherwise, it is discarded. Another deviation from the proposed algorithm is that I used an exponential weighted moving average filter, which I found to introduce fewer black borders in the generated results. Its formula is:

$$Y_t = \alpha \cdot X_t + (1 - \alpha) \cdot Y_{t-1}$$

Where Y_t is the EWMA at time t, X_t is the input value at time t, Y_{t-1} is the EWMA at the previous time step $t - 1$ and α is the smoothing factor that satisfies $0 < \alpha < 1$. The smoothing factor α determines the weight given to the most recent observation. A higher α gives more weight to the latest data, making the EWMA more responsive to recent changes. Conversely, a lower α places more emphasis on historical data, resulting in a smoother but less responsive curve. In my implementation, I chose $\alpha = 0.2$.

Dense Optical Flow Estimation

Following the Large Motion Reduction step, we can extract the dense Optical Flow. The authors do this utilizing the pretrained network FlowNet2 [54], I however used the PWCNet [8], which has shown better results.

Mask Generation

Before providing the extracted flows to the model for training or testing we must inpaint discontinuous motion regions. First, I will go over how to identify such regions and then to the inpainting process. There are four types of conditions that may

potentially lead to flow reliability issues: 1) Dynamically moving objects that do not match camera motion. 2) Inaccurate flow at such object boundaries. 3) Inaccurate flow at texture-less uniform color regions. 4) Large motion of still objects as a result of parallax.The goal is to devise a binary mask M, where $M = 1$ corresponds to valid regions and $M = 0$ otherwise. Below, I provide the scheme for identifying the four types of regions mentioned previously.

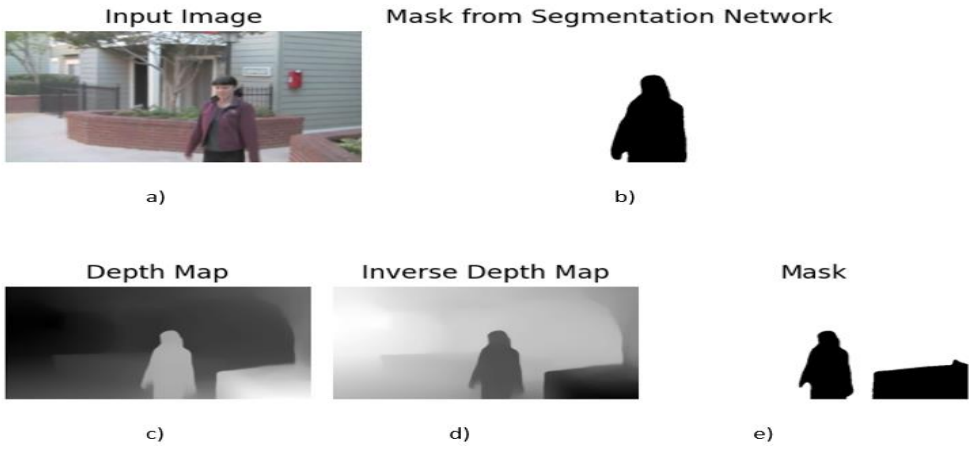


Figure 5.2: Comparison of the proposed mask initialization scheme (upper row) as opposed to my implementation (lower row). A) Depicts an example input image. b) This mask is the result of the segmentation network proposed in the paper. c) The depth map of the image generated by thy MiDaS network. d) The inverse of the depth map. e) The generated mask.

1. For detecting **dynamically moving objects**, the authors utilize a pre-trained semantic segmentation network [55]. With this network they are able to detect objects belonging to the following 11 categories: person, car, boat, bus, truck, airplane, van, ship, motorbike, animal and bicycle. This model infers a binary mask where the pixels belonging to such objects have a value of 1. However, these objects are not necessarily moving in the scene. The mask is initialized as $M = 0$ on these objects (essentially the inverse of the mask inferred by the segmentation model) and then they set $M(p) = 1$ for any pixel p that satisfies the condition $\|F_n(p) - \bar{F}_n\|_2 < 5$, where F_n is the dense optical flow from frame I_n to I_{n+1} and $\bar{F}_n$ is the mean motion of the entire frame. Segmenting scene objects in this way is a practical approach, but it does not address the possibility of the presence of other kinds of large foreground objects. To make the mask initialization process more robust, inspired by [56] which highlights the relation of scene depth and optical flow, I propose using the pretrained depth estimation network MiDaS [24]. This model infers a depth map, as shown in Figure 5.2.c, where foreground objects are assigned a value of 1 and with increasing depth the value decreases. I propose

inverting this depth map, where the closest pixel gets the value 0 and the furthest pixel gets a value of 1, by:

$$\text{Inverse depth} = \frac{\text{depth.max}(\,) - \text{depth}}{\text{depth. max}(\,)}$$

Using this inverted depth map illustrated in Figure 5.2.d, I can now create a binary mask by setting each pixel p with $\text{Inverse depth}(p) < 0.4$ to have $M(p) = 0$. I chose the value 0.4 empirically, as it produced satisfactory results as shown in Figure 5.2.e. Large foreground objects, whether dynamically moving or not can cause inaccurate flow estimation. In the mask I generated we can see the brick structure which is closest to the camera being masked as well.

2. For addressing the second type of regions, **moving object boundaries**, the authors propose detecting regions with large local standard deviation of optical flow. More specifically, they compute the standard deviation using a $5x5$ kernel, thus creating
3. the map ΔF_n. Then set $M(p) = 0$ if $\Delta F_n(p) > 3\overline{\Delta F_n}$, where $\overline{\Delta F_n}$ is the mean standard deviation of the optical flow. The resulting mask based on this condition can be seen in Figure 5.3.b

4. To deal with **uniform color regions**, we compute the magnitude of the gradients of image I_n , ∇I_n and set $M(p) = 0$ when $\nabla I_n(p) < 10$. This condition successfully masks out texture-less regions as shown in Figure 5.3.c.

5. And finally, to mask out **motions of still objects due to parallax** we set $M(p) = 0$, if the motion of the pixel is sufficiently larger than the mean frame motion, $\|F_n(p) - \bar{F}_n\|_2 > 50$.

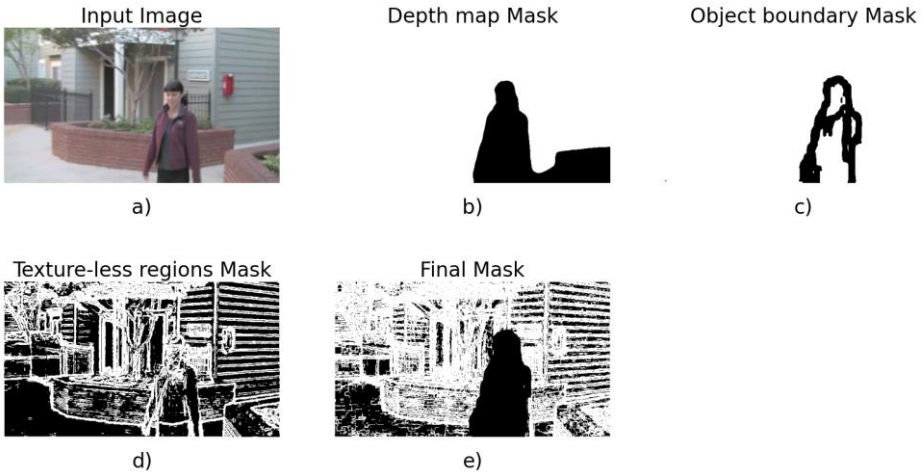


Figure 5.3: Illustration of the mask generated by each step independently b-d and the final mask used for inpainting e.

PCA Flow Hole Filling

Now that we have the masks dictating which regions to inpaint, we will go over how the inpainting procedure is achieved and explain the Principal Component Analysis concept.

Let's start by explaining the concept of Principal Component Analysis (PCA). This technique is used for dimensionality reduction and feature extraction. Its primary goal is to transform a dataset with potentially correlated features into a new set of uncorrelated features, called principal components. PCA transforms the original features into a set of linearly uncorrelated features, the principal components. This decorrelation aids in highlighting the intrinsic patterns in the data and can make it easier to interpret and analyze. This process involves the following steps:

Data Standardization:

- We begin by standardizing the dataset to have zero mean and unit variance. This step ensures that all features are on a similar scale.

Covariance Matrix:

- Then we calculate the covariance matrix of the standardized data. This matrix contains information about the relationships between different features.

Eigenvalues and Eigenvectors:

- This step involves finding the eigenvectors and eigenvalues of the covariance matrix. Eigenvectors represent the directions of maximum variance, and eigenvalues indicate the magnitude of variance in those directions.
- We then sort the eigenvectors in descending order based on their eigenvalues. This step helps in selecting the principal components associated with the highest variance.

Principal Components:

- Now we select the top k eigenvectors to form the principal components, where k is the desired reduced dimensionality. These eigenvectors represent the most significant directions of variation in the data.

Projection:

- Project the original data onto the selected principal components. This is done by multiplying the standardized data by the matrix of selected eigenvectors. The result is a new set of features, the principal components, that capture the essential information in the data.

In PCAFlow [13] the authors computed the principal components of optical flow extracted from a number of Hollywood movies from several genres. They selected 180,000 frames and computed 250 principal components for the flow in the horizontal

and vertical individually. Each of these components have 256 by 512 spatial dimensions but can be resized based on the specific application. In Figure 5.4 we visualize the first 10 principal in the horizontal and vertical direction.

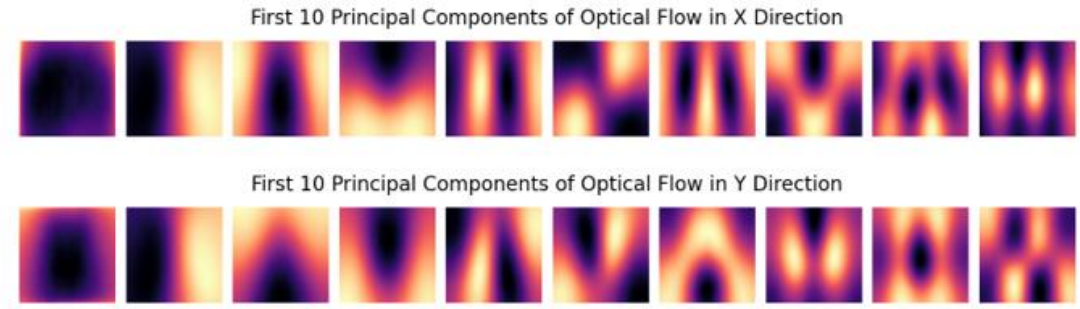


Figure 5.4: Visualization of the first 10 horizontal and vertical principal components of optical flow provided from PCAFlow.

For our inpainting method, we want to express the valid regions of the extracted dense optical flow, as a linear combination of the first 5 principal components. The valid regions are dictated by the mask, which we will introduce later, where $M_n = 1$. We will then fill the regions where $M_n = 0$ with the PCAFlow. To do this, we first have to estimate PCAFlow by solving a least squares problem to determine the weights for each of the five principal components. Since the first 5 principal components of PCAFlow are spatially smooth, we can expect the Mn = 0 regions to be filled with reasonable values that obey the overall optical flow field. The authors start by stacking the resized horizontal and vertical components into matrices $Q_x, Q_y \in R^{wh\,x\,5}$. Similarly, they reshape the optical flow field to $F_{n,x}$ and $F_{n,y}$. Finding the coefficients $c_n \in R^{5x1}$, can be expressed as the following least squares problem:

$$\min_{\mathrm{c_n}} \|Q_n^{\sim} c_n - F_n^{\sim}\|_2 + \eta \|\mathrm{c_n}\|_2$$

Where $\eta = 0.1$, is the regularization term. The solution of this problem is:

$$c_n = (Q_n^{\sim T} Q_n^{\sim} + \eta I)^{-1} Q_n^{\sim T} F_n^{\sim}$$

The results of this inpainting method can be seen in Figure 5.5 below.

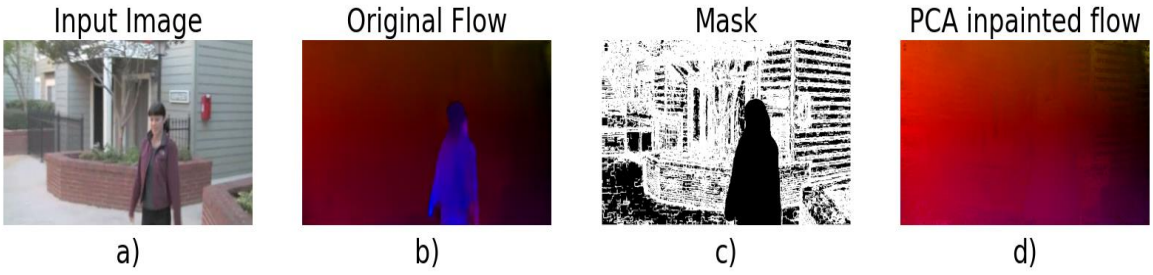


Figure 5.5: Illustration of the hole inpainting process. The moving person has been fully inpainted with reasonable values based on the overall frame motion.

In addition to flow inpainting, this approach is also valid for any sparse to dense flow approximation task. We will use it later in this thesis for approximating a dense flow field from a sparse vertex grid and compare it to the Mosaic multi holography technique. This input preprocessing pipeline, which is used both for preparing the training dataset, but also during inference time for video stabilization.

5.1.2 Network: Architecture, Input & Loss functions

Architecture

The network that we will use is based on the structure proposed by [43]. It is based on a modified U-Net architecture. This model utilizes skip connections to enhance information flow between encoder and decoder stages, facilitating the preservation of fine-grained details during spatial dimension reduction. In deeper convolutional layers, dilation is incorporated enabling the capture of larger contextual information, particularly beneficial for inferring optical flow fields. In the decoder part of the network, Skip connections are introduced by concatenating feature maps from the encoder with those from the decoder, promoting the preservation of spatial details lost during the encoding process.

Input

As the model will inevitably have a fixed number of input channels, it will only take in a segment of the video at a time. The choice of the number of channels is non-trivial, as it can have a major effect on the quality of the generated result. Using larger segments can enable the model to handle low-frequency noise better. This is however at the cost of inference speed and resources. After careful consideration, the authors decided on a 20-optical flow field input, representing a 21-frame segment of the video.

Loss functions

In the original paper, the authors use a combination of two terms as a loss function, the stability term, and the spatial coherence term.

Stability term $\boldsymbol{L_m}$: This term enforces the warped position of a pixel $p_{i,n}$, denoted as $\widehat{p_{i,n}}$ and the warped position of its correspondence in the next frame $q_{j,n+1}$, $\widehat{q_{j,n+1}}$ to be as close as possible. With the PCA inpainted input flow denoted as $\widehat{F_n}$ the correspondence of can be expressed as:

$$q_{j,n+1} = p_{i,n} + \widehat{F_n}(p_{i,n})$$

With the networks output warp field denoted as W_n the warped position of $p_{i,n}$ can be written as:

$$\widehat{p_{i,n}} = p_{i,n} + W_n(p_{i,n})$$

And finally, the warped position of the correspondence is defined as:

$$q_{j,n+1} = q_{j,n+1} + W_{n+1}(q_{j,n+1})$$

With all the definitions in place the stability term of the loss functions is:

$$L_m = \sum_{n=1}^{N-1} \sum_{i=1} \left\| \widehat{p_{i,n}} - \widehat{q_{j,n+1}} \right\|_2$$

The aim of this loss function can be better understood through its visualization in Figure 5.6.

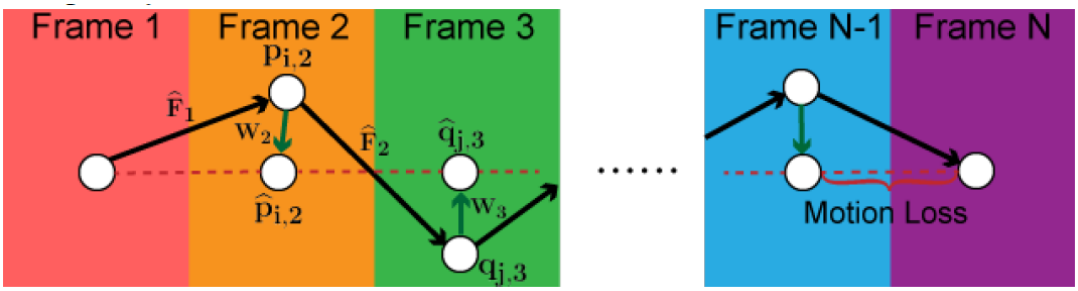


Figure 5.6: [23] The L_m term penalizes the distance between corresponding pixels in adjacent frames.

Spatial coherence term L_f: This term aims to enforce spatial smoothness of the generated warp fields. It aims to mitigate high frequency noise while at the same time avoiding local distortion. This essentially means forcing the network to 'apply' a low-pass filter on its outputs. Since the optimizers used for training are designed to minimize functions, instead of maximizing the outputs low-frequency content, we will minimize its high-frequency content. To do this will apply a high pass filter on the generated warp fields and penalize the result.

The first step is to obtain the frequency representation of the warp fields, through a two-dimensional Fast Fourier Transformation. Next, we will multiply this spectrum representation with a high pass filter. This filter mask is obtained by inverting a Gaussian map G with $\mu = 0, \sigma = 3$. The inverted gaussian can be obtained by:

$$\hat{G} = (\max(G) - G) / \max(G)$$

The resulting mask is illustrated in Figure 5.7.

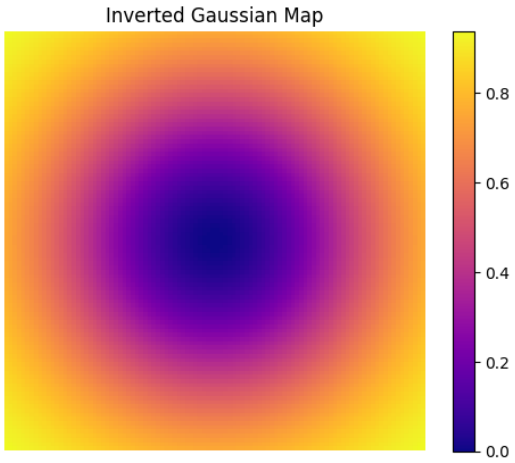


Figure 5.7: The inverted gaussian map, that attenuates low frequencies and highlights high frequencies.

5.1.3 Implementation Details

Dataset

Training this network will require a large number of unstable videos. The authors argue that DeepStab [31] lacks in motion patterns and color variations. For this reason, they opt for the RealEstate10K [43], which contains stable videos with a large number of color variations. Then they proceed to synthesize the unstable counterpart of the videos, by applying random affine transformations, with random scale, shear, translation, and rotation components, sampled from different uniform distributions. However, this approach is not able to generate realistic real-world unstable videos. We will further examine this dataset generation technique in the last chapter of this thesis. For more realistic videos, my dataset of choice was the one provided with [30]. This dataset contains 50 unstable videos, captured under a variety of conditions. (walking, running, static-scene, panning, low-light conditions).

I then proceeded in performing the necessary preprocessing steps, large motion reduction, Optical Flow extraction and inpainting. Essentially, I created a secondary dataset of inpainted optical flows.

Training Scheme

The proposed training scheme involves two stages. In the first stage the loss function consists of two terms, the previously introduced stability term $\boldsymbol{L_m}$ and spatial coherence term $\boldsymbol{L_f}$**:**

$$L_1 = L_m + 10 * L_f$$

The difference in the second stage is that we discard the $\boldsymbol{L_f}$ term:

$$L_2 = L_m$$

The first training stage lasts for 10000 iterations, whereas the second stage continues for another 5000 iterations. The authors argue that the introduction of L_f besides reducing high frequency noise, also helps L_m descent to a lower value, at the initial part of training. The optimizer of choice is ADAM [20] with $\beta_1 = 0.9, \beta_2 = 0.99$ with the learning rate set to 1e-4 for the first 2500 iterations and then reduced to 1e-5 for the rest of the training process. As they do not mention spatial dimensions for the input data, I resize all flows to 256 by 256. To augment the data, I perform random horizontal flips with a probability of 0.4 to all flows in a 20-frame segment.

Testing scheme

Since the network has a fixed length input of 20 frames, we must resort to a sliding window approach for inference. Also, since the optical flow in invalid regions is inpainted, our inferred warp fields must be inpainted in those regions as well.

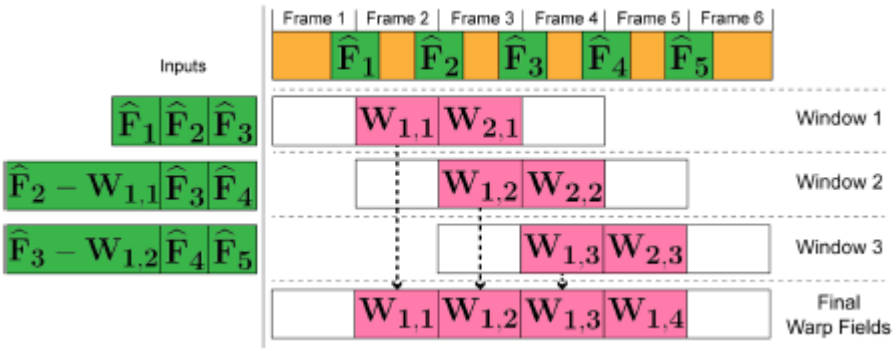


Figure 5.8: [23] Sliding window technique.

The sliding window approach operates as illustrated in Figure 5.8. Warp fields from different windows are denoted as $W_{n,k}$ with n, k representing the index within a window and window index respectively. The warp for the first frame is already known from the previous window. Another important detail is that the original flow $\widehat{F_k}$ must be updated to $\widehat{F_k} - W_{1,k-1}$ in order to match the new warped starting point. From each window, only the first warp field is inpainted and utilized to stabilize the video.

5.1.4 Discussion and Modifications

I implemented the above algorithm with some slight modifications in the large motion reduction and mask generation stages as previously mentioned. However, by following the proposed training scheme I was not able to replicate visually pleasing results. The generated videos suffered from heavy distortion artifacts and in some extreme cases they became even more unpleasant than the original unstable videos. Such a generated sequence is shown in Figure 5.9, where we can see a lot of artifacts and cropping.

Figure 5.9: Sequence of generated frames with large amounts of distortion and cropping

I will attempt to demonstrate the cause for this problem in Figure 5.10 by plotting a pixel profile generated from PCA inpainted flows against its warped-smoothed counterpart obtained by adding the networks inferred warp field.

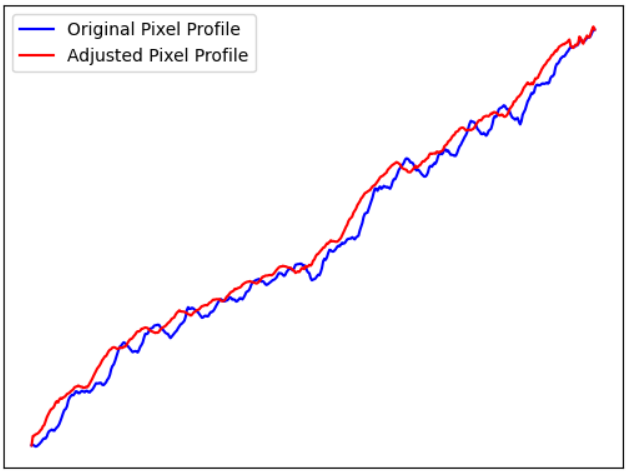


Figure 5.10: Original vs Warped Pixel Profile

We can see that the red line, which is the warped trajectory, starts deviating a lot from the original trajectory. At this point it resembles more of a straight line, caused by oversmoothing. However, when frames with high velocity are oversmoothed distortion and black borders are inevitable.

The network we trained is not at fault here, it has learned to minimize L_m, through generating a straight line. The reason the original training scheme is run only for a

small number of iterations is to prevent the model from learning to over smooth the original path. However, this is not a robust way to avoid this, especially in my case where I used a different training dataset. To overcome this issue and make the model learn a more balanced approach that preserves the patterns of the original camera motion, I will introduce L_d to the stability loss L_m:

$$L'_m = \lambda_1 * L_m + \lambda_2 * L_d$$

This term punishes any deviation from the original camera path. It forces an original pixel trajectory and its warped trajectory to stay as close as possible. The original pixel profile of pixel i can be obtained through the cumulative sum of its PCA inpainted flows $\widehat{F_{n,i}}$ by:

$$C_i = \sum_{n=1}^{N} \widehat{F_{n,i}}$$

Then its warped profile is defined as:

$$P_i = C_i + W_i$$

And finally, L_d can be written as:

$$L_d = \frac{1}{N} \sum_{n=1}^{N} \sum_{i=1}^{h*w} \|C_i - P_i\|_2$$

Notice we have not determined the balancing coefficients λ_1, λ_2 for the two terms L_m, L_d. This decision is nontrivial as it determines how smooth we want the video as opposed to how true we want to stay to its original trajectory. We could have the model determine these parameters, but it would very quickly resort to predicting $\lambda_1 = 0, \lambda_2 = 1$ which is the local minima where it essentially does not have to modify the incoming trajectories at all. Inspired by the optimization function used in MeshFlow [10], which we covered in Chapter 1.3.2 I will set the term λ_2 to fixed value of 1 and determine λ_1 dynamically using a translational indicator based on the mean velocity of the current 20-frame input. I firstly compute the average d_x, d_y values of the input flows denoted as $d_x^{\sim}, d_y^{\sim}$ and then the corresponding magnitude $T_u = \sqrt{d_x^{\sim 2} + d_y^{\sim 2}}$. Based on the linear model provided in MeshFlow, I set the λ_1 coefficient as:

$$\lambda_1 = max(-1.93 * T_u + 0.95, 0)$$

What this means is that when the input flows have high velocity then $\lambda_1$will have a lower value, so as not to oversmooth the path.

Modified Training Scheme

The training scheme I implemented still follows the basic principles of the original and consists of two stages. In the first stage the loss function is: $L_1 = L'_m + 10 * L_f$

This stage lasts for 3 epochs and then we proceed to the second stage where we omit the L_f loss:

$$L_2 = L'_m$$

Now the network is further finetune for another 2 epochs. The optimizer I used is again ADAM [20] with $\beta_1 = 0.9, \beta_2 = 0.99$. The learning rate is set to 1e-4 for the first epoch and then fixed to 1e-5.

Modified Testing Scheme

I experimented with slight modifications to the training scheme. Instead of utilizing only the second inferred warp field from each window, I use all fields except for the first frame which was previously adjusted, and the last frame which will be adjusted in the next window. To enforce further temporal consistency, I made consecutive windows overlap by a small number of frames. After experimenting with different values, I found $\text{overlap} = 2$ to produce the best results. The new testing schedule is shown in Figure 5.11, where I use 6-frame segments instead of 20 for visualization purposes.

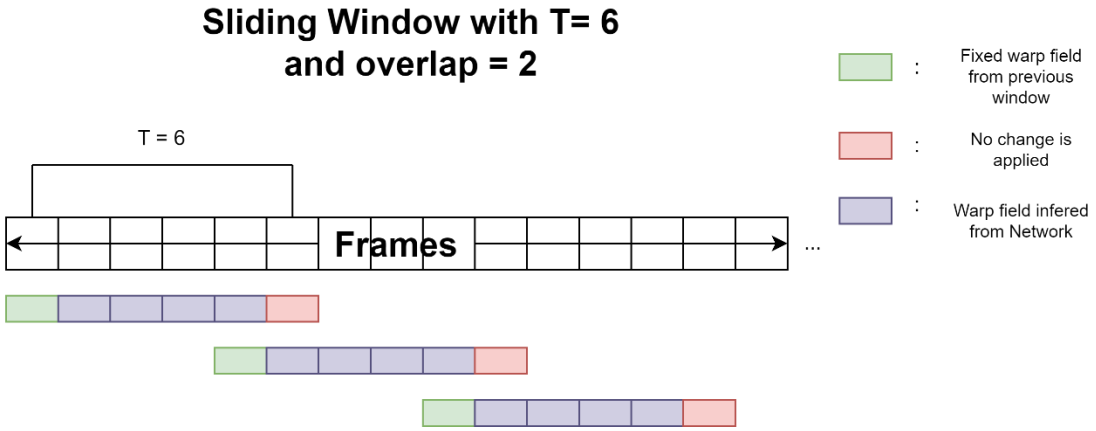


Figure 5.11: Sliding window schedule with 6-frame segments.

Further Temporal Smoothing

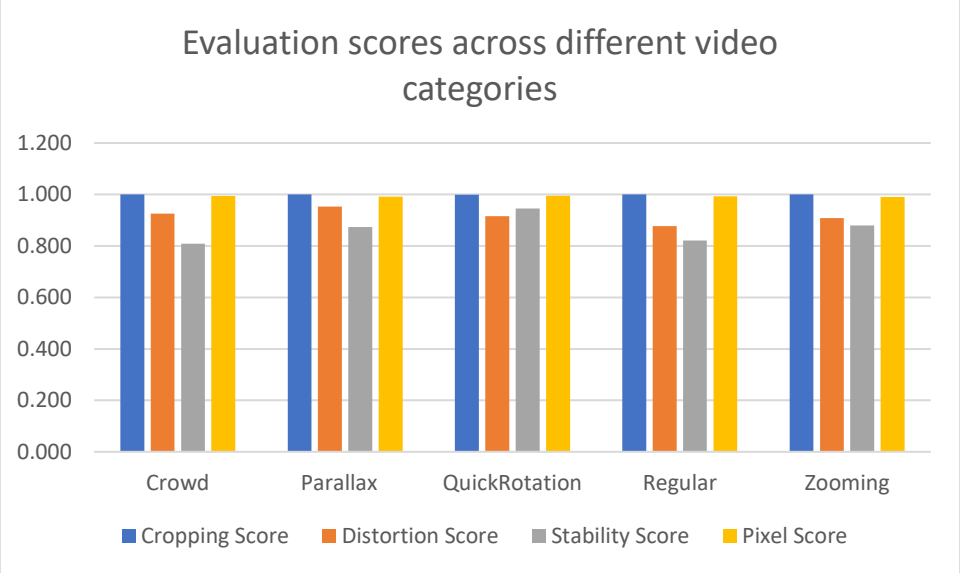
When all warp fields are estimated, we proceed to inpaint the inaccurate regions with PCAFlow, using the previously computed masks. After this stage I experimented with

introducing one more stabilization stage. A simple 1D moving average low-pass filter for all pixel profiles. I use a kernel of length 15 to mitigate any remaining high frequency shakes. To speed up this process, I utilize the GPU by applying the low-pass filter through PyTorch's [25] average pool 1D layer. I reshape the inferred warp fields from the shape [**frame count, 2, height, width**] to [**height, width, 2, frame count**] and then flatten it to [**height * width, 2, frame count**]. This way I essentially have a batch of **height * width** trajectories with 2 channels and a length equal to **frame count.** Now we can apply the moving average filter and reshape the result to appropriate dimensions.

5.1.5 Results and Discussion

Evaluation Scores

Category	Cropping Score	Distortion Score	Stability Score	Pixel Score
Crowd	1.000	0.925	0.809	0.995
Parallax	1.000	0.953	0.873	0.991
QuickRotation	0.999	0.916	0.945	0.996
Regular	1.000	0.878	0.821	0.993
Zooming	1.000	0.909	0.880	0.991



From the evaluation results we can see that the model performs reasonably well across all categories. There is no clear bias in any video category, due to the fact that the dataset we used for training contained various types of camera motion and scene layouts. However, I did not train another instance of the network on the proposed synthetic dataset to directly compare results.

As the algorithm requires optical flow estimation, its inference speed is a major drawback. It took approximately ten minutes to stabilize a 40 second video on my GTX 970 graphics card.

I also experimented with running the algorithm for multiple iterations, without any significant improvement in the stability score.

The trained model weights, inference code and supplementary materials can be found in this repository https://github.com/btxviny/Deep-Learning-Video-Stabilization-using-Optical-Flow.

5.2 Video Stabilization through Learning Motion Refinement and smoothing parameters.

In this section we will review DUT [26], that tackles the Video Stabilization problem with a divide and conquer strategy, implementing unsupervised training. The algorithm can be broken down into four stages as shown in Figure 5.12, Motion Initialization, Motion Refinement, Trajectory Smoothing and Frame Warping.

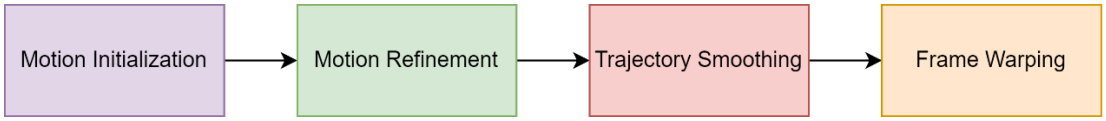


Figure 5.12: DUT stabilization algorithm.

Motion Initialization models motion using a sparse vertex grid. Using a novel multi-homography strategy we can mitigate the effects of dynamic objects. In the second stage the initialized motion gets further refined, using a deep encoder-decoder architecture to learn the residual motions for each vertex. For the trajectory smoothing stage, a novel 3D CNN is used to infer smoothing weights. As for the stable video synthesis, the authors use the mosaic technique, but I will also provide the PCAFlow warping technique [13] we mentioned in 5.1.1.

5.2.1 Motion Initialization

The purpose of this module is to suppress the effect of dynamic objects, through the use of a grid-based motion model. We divide the image into an $M \times N$ grid and each vertex $v_k, \forall k \in [1, M \times N]$ will get its motion value based on its nearby keypoints based on a multi-plane technique.

The input unstable video is defined as $\{f_i \,|\forall i \in [1, E]\}$ where E is the total number of frames. Firstly, the pretrained flow estimator network PWCNet [8] is utilized to extract optical flow between adjacent frames f_i, f_{i+1}. This dense flow is denoted as:

$$0F_n = PWC(f_i, f_{i+1})$$

However, we are only interested in the motion vectors of keypoints to alleviate the noise caused by dynamic objects and occlusion. For the keypoint detection process, we will use the network RFNet [57] to detect features with reliable motion flow.

$$\{p_{ij} = RFNet(f_i) \mid \forall i \in [1, E], \forall j \in [1, L]\}$$

Where p_{ij} is the jth keypoint detect in frame f_i, and L is the number of detected keypoints, which is set to 512 for the DUT algorithm. Now we only keep the flow of our detected keypoints:

$$m_{ij} = OF_i(p_{ij})$$

These motion vectors will be propagated to the vertex grid based on a multi-plane homography approach. Using K-means [27] we cluster the keypoint features into two planes based on their motion. This grouping can be expressed as:

$$C_i^c = \{ (p_{ij}, m_{ij}) | \forall j \in \Lambda_i^c \}$$

Where $c = 0/1$ denotes the cluster index and Λ_i^c is the keypoint index subset belonging to C_i^c. However, splitting the detected feature motions into two or more sets, only makes sense if their motions are significantly different and only if each subset has a substantial number of keypoints belonging to it. So, we assign more than one cluster only when the two following conditions are satisfied:

- When the cluster with the least features, has at least $20\% \times L = 102$ keypoints.
- When the mean motion magnitudes and angles of the two clusters differ more than 10 pixels and 10 degrees respectively.

Now we will also assign each vertex to a cluster, based on the majority cluster of its neighboring keypoints in a radius of 200 pixels. This majority in this local neighborhood is defined as $\Omega_{ik} = \{j \mid \|d_{ijk}\|_2 \leq R\}$

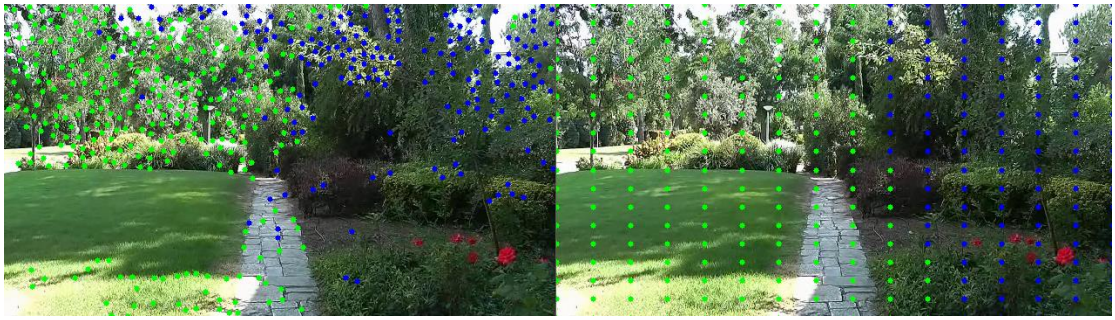


Figure 5.13: Multi-Plane Homography Motion Initialization

where $R = 200$. A visualization of this keypoint-vertex clustering scheme can be visualized in Figure 5.13. In the left picture we see the keypoints clustered in 2 planes and in the right one, the resulting vertex grid clustering.

Next based on the motion of the detected keypoints we fit two homography transformations for each plane/cluster:

$$H_i^c = Homo\left(\{p_{ij}^c | \forall j\}, \quad p_{ij}^c + m_{ij}^c | \forall j\}\right)$$

Using these transformations, we can initialize the motion for each vertex as $\widehat{n_{ik}} = H_i^c(v_{ik}) - v_{ik}$, where v_{ik} is the kth vertex of frame i.

5.2.2 Motion Refinement

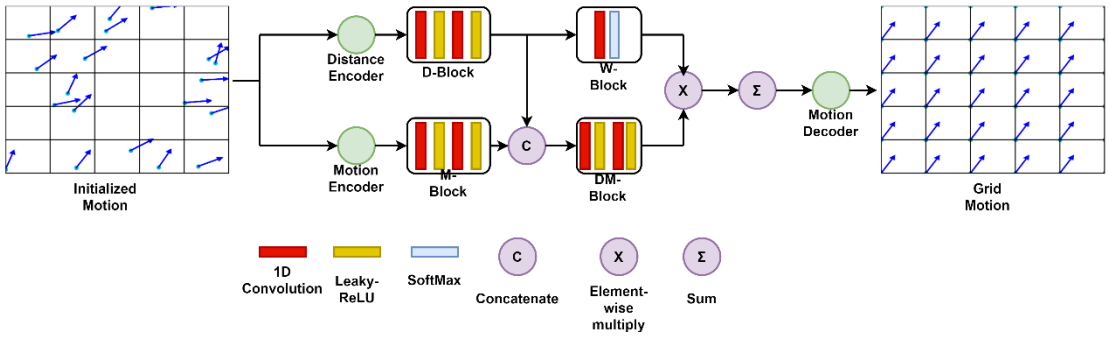


Figure 5.14: Architecture of the Motion Refinement Module

In this part of the algorithm the purpose is to further improve the accuracy of the estimated grid motion. The multiplane reconstructed motion estimate is accurate enough but there still is residual motion between the original motion estimate with PWCNet:

$$\Delta m_{ij} = p_{ij} + m_{ij} - H_i^c(p_{ij}), \forall j \in \Lambda_i^c$$

This residual motion, along with the distance vector between each grid vertex and keypoint $d_{ijk} = p_{ij} - v_{ik}, k = 1, \dots, MN$ will be the input for the Motion Refinement Network $MR(\cdot)$. This network uses 1D convolutions to associate residual keypoint motion to each grid vertex and infer the residual needed $\Delta n_{ik}$to refine the vertex motion:

$$\Delta n_{ik} = MR\left(\{(\Delta m_{ij}, d_{ijk})|\forall j \in [1, L]\}\right)$$

The residual motion and distance vectors follow different branches as shown in Figure 5.14, where they pass through separate encoders. The encoded vectors are then embedded by 1D convolutional layers. The distance embeddings are used to generate an attention vector through a Soft-Max activation, which is then multiplied with the concatenated tensor of the motion and distance embeddings. This generates the residual motion Δn_{ik} which is used to calculate the final vertex motion vector:

$$n_{ik} = \widehat{n_{ik}} + \Delta n_{ik}$$

Through the cumulative sum of this motion vectors across i we formulate the grid vertex trajectories.

The loss function used to train the Motion Refinement module is based on the following two principles:

1. The motion vector n_{ik} of a vertex v_{ik} should be close the motions of its neighboring keypoints inside Ω_{ik}.
2. The target position of each keypoints pij should be well approximated by its projection with the homography transformation introduced by the movement of the four corners of its enclosing grid.

With these considerations in mind the loss function is formulated as:

$$L_{vm} = \lambda_m \sum_{i=1}^{E-1} \sum_{K=1}^{MN} \sum_{j \in \Omega_{ik}} \|n_{ik} - m_{ij}\|_1 O_{ij} + \lambda_v \sum_{i=1}^{E-1} \sum_{j=1}^{L} \|p_{ij} + m_{ij} - H_{ij}(p_{ij})\|_2^2 O_{ij}$$

Where H_{ij} is the homography of the cell containing p_{ij}, and λ_m, λ_v

Are weights that balance the two terms. Finally, O_{ij} is an occlusion mask for each cell that mitigates the influence of dynamically moving objects. It's a binary mask with values 0 for pixels where the original flow OF_i and the backward grid -based reconstructed optical flow $\omega(\widehat{OF_i})$ differ more than a threshold. The implementation is defined in [58] and is calculated as:

$$O_i = \left(OF_i - \omega(\widehat{OF_i})\right)^2 < \alpha_1 \left((OF_i)^2 - \left(\omega(\widehat{OF_i})\right)^2\right) + \alpha_2$$

$$\text{Where } \alpha_1 = 0.01, \alpha_2 = 0.5$$

The authors also introduce a shape preserving term L_{sp} to guarantee further spatial coherence.

$$L_{sp} = \sum_{i=1}^{E-1} \sum_{m=1}^{(M-1)(N-1)} \|\widehat{v_{im}^3} - \left(\widehat{v_{im}^2} + R_{90}(\widehat{v_{im}^1} - \widehat{v_{im}^2})\right)\|_2^2$$

Where $\widehat{v_{im}^0} = v_{im}^0 + n_{im}^3$, o = 1,2,3,4, $v_{im}^1 \sim v_{im}^4$ represent the four vertices of a grid cell, moving in the clockwise direction. The corresponding motion vectors are $n_{im}^1 \sim n_{im}^4$ and R_{90} denotes the 90-degree counter-clockwise rotation.

The final loss function is:

$$L_{MR} = L_{vm} + L_{sp}$$

5.2.3 Trajectory Smoothing

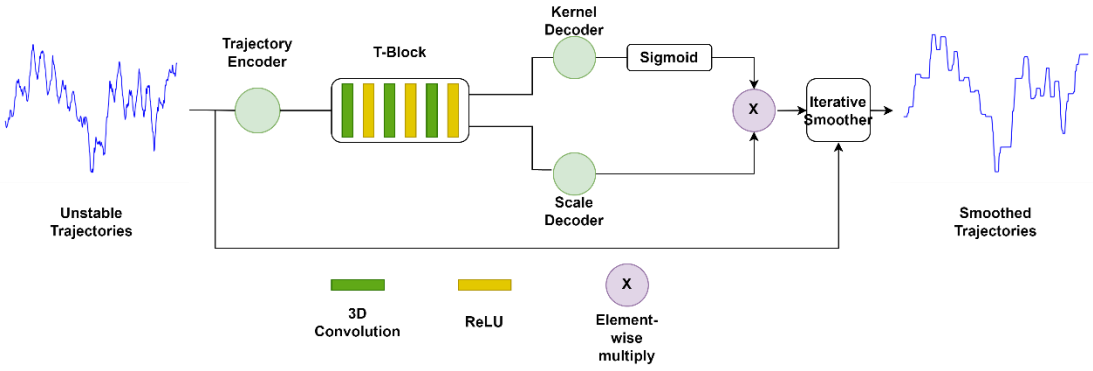


Figure 5.15: Trajectory Smoothing Module

Up to this point, we have estimated the refined grid motions n_{ik} for each vertex v_{ik}. Through the cumulative sum of this motion vectors across i we formulate the grid vertex trajectories T_k . At this stage we will use the Trajectory Smoothing module to infer dynamic weights to smooth the unstable trajectories. Before we delve into how the module works, we must first discuss the smoothing process.

The smooth trajectories should be predicted in such a way that the motions between adjacent temporal regions are small. This can be achieved by minimizing the following smoothness term:

$L_s = \sum_{j\in\Omega_i} w_{ij} \left(\left\| \widehat{T_{\imath k}} - \widehat{T_{Jk}} \right\|_2^2 \right)$ where Ω_i defines the local temporal region spanning from $-i\ to\ i$, $\widehat{T_{\imath k}}$ defines the smoothed trajectory of vertex v_k up to frame i and w_{ij} is the weight used to specify how strong the smoothing operation should be in the temporal region. Blindly enforcing the minimization of this term would result in producing straight line trajectories. This was the case in the previous implementation we discussed in 5.1. There we resolved this issue by introducing a term that punishes the deviation from the original trajectory, and a similar approach is taken here. The final objective function for the trajectory is written as:

$$L_{ts} = \sum_{i=1}^{E-1} \left(\left\| \widehat{T_{\imath k}} - T_{ik} \right\|_2^2 + \lambda L_s \right)$$

The introduction of the weighting parameters λ is based on the intuition that not all temporal regions should be equally smooth. In the previous implementation I crudely calculated this parameter based on the velocity of the frames. However, here the authors propose using the Trajectory Smoothing Module to determine this parameter. The network will also be used to predict the temporal step for Ω_i and the weights of the smoothing kernel w_{ij}.

The module is built upon a 3D CNN architecture. The inputs are the unstable grid trajectories which are embedded by a 1D convolutional encoder. These embeddings are further processed by the 3D convolutional T-Block as shown in Figure 5.15 producing an output feature map of shape $[64, E, M, N]$. This latent representation of the trajectory goes through two separate decoders, the kernel decoder which infers a feature of size $[12, E, M, N]$ and the scale decoder which produces an output of size $[1, E, M, N]$. From the first feature map 10 of the twelve channels are used as the kernel weights w_{ij}, one is used for the balancing term λ_{ik} for each location k and temporal step i and the last one for determining the radius of the temporal region. The output of the scale decoder is used as the balancing coefficient λ.

With the intrinsic parameters of the objective function inferred by the module the smooth trajectories are finally obtained, by the following iterative solver:

$$\widehat{T_{ik}}^{t} = \frac{T_{ik} + \lambda_{ik} \sum_{j \in \Omega_i} w_{kij} \widehat{T_{jk}}^{t-1}}{1 + \lambda \sum_{j \in \Omega_i} w_{kij}}$$

Lastly, when the smooth trajectories $\widehat{T_{ik}}$ have been estimated, we can obtain the warping operations to stabilize the video as follows:

$$W_{ik} = \widehat{T_{ik}} - T_{ik}$$

5.2.4 Training and Implementation Details

The dataset used for training the Network were the 60 unstable videos from DeepStab [31]. However, there is no mention of the preprocessing techniques applied to the data. Apart from that, there is no mention of the settings used to train the modules, number of epochs, optimizer, learning rate etc. Nevertheless, the authors have made their inference scheme available online which yields satisfactory results. One key parameter for the algorithm which is not mentioned in the paper is that the images are divided into 18 by 32 during the motion estimation stage, which aims to maintain the popular landscape aspect ratio of $16:9$, present in most consumer captured videos. Another important parameter is the number of iterations used to acquire the final smoothed trajectories, which is set to 20 in the code implementation.

Despite the availability of the inference scheme online, a major limitation is that it requires all data to be loaded on GPU memory at once. This prohibits its use by users affected by GPU limitations. For this reason, I implemented an alternative script that performs the motion estimation stage sequentially rather than in parallel.

5.2.5 PCA-Flow based Image Warping

Another contribution I provide is an alternative image warping algorithm, that is used to produce dense flow fields from sparse grid flow. This approach is similar to the one reviewed in 5.1.1. It is based on solving a least squares problem to determine the weights of the first five principal components of optical flow, that best explain the motions of the sparse grid vertices. The final warp field is the linear combination of the principal components using the weights we estimated. Firstly, we have to create a sparse flow matrix from the vertices' motions, and then provide a binary mask which is set to 1 only at the coordinates of the grid vertices.

The motivation for this alternative warping method is that the Mosaic technique, used in the original implementation, frequently introduces distortion artifacts close to the image boundaries, where the warped positions of neighboring cells overlap. An example of this phenomenon is shown in Figure 5.16.

Figure 5.16: Comparison of the Mosaic warping method (on the left) versus the proposed PCAFlow method (on the right).

On the other hand, PCA inferred warped fields are guaranteed to be spatially coherent. This, however, sometimes comes at the cost of stability. A more in-depth comparison will be offered in the next section, where we provide the results of the DUT algorithm across a variety of video scenes using both warping methods.

5.2.6 Results and Discussion

The evaluation results using both warping methods are shown in Figure 5.17

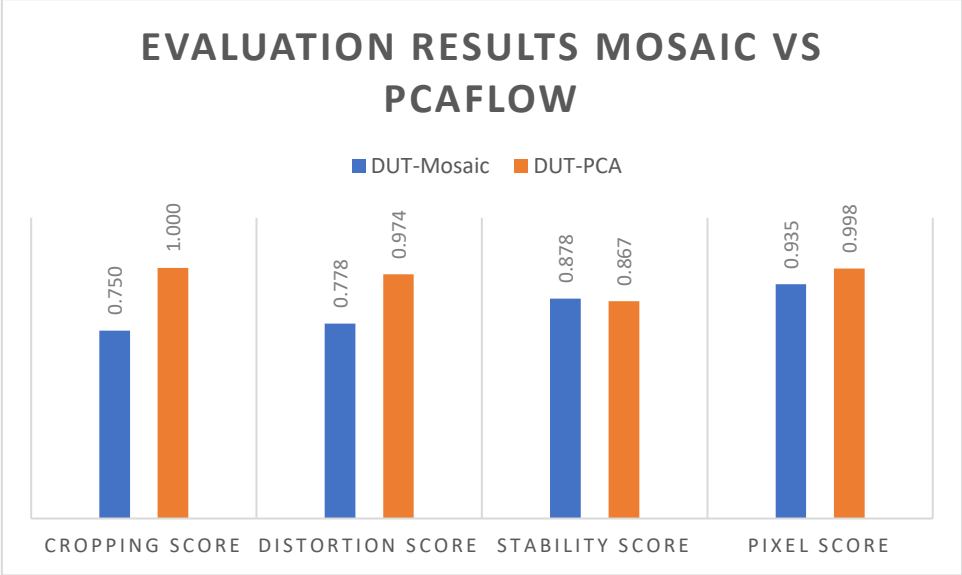


Figure 5.17: Evaluation scores of DUT using different warping methods.

We can see that the proposed warping method does improve the cropping and distortion scores, a fact that comes at the cost of stability. Since stabilization is our core objective the mosaic method is preferable.

Similarly, to the algorithm we discussed in 5.1, this method also suffers from long inference times. This is inevitable since it relies on optical flow estimation of the whole input video before the motion is stabilized.

The modified inference code with the RAFT model and my alternative warping method can be found in this repository https://github.com/btxviny/DUT-Video-Stabilization.

5.3 Optimization in CNN weight space

Addressing video stabilization by solving for pixel-wise warp fields to counteract unwanted camera motions, poses challenges. As video length increases, traditional gradient-based optimization algorithms encounter difficulties, with a 100-frame 480p video involving 854 x 480 x 100 motion vectors, making the problem computationally demanding.

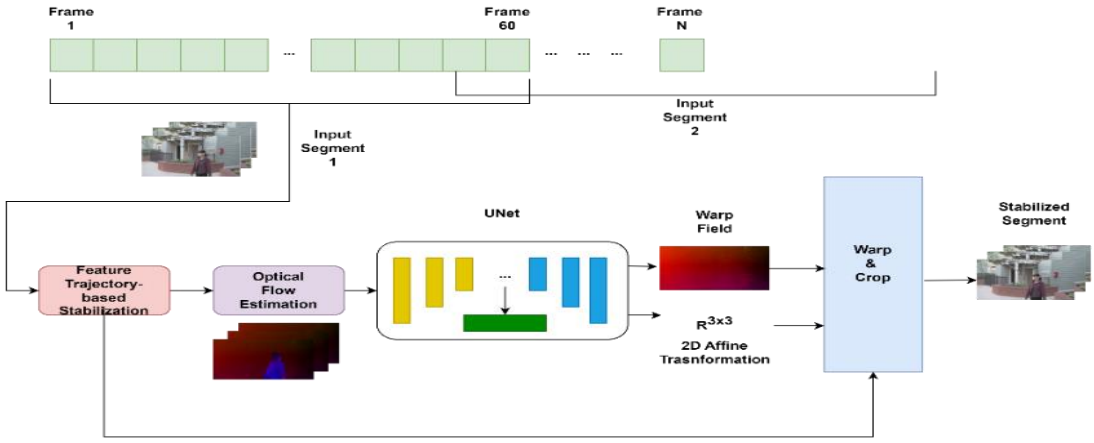


Figure 5.17: Overview of the proposed algorithm

We will review [28] which introduces the concept of test-time training, focusing on overfitting a model to a specific unstable video input. In this approach they utilize a CNN purely as an optimizer. This strategy effectively reduces the dimensionality of the problem by transferring it into a neural network weight space instead of optimizing individual motion vectors. The goal is to avoid the limitations of traditional methods that optimize individual motion vectors, which can get stuck in local minima. The is no traditional training taking place. Instead, the model is overfitted on different segments of the camera path, specifically minimizing the objective function for its input trajectory segment. The model does not aim for generalization but rather focuses on providing high-quality stabilization results for its trained trajectory segment. Per-pixel warps for the entire video are obtained using a sliding window approach, yielding the final result.

The overall algorithm is shown in Figure 5.17. The authors tested various parameters for the input segment length, the model architecture, and the number of optimization iterations. The following details are the ones that yielded the best results in the paper. The size of the input segment is set to $T = 60$ frames, with spatial dimensions 360×640. The first step is to remove large abrupt motions, with a feature-trajectory stabilization algorithm, same as the one described in 5.1.1. Next, the authors propose using the algorithm described in [59], to extract the original bi-directional optical flow. Using these flows the proposed CNN is overfitted to the 60-frame segment for 150 iterations using ADAM [20] with a learning rate of 1e-5. The results are 60 dense warp fields alongside 60 affine transformations, used for warping the pre-stabilized frames.

The above algorithm proceeds by using a sliding window technique. The next input window moves over by 58 frames. This ensures a two frame overlap that aims to achieve further temporal consistency. An important detail is that the first and last warp field are fixed, determined from the previous and next window respectively. The stability loss used is similar to the one in 5.1. but its= also implements warping with an inferred affine transformation for both the forward and backward pixel trajectories. It is balanced out by a regularization term that enforces shape preservation and also penalizes regions with large motion magnitudes.

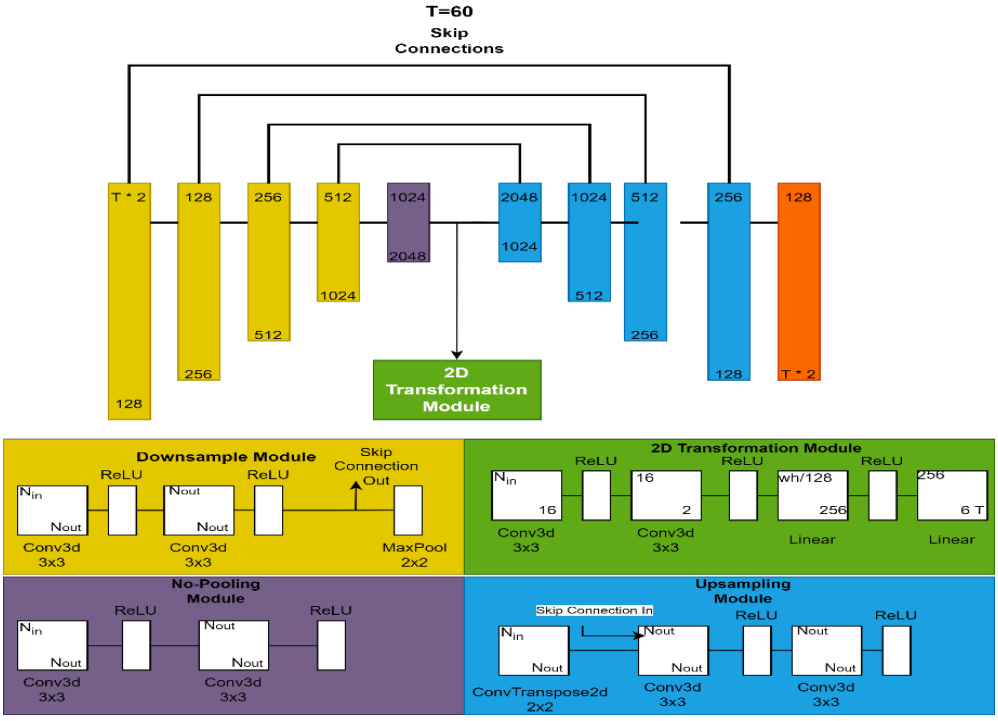


Figure 5.18: Architecture of Proposed Network

The proposed model architecture is a UNet with four encoder-decoder stages. The feature map generated by the last encoder stage is fed into a Spatial Transformer

(STN) module [36] that infers the affine transformations to warp the frames. The full network details are shown in Figure 5.18.

This method suffers from a major limitation, its computation speed. According to the original paper 30 minutes are required to stabilize a 40 second video, on a GTX1080Ti GPU. This makes it prohibitive for real time applications as it requires feature stabilization, optical flow estimation and 150 CNN updates for each frame segment.

The authors propose a way to speed up the algorithm by first training the network on a large dataset, and then finetuning it for each specific input frame segment. Despite its obvious limitations, it was included in this thesis as it demonstrates an alternative use of neural networks, in optimizing large scale problems.

Chapter Summary

In this chapter we discussed learning-based approaches, that directly build upon the traditional optimization techniques we saw in Chapter 1. Using deep learning we saw how the motion estimation and motion refinement step can be improved while guaranteeing robustness. These methods were trained in an unsupervised manner, using rather complicated objective functions. Also, they require large amounts of time to produce the final result. In the next chapters, we will delve into more conventional learning approaches using labeled data. We will simplify the problem of digital video stabilization and achieve greater inference speeds.

Chapter 6: Supervised Video Stabilization

Introduction

Contrary to the learning-based path optimization implementations we previously discussed, in this chapter we will delve into more classical deep learning approaches that rely on supervised Datasets. These datasets typically consist of stable/unstable video pairs. This kind of training data allows for the use of simpler loss functions, where the output of the network can be directly compared to a ground truth frame or transformation. I will review and implement two algorithms that take advantage of supervised learning to tackle the problem of video stabilization.

6.1 Deep Video Stabilization with Transformation Learning

In [31] alongside DeepStab StabNet is introduced, a deep neural network that builds upon the idea of Spatial Transformers and is trained in a Siamese architecture.

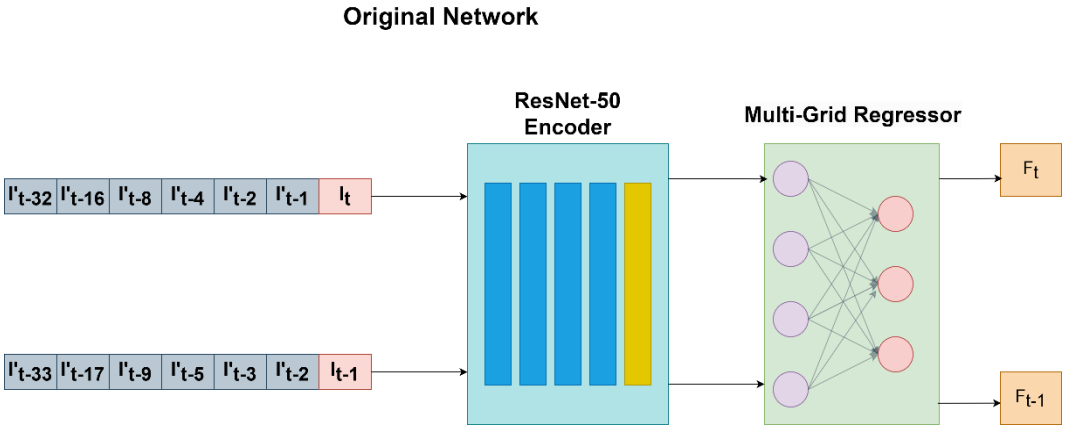


Figure 6.1: Network architecture of the original Implementation that uses ResNet-50 as a backbone encoder.

The network is comprised of a backbone feature extractor/ encoder module and a transformation regressor. Given an input frame sequence, the encoder produces embeddings that are in turn fed into the regressor that learns the necessary transformation for stabilizing the last frame of the sequence. The feature extractor is a ResNet-50 network trained on [60] for video classification, where we exclude the top fully connected layer.

6.1.1 Network Input

The inputs to StabNet are an incoming unsteady frame I_t and six conditional historical steady frames sampled from the last second S_t. They authors opted not to sample the six frames uniformly over one second, rather more densely near the incoming frame i.e. $\boldsymbol{S_t} = \{\boldsymbol{\dot{I}_{t-32}}, \boldsymbol{\dot{I}_{t-16}}, \boldsymbol{\dot{I}_{t-8}}, \boldsymbol{\dot{I}_{t-4}}, \boldsymbol{\dot{I}_{t-2}}, \boldsymbol{\dot{I}_{t-1}}\}$. All the conditional steady frames are in grayscale format instead of RGB. This choice is made to achieve greater inference speed. The model's intended use is for online applications. Online methods operate only with previous frames as inputs and should be able to achieve real-time speed.

6.1.2 Network Architecture

The choice of the pretrained encoder module is very important as it will be used to embed our incoming frame sequences. In the original paper the authors integrated a pretrained ResNet-50 network that is finetuned during training. All pretrained weights are imported, except for the first convolutional layer. The reason for this is that ResNet was trained on three-channel images whereas our input has 9 channels, 6 for the grayscale historical frames and 3 for the incoming RGB unsteady frame. We must now decide how to initialize this layer's weights, as the authors don't provide specific details. To randomly initialize the new layer's weights, would equal to not utilizing the potential of transfer learning to its maximum. As a solution to this problem, I propose the following approach:

1. Extract original pretrained RGB weights of shape [64, 3, 3, 3]
2. Compute the mean over the second dimension resulting in a shape of [64, 1, 3, 3], which we will interpret as pseudo-grayscale weights. Then tile these weights six times for our six conditional frames with a shape of [64, 6, 3, 3].
3. The final conv_1 parameters are obtained by concatenating the original RGB with the pseudo-grayscale weights, giving a final shape of [64, 9, 3, 3], which is appropriate for our input.

In the original paper it is mentioned that the weights of the backbone encoder are not frozen but left trainable for fine-tuning. However, the number of layers which will be finetuned is not specified. The lower layers of pretrained successful CNNs such as ResNet-50, have learned to extract low-level features like edges and basic shapes. and basic shapes. By keeping these layers fixed during training we can leverage the knowledge the model has gained from training on a big dataset such as ImageNet. So, I opted to only leave the top five convolutional blocks to be trained specifically for our application.

Finally, the last choice we must make regarding the model architecture is the transformation regressor. For this task we a four-layer multi-layer perception (MLP) module. Convolutional layers can handle inputs of variable spatial dimensions; however, this is not the case with linear layers. The input to the regressor needs to be

constant, independent of the input dimensions. For this reason, the output of the feature extractor is averaged over the height and width dimensions and has a constant shape of [1,2048,1,1]. We could train the model to learn an affine or a homography transformation, however in cases with large scene depth variation it wouldn't be able to perform well. For this reason, a sparse vertex grid displacement $\boldsymbol{F_t}$ is learned that is then propagated to all pixels using interpolation. Three different grid configurations were tested, $4 \times 4,\ 8 \times 8,\ 16 \times 16$, with the later providing the best visual results. An overview of this network is shown in Figure 6.1.

6.1.3 Training and Loss functions

As we mentioned, the model is trained in a Siamese architecture. This essentially means that two instances/ branches of the model are instantiated, each receiving its own input. The upper branch receives the input conditional sequence $S_t = \{\dot{I}_{t-32}, \dot{I}_{t-16}, \dot{I}_{t-8}, \dot{I}_{t-4}, \dot{I}_{t-2}, \dot{I}_{t-1}\}$ concatenated with I_t while the lower branch receives the corresponding sequence for the previous frame $S_{t-1} = \{\dot{I}_{t-33}, \dot{I}_{t-17}, \dot{I}_{t-9}, \dot{I}_{t-5}, \dot{I}_{t-3}, \dot{I}_{t-2}\}$ concatenated with I_{t-1}. This is done, so that we can force the model to be temporally consistent through a temporal loss that will be introduced below.

The training is driven by three types of different loss functions: 1) Stability loss. 2) Shape-preserving loss. 3) Temporal smoothness loss. The final loss function is based on consecutive frames I_t , I_{t-1} and is defined as:

$$L = \sum_{i \in \{t, t-1\}} (L_{stab}(F_i, I_i) + L_{shape}(F_i, G_i)) + L_{temp}(F_t, F_{t-1}, I_t, I_{t-1})$$

Where $F_t = W * I_t$ is the warped frame, with $*$ being the warping operator. The different components of the loss are defined as follows:

1) Stability Loss: This loss encourages the warped frames to stay close to the ground truth steady frames by encouraging pixel alignment through MSE and matched feature alignment.

$$L_{stab}(F_t, I_t) = \alpha_1 L_{pixel} + \alpha_2 L_{feature}$$

The pixel alignment term is:

$L_{pixel}(F_t, I_t) = \frac{1}{D}\|I_t - F_t\|_2^2$, where D is the spatial dimension of the images.

Whereas the feature alignment loss is defined as:

$$L_{feature}(F_t, I_t) = \frac{1}{m}\sum_{i=1}^{m}\left\|p_t^{si} - W * p_t^{ui}\right\|_2^2$$

Here m is the number of matched features between each steady/unsteady frame, with p^s being the stable features coordinates and p^u being the unstable features coordinates.

To compute the matched feature pairs during the preprocessing stage, keypoints are detected using SIFT features [3] instead of SURF features [2] in the original paper, as the later algorithm is still under patent. This process takes place at a 2x2 sub-image level to guarantee uniform feature detection. The

outliers are rejected using the RANSAC algorithm [61] to fit a homography in each corresponding sub-image pair. This loss can be interpreted as the mean squared eyclidean distance of the stable keypoint positions and the warped positions of the unstable features. The coefficients α_1, α_2 aim to balance the contribution of the two terms, with default values $\alpha_1 = 50.0$ and $\alpha_2 = 1$ in the original paper. However, since I scale the feature positions between [-1, 1] I set $\alpha_1 = 10$ and the two terms have the same order of magnitude.

2) Shape-Preserving Loss: To maintain the accuracy of our model, which predicts the positions of mesh vertices in stabilized videos, it's crucial to ensure that grid shapes are preserved, minimizing distortion artifacts, and promoting consistent transformations among neighboring grid cells. In other words, we want the resulting mesh grid to stay close to the original. This is the same shape preserving term as the one used in [12].
3) Temporal Smoothness Loss: Applying the predicted transformations to each frame individually can introduce wobble artifacts. Therefore, to guarantee consistency, we take advantage of the Siamese architecture, by feeding samples of two successive timestamps to the network.

$$L_{temp}(F_t, F_{t-1}) = \lambda \frac{1}{\mathrm{D}} \left\| Ft - W_{original} * F_{t-1} \right\|_2^2$$

Here $W_{original}$ is the optical flow from stable frame I_{t-1} to I_t which is computed during the preprocessing stage. In the original paper the authors used the [7] algorithm for this purpose. However, this is a very time-consuming process when run for 60 videos, so I opted to use the pretrained optical flow estimator network [8], utilizing the inference speeds of Deep Learning. This network has shown great results in optical flow estimation and is used throughout this thesis. However, its compatibility is limited since it requires a custom correlation CUDA [62] layer. Another great alternative is RAFT [63], which is available through the torchvision [64] library and requires no custom layers. Here $\lambda = 10$ is a constant coefficient to balance the contribution of this loss.

6.1.4 Implementation Details

All samples are resized to a spatial dimension of $W = H = 256$ for efficiency. The pretrained ResNet-50 model is loaded, and its top five layers are set to be trainable. Since the feature encoder was trained on the ImageNet dataset, we must standardize our data according to its per-channel mean = [0.485, 0.456, 0.406] and standard deviation = [0.229, 0.224, 0.225], in order to make full use of the pretrained weights. The batch size is set to 4 samples and the choice of optimizer is ADAM [20], with the learning rate set to 2e-5. The resulting dataset has a total of approximately 30.000 samples. The learning rate is decayed by a factor of 0.1 after every epoch and the whole training process was run for 5 epochs.

Figure 6.2: On the upper left-hand side is the resulting warped frame during epoch 1, and on the right side is the ground truth steady frame. The matched features are drawn in both images with small circles, green for the unsteady and red for the steady frame. In the lower part of the figure, we see the results for a random sample in epoch 4 where the resulting frame's quality has drastically improved.

6.1.5 Modifications

In recent years VGG-16, VGG-19 [32] have shown great results in feature extraction, and are widely used in perceptual losses, neural-style transfer, and many other applications. For this reason, I trained another instance of StabNet using VGG-19 as a backbone encoder instead of ResNet, which ultimately led to faster convergence.

Another modification I tested was altering the network's conditional input. I opted for the input $\boldsymbol{S_t = I_{t-16}, I_{t-8}, I_t, I_{t+8}, I_{t+16}}$, which now consists of only unstable frames, but future frames all also provided. In this manner the algorithm can no longer be categorized as online, but real time applications can be achieved by using a one second frame buffer. Also, by reducing the size of the input we can now provide the images in RGB format while maintaining inference speed and utilizing the pretrained weights of the encoder better. To achieve this, I extract the RGB weights of the conv1 layer of VGG and tile them five times to make them appropriate for the input size. The proposed modified StabNet is shown in Figure 6.3.

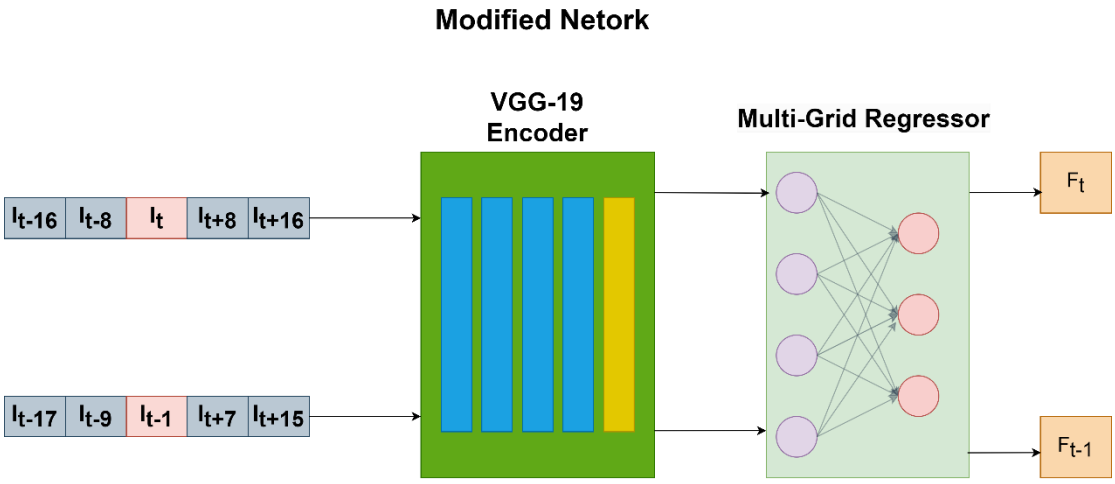


Figure 6.3: Architecture of Modified Network

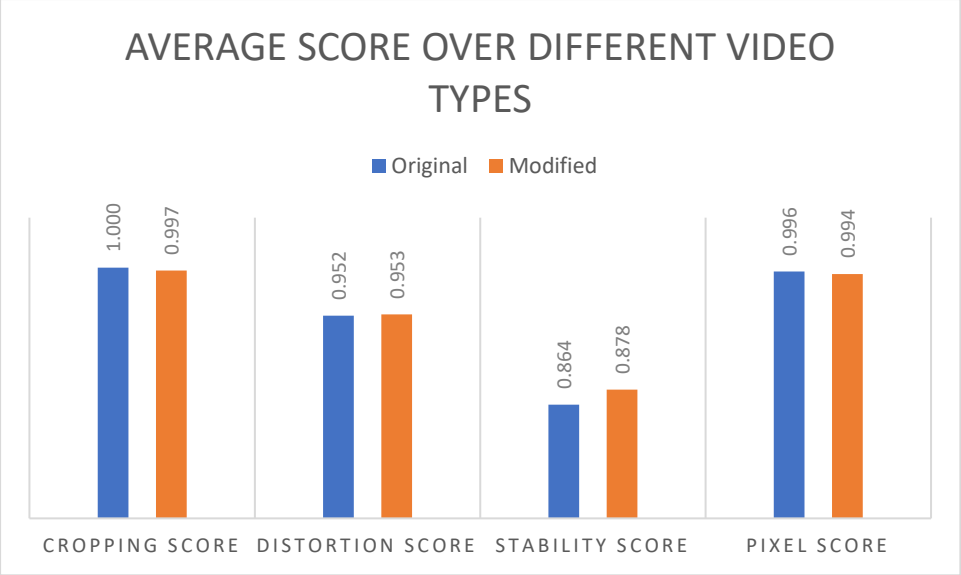
6.1.6 Results and Discussion

Original Network Evaluation Results

Category	Cropping Score	Distortion Score	Stability Score	Pixel Score
Crowd	1.000	0.972	0.825	0.997
Parallax	1.000	0.974	0.883	0.996
Quick Rotation	1.000	0.956	0.939	0.998
Regular	1.000	0.907	0.791	0.997
Zooming	1.000	0.952	0.879	0.994

Modified Network Evaluation Results

Category	Cropping Score	Distortion Score	Stability Score	Pixel Score
Crowd	0.996595	0.957544	0.843406	0.995041
Parallax	0.997381	0.963442	0.879606	0.99214
Quick Rotation	0.995954	0.93755	0.939005	0.996597
Regular	0.997064	0.952734	0.814041	0.994217
Zooming	0.998553	0.956101	0.915677	0.990562



From the evaluation results the modified network slightly outperforms the proposed one in terms of the stability metric. Another improvement is the inference speed. For 360×640 videos with my GTX 970 graphics card the original network had a speed of 0.123 seconds per frame, while the modified one achieved 0.107.

Although this method provides promising results for video stabilization, it's potential is limited due to the dataset's inherent flaw, the perspective mismatch between steady and unsteady frames. An example of this is that while computing matched keypoint features, the threshold of 64 minimum matched features, was not met quite often. This was due to the camera pose and content variation between the steady and unsteady cameras. A general rule in Deep Learning is that a model is only as good as its training data.

My implementation of the aforementioned algorithm, training code, model weights and supplementary material can be found in this repository https://github.com/btxviny/StabNet. While StabNet introduced the potential of supervised learning in the field of video stabilization, it still required time consuming preprocessing steps and was limited to transformation learning through carefully crafted loss functions. In the rest of this chapter, we will present a purely generative implementation that is able to directly predict stable frames given an input frame sequence.

6.2 Original (Vanilla) GANs

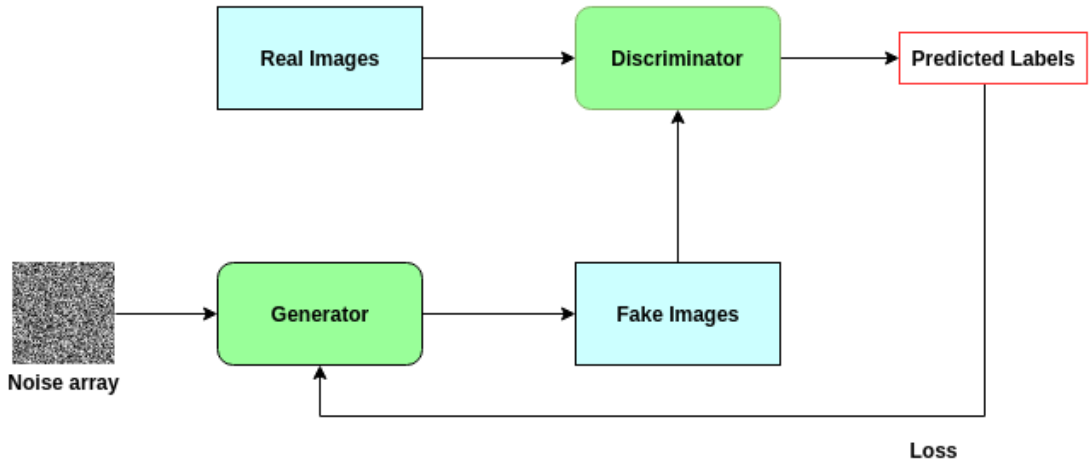


Figure 6.4: Illustration of a classical GAN training scheme

Generative Adversarial Networks (GANs) [65], are a class of machine learning models used for generating data, often in the form of images or other complex data types. GANs consist of two primary components: the generator and the discriminator. They are designed to work together in a minimax game, where the generator tries to create data that is indistinguishable from real data, while the discriminator aims to differentiate between real and generated data.

In the original paper, the generator is a neural network that takes random noise as input and generates data, such as images. It tries to produce data that is similar to real data. On the other hand, the discriminator is a neural network that acts as a binary classifier. It takes both real data (e.g., actual images) and generated data (output from the generator) as input and tries to distinguish between them. Its goal is to correctly classify real data as real and generated data as fake.

The loss function for the discriminator is binary cross-entropy, where the expected output for real data is 1 and 0 fake. The aim of the generator is to fool the discriminator and so it makes sense that its loss function is the binary cross-entropy of the discriminator's output for the generated images and a label of 1. The training is terminated when the losses of both models reach an equilibrium.

In other words, the two models play the following minimax game:

$$\min_G \max_D V(D,G) = E_{x\sim p_{\text{dt}}(x)}[\log D(x)] + E_{z\sim p_z(z)}\left[\log\left(1 - D\big(G(z)\big)\right)\right]$$

Where:

- G represents the generator network.
- D represents the discriminator network.
- x represents real data sampled from the true data distribution $p_{data}(x)$.
- z represents random noise from a prior distribution $p_z(z)$.
- $D(x)$ is the discriminator's output for real data.
- $D\big(G(z)\big)$ is the discriminator's output for fake data.

The minimax objective function in a GAN can be related to minimizing the Jensen-Shannon divergence (JSD) of the real and fake data distributions. The JSD is a symmetric variant of the Kullback-Leibler (KL) divergence [66], which measures the difference between two probability distributions.

A very popular variation of GANs, are Conditional GANs or cGANs [67], where the generator no longer samples noise from a distribution but takes in a conditional input. In the context of video stabilization this input could be a sequence of frames. The great advantage of utilizing cGANs for our task, in that we no longer need to crop black borders in the stabilized frames, as we are no longer applying any warping transformation. Apart from that, they allow for the use of simpler loss functions. However, before presenting Generative Video Stabilization Models, we must first mention some disadvantages that come along adversarial training. These issues include:

1. Hyperparameter Tuning: GANs involve multiple hyperparameters that need to be carefully tuned, including learning rates, batch sizes, and architectural choices. Finding the right combination can be a time-consuming process.
2. Model Architecture Symmetry: GANs have a symmetric architecture with the generator and discriminator, which can make training challenging. Achieving a balance between these two components is crucial for successful convergence.
3. Mode Collapse: Mode collapse occurs when the generator produces limited or repetitive samples, failing to capture the full diversity of the data distribution. This can result in a lack of variety in generated content.
4. Training Monitoring: Monitoring GAN training can be complex. There's no clear loss function that directly indicates the quality of generated data, making it necessary to rely on heuristics and visual inspection.
5. Vanishing Gradients: The use of cross-entropy can lead to vanishing gradients for samples that are correctly classified but are significantly different from the real data. This means that the gradient information needed to update the generator's parameters effectively becomes very small, causing slow or stalled training.

6.2.1 Improved Adversarial Training

We will discuss two variations on the classical GAN, that aim to counter the difficulties mentioned previously.

Least Squares GANs: In LSGANS [68] the authors argue that using the sigmoid cross entropy function for the discriminator D, can introduce vanishing gradients when updating the generator G with fake samples that are correctly classified as real but are still significantly different from genuine data. As they lie on the correct side of the cross-entropy decision boundary, they will introduce almost no error. They propose removing the sigmoid activation at the last layer of the discriminator, leaving the model free to generate linear values for each input. They combine this with a least squares loss function that is able to move fake samples towards the decision boundary. This means that fake data which are classified as real but still differ from the real data distribution will also be penalized heavily.

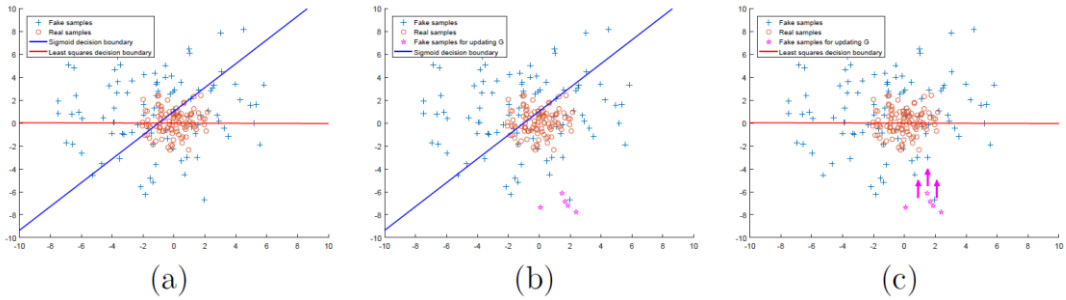


Figure 6.5: Illustration of the contrasting behaviors of two loss functions. (a) Depicts the decision boundaries of the two loss functions. It's crucial for the decision boundary to intersect with the real data distribution for successful GANs learning. If this isn't the case, the learning process can become saturated. (b) Shown is the decision boundary associated with the sigmoid cross-entropy loss function. It yields very small errors for fake samples (depicted in magenta) when updating the generator because these samples fall on the correct side of the decision boundary. (c) Depicts the decision boundary of the least squares loss function. It penalizes fake samples (in magenta),

compelling the generator to generate samples closer to the decision boundary, thereby fostering a more balanced and effective learning process.

This new loss function for the discriminator and generator respectively can be define as:

$$\min_D V_{LSGAN}(D) = \frac{1}{2} E_{x \sim p_{\mathrm{dt}}(x)}[(D(x) - b)^2] + \frac{1}{2} E_{z \sim p_z(z)} \left[\left(D\big(G(z)\big) - a \right)^2 \right]$$

$$\min_G V_{LSGAN}(G) = \frac{1}{2} E_{z \sim p_z(z)} \left[\left(D\big(G(z)\big) - c \right)^2 \right]$$

Where a, b are the labels are the labels for fake and real data and c denotes the value that G wants D to believe for fake data, which is set equal to b. While the classical GAN minimax game could be related to minimizing the Jensen-Shannon divergence of the real and fake data distributions, here the authors relate LSGANs to f-divergence.

Training a generator using this scheme offers greater training stability by alleviating the issue of vanishing gradients. Also, the value of the loss directly relates to the quality of the generated images, thus allowing for better training monitoring. Despite these advantages over vanilla GANs, we still must be very cautious when choosing the model architecture and tuning hyperparameters.

Wasserstein GANs: WGANs [69] introduce a novel approach to training GANs. One of the significant departures from traditional GANs is the replacement of the discriminator with a 'critic.' This change reflects a shift in perspective from a binary classification task to a more nuanced evaluation of the generated data. Similar to LSGANs, the output of the critic is not passed through a sigmoid activation function, leaving the model free to generate linear values for each input. The loss function in WGANs, which minimizes the Earth Mover's Distance (also known as the Wasserstein distance), quantifies how much 'work' is required to transform the generated distribution into the real data distribution. This metric ensures a direct and informative measure of the quality of generated images. The lower the loss, the closer the generated images align with real data, providing a clear and intuitive metric for training monitoring.

To ensure the successful training of WGANs, several key considerations come into play. The Earth Mover's Distance calculation in WGANs depends on the Lipschitz constraint, a mathematical property enforced on the critic network. This constraint

ensures that the critic's output doesn't change too rapidly with respect to the input data. Specifically, it enforces a 'Lipschitz continuous' constraint, which means that the gradient of the critic network must not exceed a certain value (typically denoted as 'K'). This constraint is necessary to make the Wasserstein distance well-defined and computable.
The authors impose this constraint on the critic through gradient clipping. Gradient clipping involves setting a threshold value ('c') for the magnitude of gradients during backpropagation. If, during training, the gradient of the critic's output with respect to its input exceeds this threshold ('c'), the gradient is scaled down to 'c.' This scaling ensures that the critic's gradient remains within a certain bound, which is essential for adhering to the Lipschitz constraint.

As for the loss functions used in Wasserstein Generative Adversarial Networks, we want to encourage the critic to predict large scores for real data and low scores for fake data, while the generator aims to have large critic scores for its generated outputs. In other words, the critic's objective is to maximize the Wasserstein distance of the two data distributions, while the generator's goal is to maximize the scores for its outputs.

Critic Loss Function:

$$\mathcal{L}_{\text{critic}} = -\left(E_{x\sim p_{\text{r}}}[\text{Critic}(x)] - E_{z\sim p_z}\left[\text{Critic}\left(G(z)\right)\right]\right)$$

Where:

- $\text{Critic}(x)$ denotes the critic's score for real data.
- $\text{Critic}\left(G(z)\right)$ denotes the critic's score for fake data generated by the generator.

Generator Loss function:

$$\mathcal{L}_{generator} = -\text{E}_{\text{z}\sim \text{p}_\text{z}}\left[\text{Critic}\left(\text{G}(\text{z})\right)\right]$$

The reason behind the minus in front of both loss functions is that standard optimization algorithms such as RMSprop aim to minimize a function, when in our case we need to maximize it.

Another significant architectural change in WGANs is the preference, for instance normalization over batch normalization. Instance normalization offers a more robust and reliable option in the context of WGANs, contributing to smoother convergence and mitigating issues that batch normalization might introduce. Moreover, it's important to avoid using momentum-based optimizers like ADAM [20] with $\beta_1 > 0$, as it can cause training to become unstable. Instead, one should opt for non-momentum-based optimizers like RMSProp [19]. Finally, the authors propose updating the critic's parameters more frequently than the generator's. They suggest updating the generator after five critic updates.

The algorithm in Figure 6.6 describes how the training scheme of a WGAN using gradient penalty is achieved.

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.
Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.

while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

Figure 6.6: WGAN training scheme

In the original paper they admit that enforcing the Lipschitz constraint through gradient clipping is not optimal. If the clipping parameter is large, then it can take a long time for any weights to reach their limit, thereby making it harder to train the critic till optimality. If the clipping is small, this can easily lead to vanishing gradients. In [70] the authors suggest an alternative way to impose the constraint, via gradient penalty. The key idea behind gradient penalty is to add a term to the loss function that penalizes the norm of the gradient of the critic (discriminator) with respect to randomly interpolated data points between real and fake samples. This encourages the gradient to stay within a desired range without explicitly clipping it.

In summary, Wasserstein Generative Adversarial Networks offer a compelling set of advantages that address many of the limitations of classical GANs. These advantages contribute to more stable, interpretable, and efficient training processes, while the model architecture's impact on training stability is significantly reduced, providing greater freedom in designing and adapting architectures for various generative tasks.

Both the traditional cGAN and WGAN, will be used in training the following video stabilization algorithm.

6.3 Purely Generative Video Stabilization Approach

In this section we will discuss an approach that aims to simplify the over-complicated formulation of video stabilization. [33] tackles the task in a purely regressive deep learning fashion, without any motion estimation modules. They address the issue of the lack of supervised data, by providing a dataset generation pipeline that offers an improvement on DeepStab [31], eliminating the perspective mismatch.

6.3.1 Dataset Generation Pipeline

Given an unsteady frame sequence they create its stabilized version through iterative frame interpolation. The idea was originally introduced by [39]. In essence, video frame interpolation methods act like a low-pass filter through generating a middle frame $\widehat{I_{1,2}}$ between two preexisting consecutive frames I_1, I_2. Through the generation of the middle frame, the sequence is alleviated from high frequency jitter. When this process is run in an iterative arrangement, great stabilization results can be achieved. Also, typically a skip parameter is used, which allows for the use of non-consecutive frames which can lead to greater stabilization. For example, when $skip = 2$, the frames I_{t-2}, I_{t+2} will be used to interpolate $\widehat{I_t}$. For the middle frame generation, the authors use the pretrained model CAIN [41], which was trained on the VIMEO [50] triplet dataset. By creating the intermediate frame in an iterative fashion, the results include lesser high frequency noise at the cost of blur accumulation and artifacts at the boundaries of dynamically moving objects. To remove such distortions, a refinement network is used that essentially inpaints the interpolated frame with the necessary information from the original frame. Their refinement network is based on ResNet with a modified version of the channel attention module from CAIN. The modified attention module treats the features with a succession of space-to-depth operations followed by global average pooling, and a 1x1 convolution layer. The output of this layer is passed through a sigmoid function and then multiplied (elementwise) to the input features.

To generate the training dataset, the authors used the unstable videos from DeepStab and created their stable counterparts by interpolating each video for 4 iterations with the skip parameter set to 5. This dataset was made publicly available online and will be referred to as DeepStab-Modded for the rest of this thesis. Although deep iterative frame interpolation offers satisfying results, the distortions it introduces are non-negligible. This combined with the long inference time for each video, constitutes the method far from ideal. A more in-depth review of iterative frame interpolation will be offered in the next chapter.

6.3.2 Network Architecture and Input

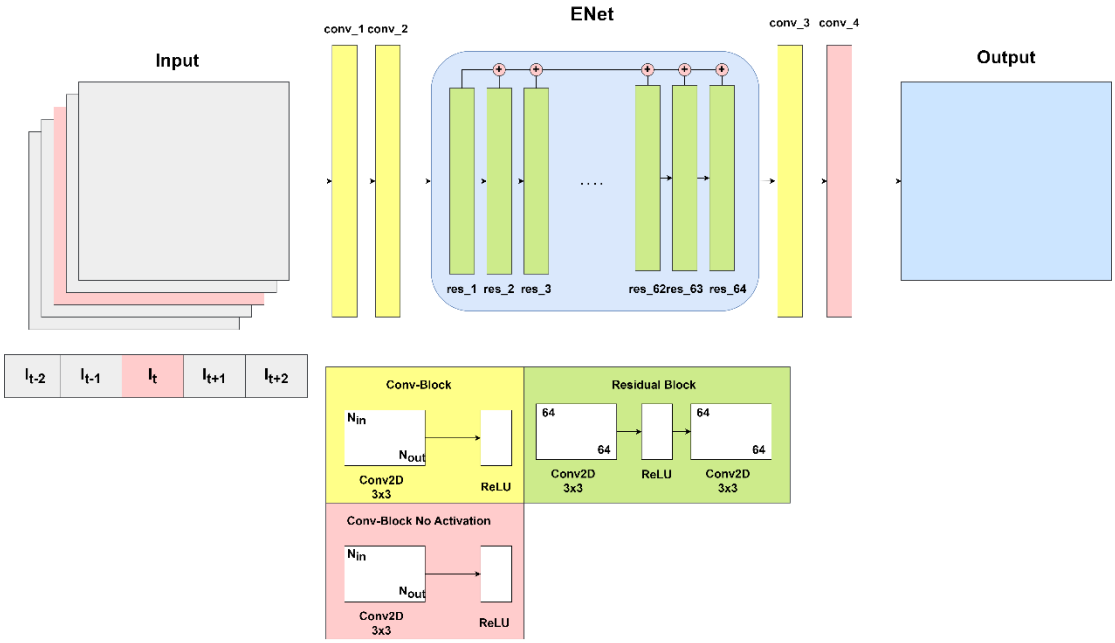


Figure 6.7: Original Network Architecture

In a simple and straightforward approach to video stabilization using DeepStab-Modded, they have minimized perspective discrepancies, enabling a model to focus on the spatial relationships between stable and unstable videos. The authors experimented with various model architectures, including U-Net and ResNet structures, before settling on the baseline architecture for this task. The chosen stabilization network is a modified version of ENet, a super-resolution network, which is based on the ResNet architecture. This network's deep architecture takes multiple input frames, allowing it to leverage spatio-temporal information and a broad receptive field. The model takes five consecutive unstable frames as input $S_t = \{I_{t-2}, I_{t-1}, I_t, I_{t+1}, I_{t+2}\}$ and produces a stabilized version of the middle frame $\widehat{I_t}$. The authors decided to use five input frames for the stabilization network through empirical evaluation.

6.3.3 Training Strategy

The training of the network is broken down into three stages where the learning rate if progressively decreased inspired by the principles introduced [71]. During the first stage the network learns to produce stable frames, that are of inferior quality. In the second part of training the quality of generated frames is enhanced through adversarial training. Finally in stage three, which the authors call ‘Strengthening’, the model is encouraged to learn the abstract reasoning for improving the stability and temporal consistency.

Stage 1: Stable Frame Generation

During this initial part of training the network learns to produce stable intermediate frames given an unsteady sequence as conditional input. Then the generated image is compared to its ground truth counterpart from the DeepStab-Modded dataset. They employ an $\mathcal{L}_2$- based loss function, typically known as MSE (mean-squared- error):

$$\mathcal{L} = \left\| \widehat{I_t} - I_{gt,t} \right\|_2^2$$

The findings in [72] suggest that using this reconstruction loss for image restoration as opposed to a $\mathcal{L}_1$-based one, can lead to blur accumulation. As our task here is not too dissimilar to image restoration, I opted to use the MAE (mean-absolute-error) loss:

$$\mathcal{L} = \left\| \widehat{I_t} - I_{gt,t} \right\|_1$$

The difference in the quality of images produced after training stage 1 using each loss functions can be observed in Figure 6.8.

Figure 6.8: Visual comparison of image quality when using MSE loss (left) and MAE (right)

As for the input of the network, all sequences were resized to 256x256 and scaled to the value range of $[0,1]$. To augment the data, random horizontal and vertical flips were applied alongside random brightness, gamma, hue, and contrast adjustments. Despite the min-max scaling there were some cases where the network would produce solid color artifacts in regions of sudden brightness changes as shown in Figure 6.9. This effect was due to not standardizing the data beforehand. I went ahead and calculated the per-channel mean and standard deviation for the whole DeepStab-Modded dataset, $mean = [0.155,0.161,0.153]$ and $std = [0.228,0.231,0.226]$. I then standardized the data to have $mean = 0, std = 1$ by:

$$Standardized\ Data = \frac{\text{Normalized Data} - \text{mean}}{\text{std}}$$

Figure 6.9: Two cases were artifacts are present due to the lack of standardization. For example in the left image, some solid color artifacts can be seen on the brighlty illuminated part of a street column.

This issue could also be mitigated by introducing learnable normalization layers in the model, such as Batch Normalization. However our model is going to be trained in an adversarial manner, and such layers are typically avoided as they affect adversarial training stability.

The optimizer used for this stage was ADAM [20] with the learning rate set to 1e-4 and the training was run for 5 epochs.

Stage 2: Quality Enhancement

At this point the model has learned to produce stable frames, but they are quite blurry. Inspired by DEBLURGan, an image restoration model, we further finetune the model in an adversarial manner. We will use WGAN-GP that was previously mentioned, that shows great performance is image restoration tasks and is robust to the generator's architecture. In order to deter the model from producing solid-colored frames, we further introduce a VGG-19 perceptual loss $\mathcal{L}_{content}$, that basically functions as a regularization term.

$$\mathcal{L}_{content} = \left\| \phi(\hat{I}_t) - \phi(I_{gt,t}) \right\|_2^2$$

The final loss is defined as:

$$\mathcal{L}_{generator} = \mathcal{L}_{content} + \mathcal{L}_{adversarial}$$

Here $\phi(\quad)$ represents the relu_3_3 of a pretrained VGG-19 network. The loss is essentially the Euclidean distance of the normalized embeddings for the two images. The critic is the same as the one used in DeblurGAN [73], that has alternating

convolutional and Leaky-ReLU operations. The only difference is the substitution of Batch Normalization layers with Instance Normalization layers. As for the adversarial training scheme, the critic and generator both have RMSProp optimizers with the learning rate set to 5e-5. The generator is updated once per five critic updates.
The generator's adversarial loss is:

$$\mathcal{L}_{\text{adversarial}} = -\mathrm{E}_{\mathrm{z}\sim\mathrm{p}_{\mathrm{z}}}\left[\mathrm{Critic}\left(\mathrm{G}(\mathrm{z})\right)\right]$$

And the critic's loss is:

$$L_{\text{critic}} = -\left(E_{x\sim p_{\mathrm{r}}}[\mathrm{Critic}(x)] - E_{z\sim p_z}\left[\mathrm{Critic}\left(G(z)\right)\right]\right) + \lambda * gradient\ penalty$$

With λ = 10, as was empirically determined in Deblur GAN. The gradient penalty term is computed as follows. Through a random coefficient α, we create interpolated samples that lie in a straight line halfway between real and generated data:

$$interpolated = (\alpha * real\ samples + (1-\alpha) * fake\ samples)$$

We compute the critics output for the interpolated samples, and then compute the gradient. We penalize the model by the squared difference of the normalized gradient and the value 1. This last step is what actually enforces the Lipschitz constraint on the critic. This method as we previously discussed is far superior to gradient clipping.
The deblurring capabilities of WGAN-GP become apparent after only a small number of iterations as depicted in Figure 6.10. This stage was run for 5 epochs, and the visual quality of the produced images was directly correlated to the generator's loss value, making training monitoring very straightforward.

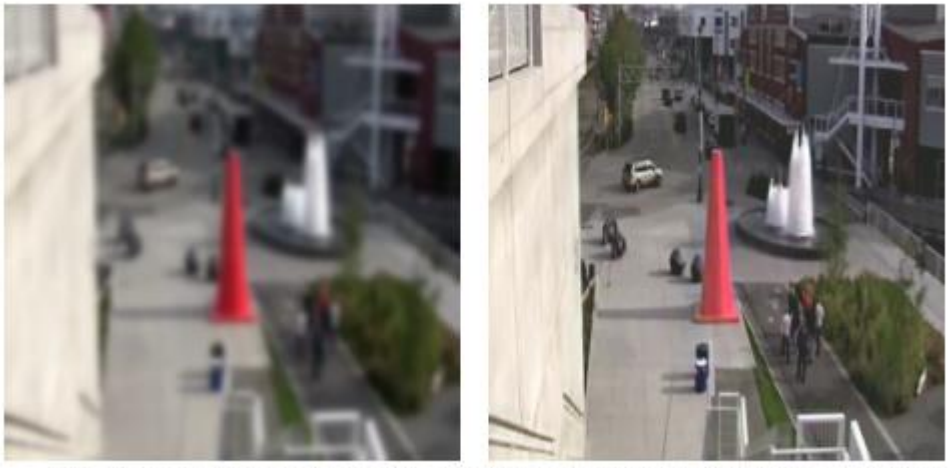


Figure 6.10: Deblurred results after training with the Wasserstein GAN (right) compared to the output of the network after stage 1.

Through this training scheme we have achieved greater image quality. However, our results lack the natural attributes of a real world captured sequence and often contain

temporal inconsistencies. This is due to the fact that up to this point, the model was trained on the synthetic DeepStab-Modded dataset. The ground truth images themselves contained wobble artifacts introduced by frame interpolation. Also, the model has been trained using individual frames and not frame sequences.

Stage 3: Strengthening

In this training stage we will enforce further temporal smoothness and stability. In order to encourage the model to produce temporally smooth sequences we will train it against a temporal discriminator that learns to distinguish between real and generated frame sequences. As the model does not include any motion estimation modules, for stability we will introduce a specially crafted contrastive motion triplet loss, utilizing the video embeddings of a pretrained network for action recognition. The final loss function is a combination of the aforementioned terms and some regulatory terms and is defined as:

$$\mathcal{L} = \lambda_1 * \mathcal{L}_{\phi} + \lambda_2 * \mathcal{L}_{CX} + \lambda_3 * \mathcal{L}_{td} + \lambda_4 * \mathcal{L}_{id} + \lambda_5 * \mathcal{L}_{cml}$$

Each term will be explained in detail, but first we must introduce the new training data that will be used throughout this training stage.

New Training Data:

However, as previously mentioned, DeepStab-Modded contains wobble artifacts around dynamically moving objects and is no longer suitable as we aim to produce natural looking sequences. At the same time using the original DeepStab dataset is also prohibited due to the perspective mismatch. To overcome these issues, the authors create a subset of the original DeepStab dataset, where the corresponding frame pairs have little content variation. They do this through template matching but provide no further information. In my implementation I center crop every corresponding frame pair at 0.75 * frame height, 0.75 * frame width and use the unstable cropped frame as a template. Using OpenCV's [74] normalized correlation coefficient matching method, I compute the correlation score for every matching frame pair in the DeepStab Dataset. The mean correlation score was 0.751 with a standard deviation of 0.154. With this knowledge I created a video pair for every corresponding frame sequence based on two conditions: a) that the correlation score is equal to or greater than 0.751 and b) that the minimum length of such a sequence is a hundred frames. Using this scheme, the resulting cropped subset of DeepStab had approximately 5000 samples. Such a frame pair can be seen in Figure 6.11

Figure 6.11: Cropped subset of DeepStab. The content variation is still present, but it is negligible.

Contrastive Motion Loss:

According to the authors the introduction of the contrastive motion loss is crucial, as it enhances stability by $2-3\%$. We will utilize a video action recognition model that was pretrained on the KINETICS-400 [48] dataset as a video encoder. By passing steady and unsteady corresponding video sequences from the cropped subset of DeepStab we can acquire the positive $\boldsymbol{A}$ and negative $\boldsymbol{N}$ embeddings respectively. By passing the generated sequences through the encoder we get the positive $\boldsymbol{P}$ embeddings. The loss function is defined as:

$$\mathcal{L}_{cml} = max(d(A,P) - d(A,N) + \alpha, 0)$$

Where $d(x,y) = \|x - y\|_2$ is the Euclidean distance of the embeddings. The margin α is a hyperparameter that determines the minimum difference between the distances that the model should aim for that was equated to 1. The coefficient for this loss, λ_5 was empirically set to 0.01. The aim of this function is to bring the distribution of generated sequences closer to their real-world steady counterparts. This is illustrated in Figure 4.13.

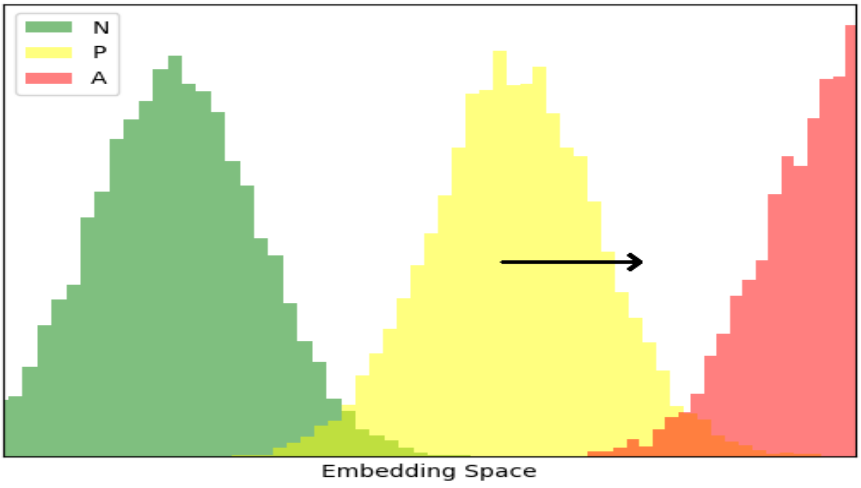


Figure 6.12: The aim of the contrastive motion loss is to bring the distribution of the generated sequences embeddings P(yellow) closer to the real-world steady sequence embeddings $A(red)$.

The authors use the pretrained model described in [75], however the weights were available in '.lua' format used with the Torch [76] library. As I am using PyTorch [25] instead, I opted not to use this model in order to avoid compatibility issues. So, I had to determine which model to use instead. [77] provides an in-depth comparison of different ResNet architectures for Action Recognition. These include:

- **R2D(2D convolutions over the entire clip)**: This approach applies 2D convolutions to each frame of a video clip independently. It treats each frame as a separate image and uses traditional 2D convolutional neural networks for video analysis.
- **R3D (3D convolutions)**: 3D convolutions are designed for spatiotemporal data, such as video sequences. They consider the spatial and temporal dimensions simultaneously, making them suitable for action recognition tasks.
- **MCx (Mixed 3D2D Convolutions)**: MCx architecture combines 3D and 2D convolutions in a mixed manner. It allows the network to capture both spatial and temporal features while avoiding excessive computational complexity.
- **rMCx (Recurrent Mixed 3D2D Convolutions)**: rMCx extends MCx by introducing recurrent connections to capture longer-term temporal dependencies in video data. Recurrent layers help model sequential information within video clips.
- **R(2+1)D (2+1D Convolutions)**: R(2+1)D architecture separates the 3D convolution into two parts: a 2D spatial convolution followed by a 1D temporal convolution. This decomposition reduces the number of parameters and computation, making it efficient for video analysis.

There is also a newer model architecture, SlowFast ResNets [78] which show promising results in video action recognition. SlowFast networks have two pathways: a "slow" pathway and a "fast" pathway, which capture different temporal scales: A) The "slow" pathway operates at a lower frame rate, capturing the longer-term temporal context. B) The "fast" pathway operates at a higher frame rate, capturing short-term dynamics. These two pathways are combined to create a more comprehensive representation of spatiotemporal features.

As we can see there is a plethora of options, and the choice of model is not so straightforward. In Figure 6.13 we see a table with the classification accuracies of the different model architectures.

Net	# params	Clip@1	Video@1	Clip@1	Video@1
Input		8×112×112		16×112×112	
R2D	11.4M	46.7	59.5	47.0	58.9
f-R2D	11.4M	48.1	59.4	50.3	60.5
R3D	33.4M	49.4	61.8	52.5	64.2
MC2	11.4M	50.2	62.5	53.1	64.2
MC3	11.7M	50.7	62.9	53.7	64.7
MC4	12.7M	50.5	62.5	53.7	65.1
MC5	16.9M	50.3	62.5	53.7	65.1
rMC2	33.3M	49.8	62.1	53.1	64.9
rMC3	33.0M	49.8	62.3	53.2	65.0
rMC4	32.0M	49.9	62.3	53.4	65.1
rMC5	27.9M	49.4	61.2	52.1	63.1
R(2+1)D	33.3M	**52.8**	**64.8**	**56.8**	**68.0**

Figure 6.13: [77] Action recognition accuracy for different forms of convolution on the Kinetics validation set. All models are based on a ResNet of 18 layers and trained from scratch on either 8-frame or 16-frame clip input. R(2+1)D outperforms all the other models

It is apparent that the R(2 + 1)D architecture outperforms all other models. However, this does not necessarily mean that it is the optimal model for encoding our sequences. For this reason, I devised a test to determine which model to use. From the cropped subset of DeepStab I chose 100 random steady/unsteady sequence pairs and calculated their embedding distances with four different models as encoders [R3D_18, MC3_18, R(2+1)D_18, SLOW_R50]. The input sequences were standardized using the KINETICS-400 [48] per channel mean = [0.43216, 0.394666, 0.37645] and standard deviation = [0.22803, 0.22145, 0.216989]. Across this 100-sequence sample the 4 models produced the mean embedding distances shown in Figure 6.14.

Models	Mean Steady/Unsteady Embedding Distances
R3D_18	4.6964
MC3_18	4.5805
R(2+1)D_18	5.5579
SLOW_R50	3.9989

Figure 6.14: Mean embedding distances for different action recognition models.

From these findings, we can infer that the R(2+1)D_18 model is better able to differentiate between steady and unsteady video sequences and therefore it will be the best choice for the video encoder.

Temporal Smoothness and Consistency:

Up to this point, the model was trained using individual frames rather than sequences of frames. The results often contain distortions such as wobble artifacts. To overcome this issue, the authors propose using a temporal discriminator that learns to differentiate between real and generated stable frame sequences. As they do not specify any architecture details my model was based on 3D convolutions followed by fully connected layers, with the last one being passed through a sigmoid activation function. The generator G and discriminator D are trained using a vanilla GAN adversarial loss. For the generator the loss is defined as:

$$\mathcal{L}_{td} = E_{F_t}\left[1 - log\left(D\big(G(F_t)\big)\right)\right]$$

And the discriminator loss as:

$$\mathcal{L}_{discriminator} = E_{F_t}\big[1 - log\big(D(F_t)\big)\big] + E_{F'_t}\left[log\left(D\big(G(F'_t)\big)\right)\right]$$

The contribution parameter for this loss was empirically set to $\lambda_3 = 0.1$.

Content Preservation / Regulatory Terms:

The model might learn to predict solid-colored frames when using the proposed temporal discriminator and contrastive motion losses. So, the addition of content preserving losses is mandatory. These include a VGG-19 perceptual loss identical to the one used in Stage 2, a contextual loss as proposed in [79] and an image discriminator. The authors measure contextual similarity using VGG-19 features and employ the unaligned, unstable images as targets for this contextual similarity when generating frames. This choice is made because we aim to ensure that the stabilized videos maintain the content of the input videos across various spatial locations. The loss function is formulated as follows:

$$\mathcal{L}_{CX} = -log\left(CX\Big(\phi^l(f'_t), \phi^l(f_t)\Big)\right)$$

Where f'_t denotes the generated steady frame, f_t is its corresponding ground truth frame and ϕ^l represents features extracted through layer l (relu_3_3) of a pre-trained VGG-19 network. Lastly CX denotes the contextual similarity between the extracted VGG-19 feature embeddings. More details on contextual similarity are provided in [79].

In addition to this loss the authors include a VGG-19 perceptual loss and image adversarial loss to guarantee high quality image generation. So, for this third part of training, we have to use a total number of 5 models, our generator, the video encoder, the temporal discriminator, the image discriminator and the VGG19 feature encoder.

The contribution coefficient for both the contextual and the perceptual losses was set to one $\lambda_1 = \lambda_2 = 1$. Due to GPU memory limitations, I did not include the image discriminator $\lambda_4 = 0$.

Implementation Details:

The optimizer used during this stage for both the generator and the temporal discriminator was ADAM [20] with a reduced learning rate of 1e-5. The authors crop the input frames at 160x160 and opted for 16-frame sequences. These parameters were prohibitive for my GPU, so instead I opted for 128x128 sample dimensions with 8-frame long sequences. The cropped Dataset is much smaller than the one previously used, and the training now was run for 10 epochs.

6.3.4 Modifications

The proposed ENet network is very deep with 64 residual blocks and also every convolutional layer has 64 filters. Its big size enables it to learn spatio-temporal pixel relations in depth, but this however comes at the cost of inference speed. With my GTX970 graphics card, it took approximately two minutes to produce a 400-frame video. This inference time can be improved if the input frame sequences are fed into batches. However, as an alternative I wanted to train another smaller network architecture to test if it is able to achieve comparable results with smaller inference time. I opted to use a U-Net architecture with integrated gated convolutions [34]. Gated convolutions essentially are learnable binary masks for each pixel location that are activated using a sigmoid function. This mask gets multiplied with the original feature map elementwise. The integration of this module has shown great success in image inpainting tasks, where the inputs might contain holes or unseen regions. Though our task is video stabilization, it is not too dissimilar to image inpainting. The network comprises of five encoder-decoder blocks each with gated convolution modules as shown in Figure 6.15.

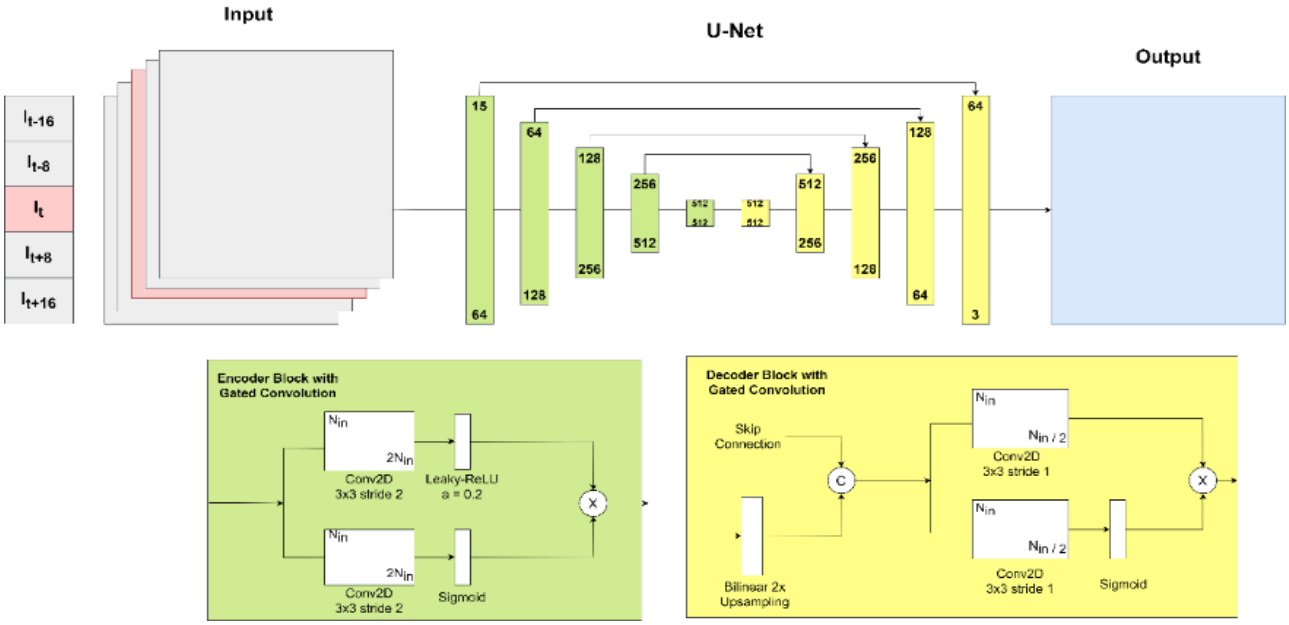


Figure 6.15: Proposed U-Net Network

Another modification I made is the conditional input. I chose the input $S_t = \{I_{t-16}, I_{t-8}, I_t, I_{t+8}, I_{t+16}\}$ where the input frames are further away from each other temporally. In the analogy of the network as a low-pass filter, the modification to the input could be viewed as using a larger kernel. The training scheme for the modified network remains exactly the same. When using the modified input with the already trained ENet, the results contained large amounts of artifacts.

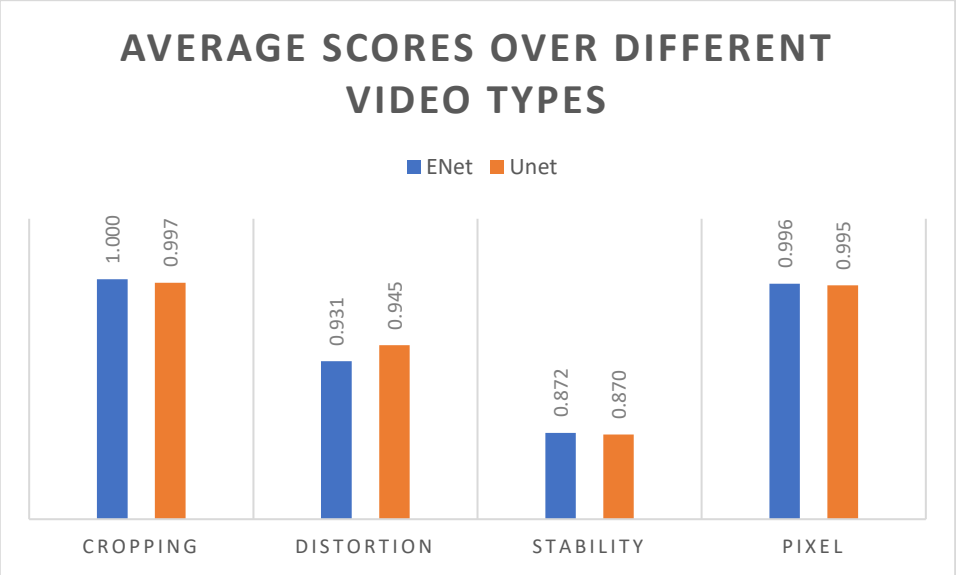
6.3.5 Results and Discussion

ENet Evaluation Scores

Categories	Cropping	Distortion	Stability	Pixel
Crowd	1.000	0.966	0.822	0.997
Parallax	1.000	0.972	0.885	0.995
QuickRotation	1.000	0.890	0.942	0.997
Regular	1.000	0.907	0.806	0.997
Zooming	0.999	0.922	0.904	0.993

UNet Evaluation Scores

Categories	**Cropping**	**Distortion**	**Stability**	**Pixel**
Crowd	0.996	0.969	0.842	0.995
Parallax	0.997	0.956	0.877	0.993
QuickRotation	0.998	0.947	0.930	0.997
Regular	0.996	0.914	0.810	0.996
Zooming	0.997	0.937	0.894	0.992



The proposed U-Net model achieves comparable results to the much deeper E-Net, and even better distortion scores. However, its great advantage lies in its inference speed. For 360×640 videos with my GTX 970 graphics card the original network needs 0.284 seconds per frame, while the modified one only 0.102.

A visual comparison of the results of both models versus the original frame sequence is shown in Figure 6.16. We can clearly see that both models alter the color profile. Measuring this effect, was the intuition for me introducing the pixel score. The results produced with ENet have a slightly green hue, whereas UNet's results produce a warmer tint. Both cases are not ideal, but the alteration is negligible.

My implementation of this algorithm, training code, model weights and inference scheme can be found here https://github.com/btxviny/Deep-Motion-Blind-Video-Stabilization.

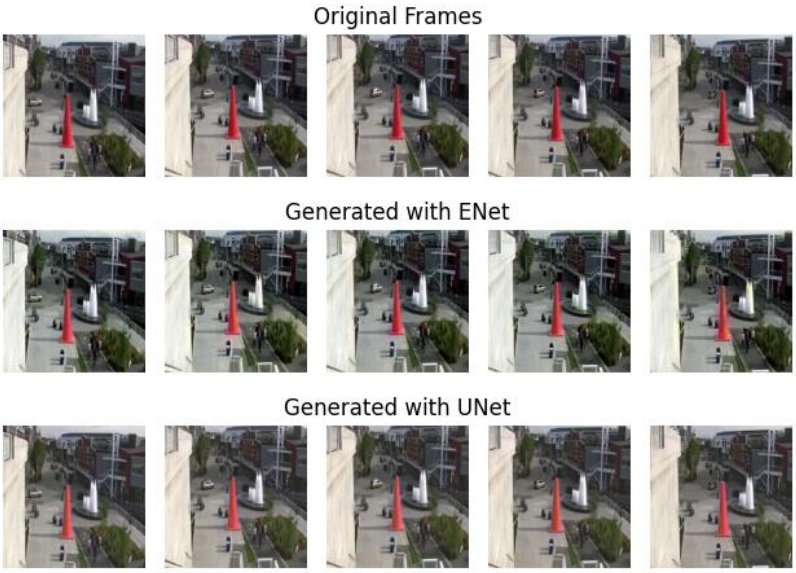


Figure 6.16: Visual comparison of the ENet and UNet generated results.

Chapter Summary

In this chapter, we delved into methods for addressing video stabilization using conventional deep learning approaches, relying on supervised data. Key concepts covered included transfer learning, adversarial training, and the integration of deep learning-based perceptual losses. However, the upcoming chapter will explore a distinct approach, shifting the focus to deep learning frame interpolation as a means to achieve stable video output.

Chapter 7: Video Stabilization through Frame Interpolation

Introduction

As we briefly introduced in 6.3.1, an unsteady video sequence can be stabilized through frame interpolation which essentially operates like a low-pass filter. By generating an intermediate/middle frame $\widehat{I_{t+0.5}}$ between two preexisting consecutive frames I_t, I_{t+1} in an iterative manner we can alleviate the sequence of jerkiness and sudden motion, often present in amateur captured video. Through this process we create the intermediate frame between two adjacent frames. This idea was originally introduced by [39]. From an interpolation standpoint, the interpolated middle frame serves as the representation of the frame that would have been recorded between two consecutive frames. Consequently, the interpolated frame is indicative of the temporal midpoint, presumed to align precisely with the halfway point of inter-frame motion. Through the course of this chapter, we will dive into Deep Learning Frame Interpolation Techniques and how they can be finetuned for video stabilization. We will discuss my implementation of DIFRINT [40], a deep neural network trained specifically for video stabilization. We will then compare it to CAIN [41], an award-winning pretrained network for frame interpolation, which was used in 6.3.1 for generating a synthetic supervised dataset.

7.1 DIFRINT Implementation

DIFRINT proposes a deep frame interpolation architecture intended for video stabilization, that is depicted in Figure 7.1. The training will take place in an unsupervised manner. We will go over the training and testing scheme separately as they differ and finally, we will introduce two inference parameters: number of iterations and the 'skip' parameter, which are both crucial for achieving high quality stabilization.

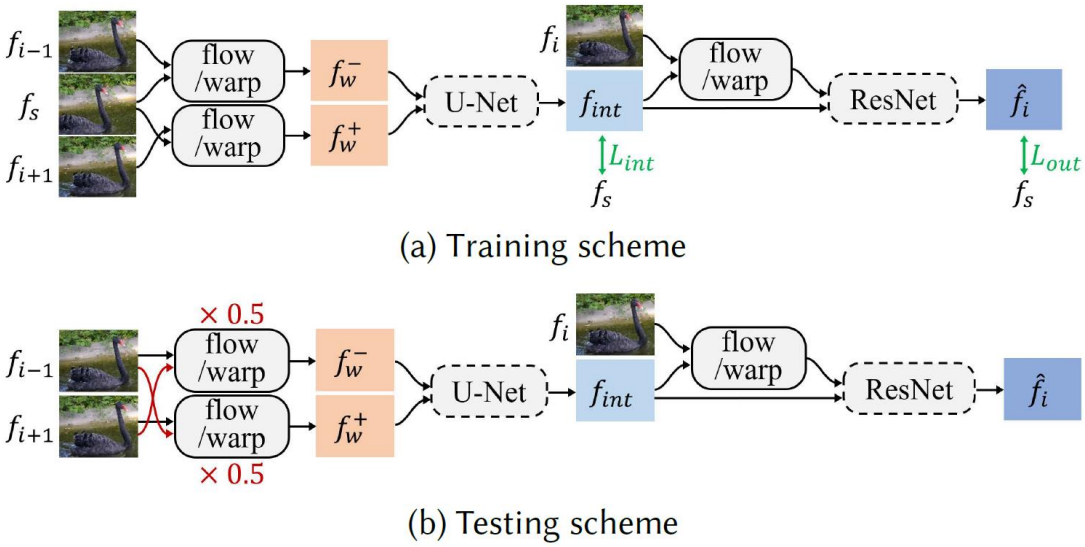


Figure 7.1: The DIFRINT [40] framework during (a) training and (b) testing.

7.1.1 Training Scheme

The basic idea is that given two adjacent frames f_{i-1}, f_{i+1} we want to generate the intermediate frame f_{int}. The first part is generating the input to the network. Instead of simply concatenating the two images and feeding them to the network, the authors warp the two frames towards the intermediate frame fi, through optical flow estimated with PWC-Net [8], thus producing the warped frames f_w^-, f_w^+. Both these frames represent half-way points that originate from f_{i-1}, f_{i+1}. The warped frames are then concatenated across the channel dimension and fed into a U-Net module, which learns how to combine information at different scales. Each individual input frame may contain unseen regions, but combined they complement each other, enabling the U-Net to learn how to fill those regions. However powerful the U-Net might be at learning latent space representations at different scales, using its outputs in an iterative manner is prone to introduce blurring and artifacts. This is more prominently apparent at the boundaries of dynamically moving objects. To mitigate this effect the authors, add a ResNet module in series, with the aim to reintroduce fine details into the interpolated output of the U-Net. The input to the ResNet is the interpolated frame along with the original frame f_i warped towards f_{int} which generate the final, high-quality frame $\hat{f}_i$. A key issue that arises when attempting to

train the model is the absence of a ground truth middle frame. Using f_i is not ideal, as it's not guaranteed to be the true intermediate frame. For this reason, the authors create a pseudo-ground truth frame f_s, which essentially is a warped version of f_i. More specifically each frame f_i is randomly translated in the vertical or horizontal direction by a small scale (at most one eighth of the frame width). During training both f_{i-1}, f_{i+1} are warped towards f_s aiming to reconstruct it.

7.1.2 Loss Functions

For both trainable modules of the network namely the U-Net and ResNet
We will use a combination of a pixel-wise loss and a VGG-19 perceptual loss. The choice for the pixel-wise loss is $\mathcal{L}_1$-based inspired by in [72], which shows that $\mathcal{L}_2$-based losses might introduce blurry results. For the U-Net module the loss is defined as:

$$L_{out} = \left\| f_s - \widehat{f_i} \right\|_1 + \left\| \phi(f_s) - \phi\left(\widehat{f_i}\right) \right\|_2^2$$

Then a similar specific loss is applied to U-Net, as the authors found its use to speed up training and lead to better results:

$$L_{int} = \|f_s - \mathrm{f_{int}}\|_1 + \|\phi(f_s) - \phi(f_{int})\|_2^2$$

Here ϕ(), denotes the output of the relu3_3 layer of a pretrained VGG-19 network.

7.1.3 Testing Scheme

After training is fully completed the network can be now used for frame interpolation. Two neighboring frames f_{i-1}, f_{i+1} are warped halfway towards each other using bi-directional optical flow (x 0.5). The warped frames are fed into the U-Net, yielding the interpolated frame f_{int} which is refined by the trained ResNet using f_i. The algorithm will be run for multiple iterations, which aims to enforce greater stability. A larger number of iterations can be equated to applying a low-pass filter with a larger kernel. The advantages of repeating the algorithm more than once, can be shown in Figure 7.2, which compares a sequence of frames generated after one iteration as opposed to the same sequence after five iterations. Figure 7.3 shows which frames affect the generated output throughout different algorithm iterations. However, with each iteration, it is inevitable that artifacts and blur will occur. For this reason, it is essential that we use the original frame f_i along with the interpolated result as input to the ResNet for all subsequent iteration. This will keep as much of the original information as is possible. Another parameter for managing stability is the skip parameter. This parameter dictates which frame to use for interpolation. When $skip = 1$,then the frames f_{i-1}, f_{i+1} will be interpolated to generate $\widehat{f_i}$. Setting this parameter to 2, means that f_{i-2}, f_{i+2} will be used as conditional input and so on.

Another important detail is that from the total video sequence, the first and last 'skip' number of frames will remain unaltered.

Figure 7.2: [40] Comparison of a corresponding frame sequence after one iteration (upper row) as opposed to the same sequence after five iterations (lower row). The results after one iteration show some temporally local oscillation, which has been significantly mitigated after five iterations.

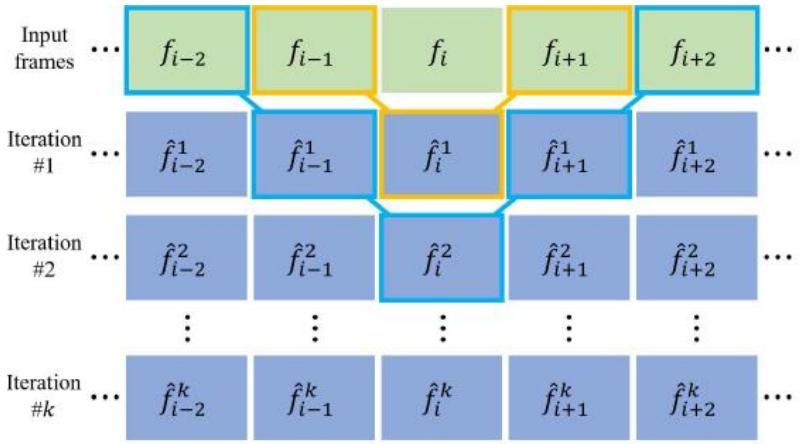


Figure 7.3: [40] Further Iterative interpolation, leads to more distant frames affecting the interpolated result, thus introducing greater stability.

7.1.4 Implementation Details

1. Training Data: The authors opted to use the DAVIS [49] dataset for training, as it contains a variety of scenes, containing static and dynamic backgrounds with color variations. This dataset includes approximately 8.000 samples which are augmented through random flips, and color modifications.

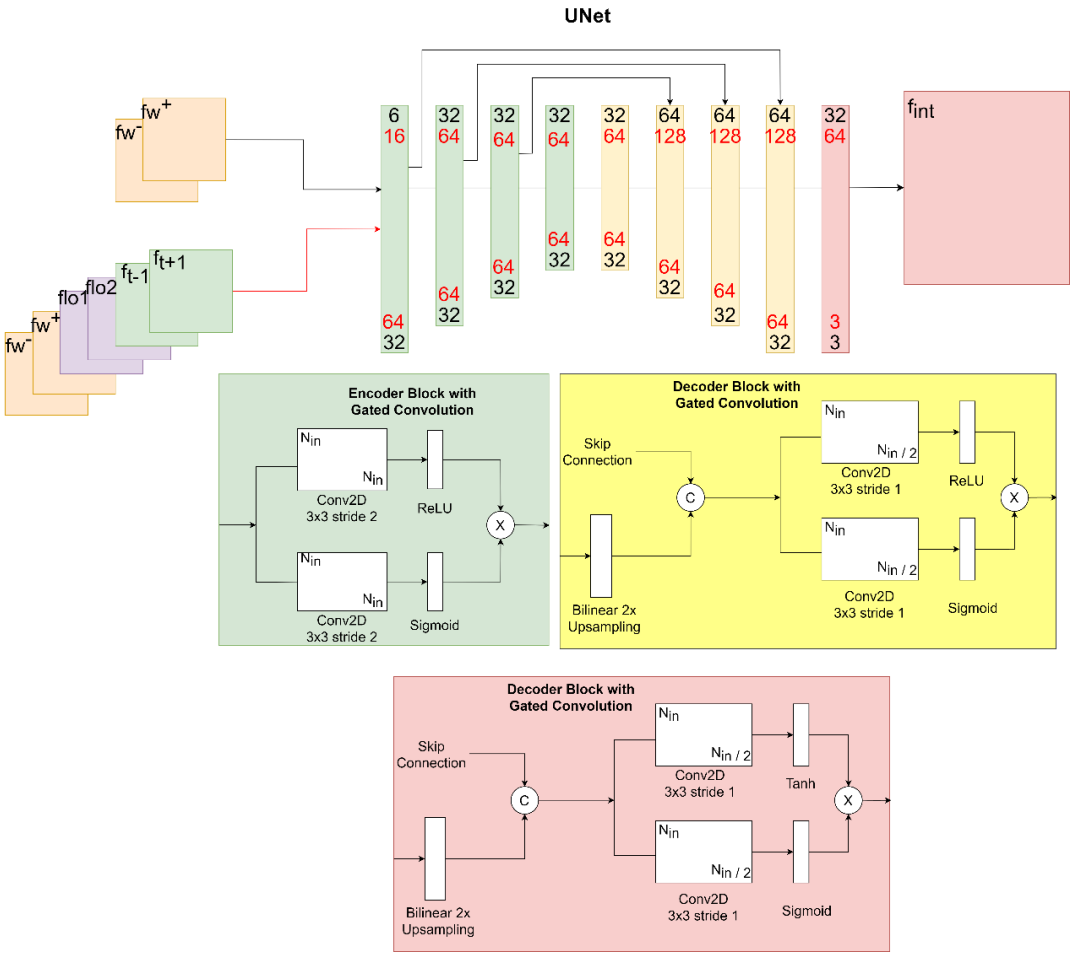


Figure 7.4: U-Net architecture. The black arrow and black channel numbers refer to the architecture proposed in the original paper. The red channel numbers along with the modified red input are for my alternative implementation.

2. Network Architecture: The first choice is the optical flow estimation network to use. Following the original architecture, I also used PWC-Net that requires custom correlation and cost volume layers to be built beforehand. However, any pretrained network is also a valid choice. In the original paper the U-Net module employed 3 skip connections and each convolutional layer employed 3x3 filters with the number of hidden channels set to 32. Each encoder layer downsized the input feature map by a factor of 2, through adjusting the kernel stride to be equal to 2 avoiding using pooling layers. At the decoder part of the network upscaling was achieved through bilinear interpolation instead of using learnable transpose convolutional layers. This was because such modules can interfere with training stability as they are prone to produce checkerboard patterns. The non-linear activation function of all intermediate layers was Leaky-ReLU to avoid vanishing gradients, whereas the last layer had Tanh as its activation function to bring the output to the [-1,1] range which is also the range used for the input. This network is shown in Figure 7.4. As for the ResNet architecture shown in Figure 7.5, 5 residual blocks were utilized, with the number of hidden channels set to 32 and the filter size set to 1x1. The latter choice was to minimize the noise from adjacent

pixels and to achieve greater inference speed. The authors also implemented gated convolutions at each convolutional layer for both U-Net and ResNet.

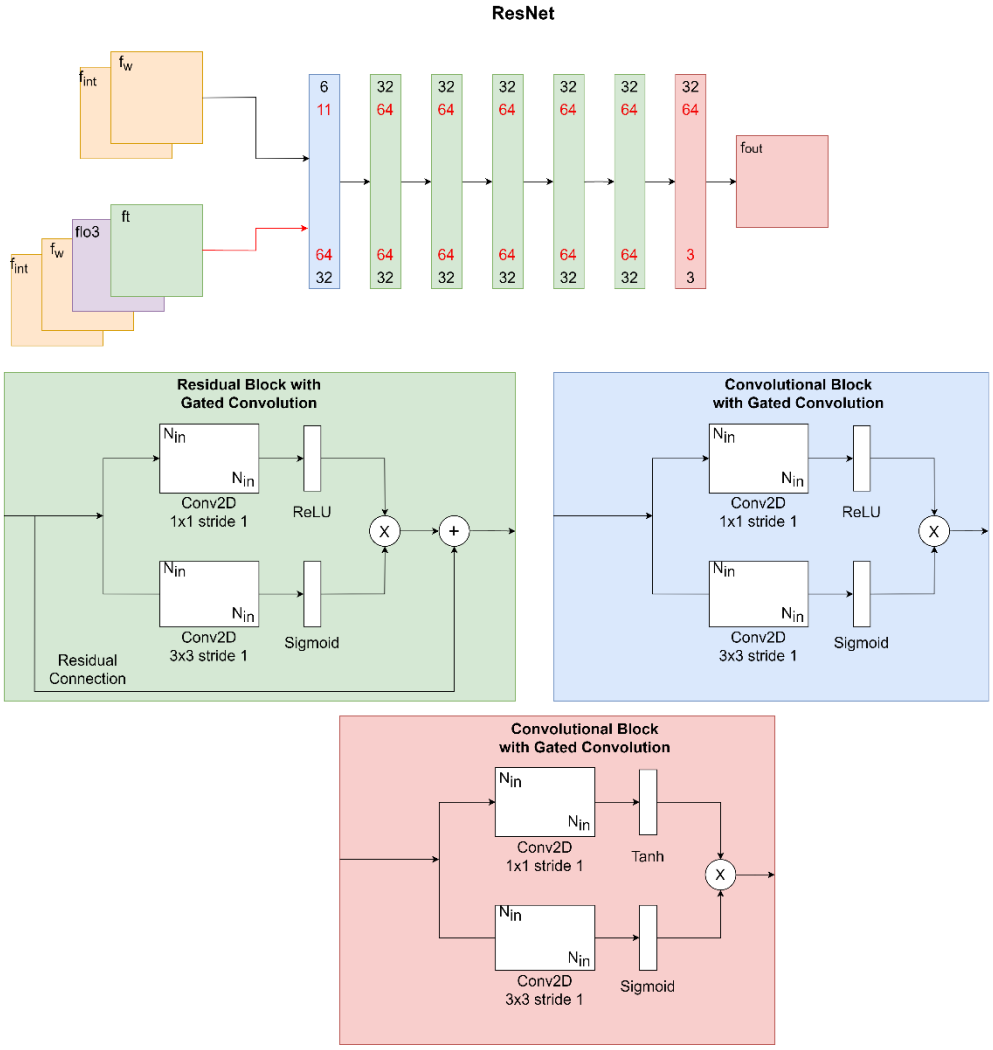


Figure 7.5: ResNet architecture. The black arrow and black channel numbers refer to the architecture proposed in the original paper. The red channel numbers along with the modified red input are for my alternative implementation.

3. Training Settings: All input samples were resized to 256x256 and scaled to the range [-1,1]. In addition, random augmentations were applied to avoid any potential bias. The choice of optimizer was ADAM [20] with $\beta_1 = 0.9, \beta_2 = 0.999$ and the learning rate was set to 1e-3. The batch size was set to 8 samples instead of 16 due to GPU memory limitations. The framework is trained for 200 epochs. I applied a linear decay to the learning rate after epoch 100: $learning\ rate = 0.001 - 0.00001 * (epoch - 100)$.

7.1.5 Discussion and Modifications

Modified Architecture and Input

Using the original network described in the paper, I could not replicate their visual results. The generated results are quite blurry as shown in Figure 7.6. For this reason, I tried changing the number of hidden channels from 32 to 64 for both trainable networks. This led to a huge increase in the number of total trainable parameters and increased inference time. I also modified the input to the networks. More specifically since our PWC-Net optical flow estimator already computes the flow from f_{i-1} to f_s, from f_{i+1} to f_s and finally from f_t to f_{int} I decided to include them as input to the trainable modules. Now the U-Net's input is f_w^-, f_w^+, $flow_{i-1,s}$, $flow_{i+1,s}, f_{i-1}, f_{i+1}$ all concatenated across the channel dimension resulting in an input tensor of shape [batch size, 16, height, width]. The modified ResNet's input is f_{int}, f_t warped towards f_{int} denoted as f_w , the corresponding flow $flo3$ and f_t . All these inputs concatenated result in a tensor of shape [batch size, 11, height, width]. The modified architectures for both networks are shown in Figures 5.5, 5.6 respectively denoted with red arrows and numbers. Despite these modifications the framework still produced very blurry results.

Figure 7.6: Visual Results of the modified architecture

Modified Training Dataset

Since the release of the paper, a newer dataset, Vimeo-Triplet [50] has been released. This dataset is specifically designed for the task of frame interpolation and contains three consecutive video frames where the second frame is guaranteed to be the ground truth middle frame. As this dataset has shown great results in training CAIN [41] which will be further explained in the upcoming section, I opted to use it for training DIFRINT as well. Now training is implemented in a purely supervised manner, without having to create pseudo-ground truth intermediate frames. This dataset is far larger, containing approximately 80.000 samples.

I also train the UNet and ResNet components separately. For the UNet module I set its conditional input to be the one shown in red in Figures 7.4, 7.5, with the number of hidden channels set to 64. I initialized the UNet model with the learned weights from training on the DAVIS dataset and set the learning rate to a relatively small value of 5e-5. I then trained it for a total of 10 epochs. As for the ResNet module, we can no longer use the original frame f_t in each conditional input, since it is now the target image we want to achieve. Its input was set to be f_{int} and f_t warped towards f_{int} denoted as f_w same as the original implementation in the paper. Because, the UNet network had already learned to produce realistic results, the ResNet only took about 3 epochs to converge. The optimizer used was ADAM with a learning rate of 1e-4. Using the modified input, architecture, and training scheme the framework was able to learn frame interpolation more efficiently, producing visually pleasing results.

Testing settings

For inference, the authors suggest using 5 iterations with the skip parameter set to 2. I found that in scenes with dynamically moving objects setting this parameter to 1 generates less distorted results. Any value for the skip parameter larger than 2 introduces heavy distortion artifacts as shown in Figure 7.7. In cases with severe camera motion, running the algorithm for more than 5 iterations can yield better results.

Figure 7.7: a) In the upper row we can see an interpolated frame sequence using a skip parameter of 2. There are no apparent distortion artifacts. However, in the lower row where the skip parameter was set to 5, there are obvious artifacts around the person's arms and there is also a heavy amount of blur.

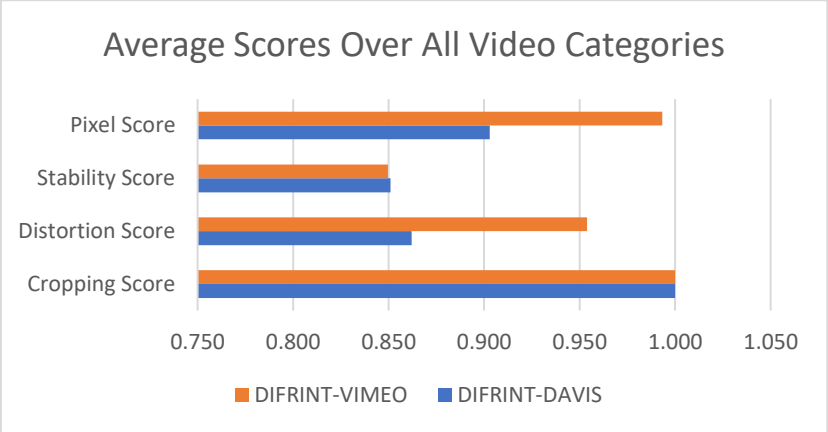
7.1.6 Evaluation Results

DIFRINT-DAVIS

Category	Cropping Score	Distortion Score	Stability Score	Pixel Score
Crowd	1.000	0.930	0.720	0.880
Parallax	1.000	0.900	0.880	0.911
QuickRotation	1.000	0.780	0.930	0.922
Regular	1.000	0.820	0.805	0.879
Zooming	1.000	0.880	0.920	0.923

DIFRINT-VIMEO

Category	Cropping Score	Distortion Score	Stability Score	Pixel Score
Crowd	1.000	0.978	0.722	0.994
Parallax	1.000	0.950	0.863	0.991
QuickRotation	1.000	0.912	0.929	0.996
Regular	1.000	0.967	0.810	0.993
Zooming	1.000	0.962	0.924	0.992



From the comparative evaluations results we see that the instance of DIFRINT finetuned on the Vimeo-Dataset, heavily outperforms the instance trained on DAVIS on all scores except Stability and Cropping. Both models score very high on cropping as they do not introduce any black borders. The difference in the stability scores is negligible and it's a necessary trade off in order for the Vimeo trained model to produce clear images. The visual comparison of the results generated by both models after 5 iterations with the skip parameter set to 2 is shown in Figure 7.8.

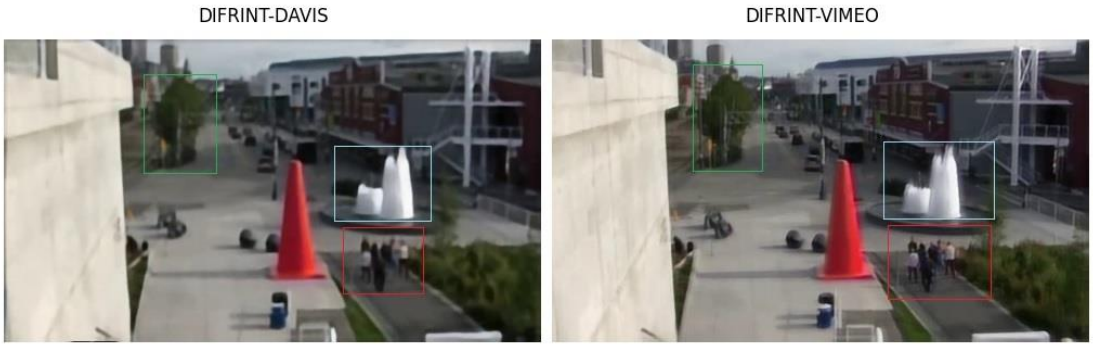


Figure 7.8: Visual comparison of the results of the two models. In the highlighted sections of the images the difference in sharpness and clarity is very apparent.

However, this algorithm struggles in terms of its inference speed. On top of using an optical flow estimation network which already needs time to estimate optical flows, we also increased the number of hidden channels to 64. This led to a total number of 11.600,920 in the case of the Vimeo trained model. It takes about 10 minutes to stabilize a 40 second video with spatial dimensions 360×640, with 5 iterations.

I provide the training code and test scheme for the network finetuned on the VIMEO dataset here https://github.com/btxviny/Video-Stabilization-through-Frame-Interpolation-using-DIFRINT.

7.2 Frame Interpolation With CAIN

In this section we will perform video stabilization through frame interpolation in the same manner as DIFRINT but substitute the U-Net model with CAIN. This network was trained solely for the task of frame interpolation. Through the use of this network, the authors of DMBVS create their synthetic dataset as we saw in 6.2.1. They argue that conventional video frame interpolation models (specifically trained for this task) can outperform DIFRINT in handling dynamic motion scenarios and produce more stable and higher quality videos. We will introduce the network's architecture, training strategy and its advantages over DIFRINT. Finally, we will discuss the inference scheme used to generate stable videos.

7.2.1 Network Architecture and Loss function

The key idea that differentiates CAIN from other frame interpolation models, is that does not require a pretrained optical flow estimation network. It uses the innovative channel attention module [80], which is also the reason for the network's name (Channel Attention Is All You Need). Attention mechanisms play a crucial role in directing a neural network's focus towards significant regions within its feature representations. The applications of attention in deep neural networks are diverse, spanning across various domains such as sequence-based models, image classification, image localization, and image super-resolution and more.t

The networks input are two neighboring frames I_1, I_2 which are concatenated across the channel dimension. The input then gets down-sampled using a learnable PixelShuffle module [81] instead of traditional strided convolution. The key advantage of pixel shuffle down-sampling is that it reduces the spatial dimensions by rearranging pixels, while avoiding the loss of information that can occur with strided convolutions. It's particularly useful when dealing with tasks like image super-resolution or generating high-resolution images from low-resolution inputs. Then the transformed input is fed through a series of 5 residual groups each consisting of 12 residual blocks that incorporate channel attention modules. The Channel Attention Module using average pool reduces the dimensions of its input feature map to $1 \times 1 \times C$. The

following two 1×1 convolutional layers operate similar to dense layers, regressing C coefficients. These coefficients are then utilized to perform element-wise multiplication with the channels of the original input feature map. Finally, the output feature map gets upscaled with an upsampling PixelShuffle module producing $I_{1,2}$. The overall architecture of the model is shown in Figure 7.9.

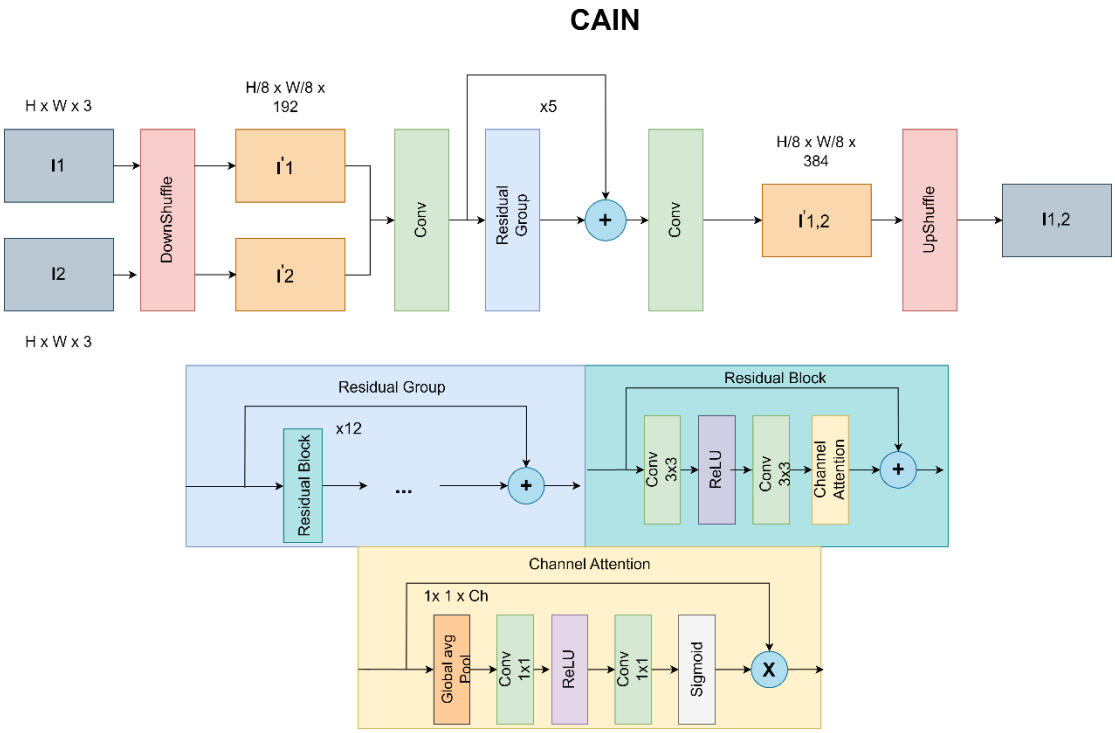


Figure 7.9: Architecture of CAIN network

The network is trained using the Vimeo-Triplet Dataset using a combination of an $\mathcal{L}_1$ and VGG-19 perceptual loss: $\mathcal{L} = \lambda_1 * \left\|\widehat{I_t} - I_{gt}\right\| + \lambda_2 * \left\|\phi\left(\widehat{I_t}\right) - \phi\left(I_{gt}\right)\right\|_2^2$, where $\phi()$, denotes the output of the relu3_3 layer of a pretrained VGG-19 network. I will not attempt to retrain this model. Instead, we will use its pretrained weights which yield exceptional results.

7.2.2 Inference Scheme

Much like DIFRINT, when using CAIN for video stabilization in an iterative manner, blur and distortion artifacts are produced with each successive iteration. However, in this case they mainly appear at the boundaries of dynamically moving objects. The authors of DMBVS train ResNet model as a refinement network with the purpose of reintroducing information from the original unsteady frames that is lost through successive iterations. This model consists of five residual blocks with integrated attention mechanisms. Each convolutional layer uses a 1x1 filter to avoid noise accumulation and to increase inference speed. This model takes as input the output CAIN denoted as $f_{t,int}$ along with four neighboring original unsteady frames

$[U_{t-2}, U_{t-1}, U_{t+1}, U_{t+2}]$ and aims to reproduce the refined version of $f_{t,int}$, denoted as $f_{t,ref}$. It is trained using a combination of an $\mathcal{L}_1$ and a gradient map loss. The gradient map loss calculation entails computing the per-channel gradient magnitude for the input and target and estimating their $\mathcal{L}_1$ difference. The gradients in the vertical and horizontal direction are computed by convolving each channel i with the following Sobel kernels [82] respectively.

$$G_y^i = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} * I_i$$

$$G_x^i = \begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} * I_i$$

Then the magnitude of the gradient for each channel $G_c(I_i)$ is computed as:

$$G_c(I_i) = \sqrt{\left(G_y^i\right)^2 + \left(G_x^i\right)^2}$$

Finally, the gradient magnitudes for all channels are concatenated in a single tensor for both the input and target images denoted as $G(I)$, and then the loss is estimated as the $\mathcal{L}_1$ distance of these tensors.

$$GM\left(\hat{I}, I_{gt}\right) = \left\| G\left(\hat{I}\right) - G\left(I_{gt}\right) \right\|$$

What it essentially penalizes is the difference of the high frequency edges and corners of each channel of the two images. This is very useful in our specific task of image restoration/ refinement.

As no specific training details are mentioned, I trained it on the DAVIS dataset instead of VIMEO, because we need more than three frames as conditional input. The model was set in series after a frozen weights instance of CAIN. The optimizer was ADAM with a learning rate of 2e-4. The proposed architecture is very light-weight and its training converged very fast (less than two epochs). I tried decaying the learning after a specific number of iterations, but there was no further improvement. This was no issue however, as the visual results are satisfactory.

A key difference with the DIFRINT testing algorithm is that the skip parameter is always set to 1, so as not to introduce artifacts. Also, the authors only refine the results using the ResNet model once every four CAIN iterations. In my case however, I found that the ideal number of iterations is 3, where the refinement step only takes place during the last iteration. In Figure 7.10 the results after 3 iterations with and without the refinement step are compared.

a)

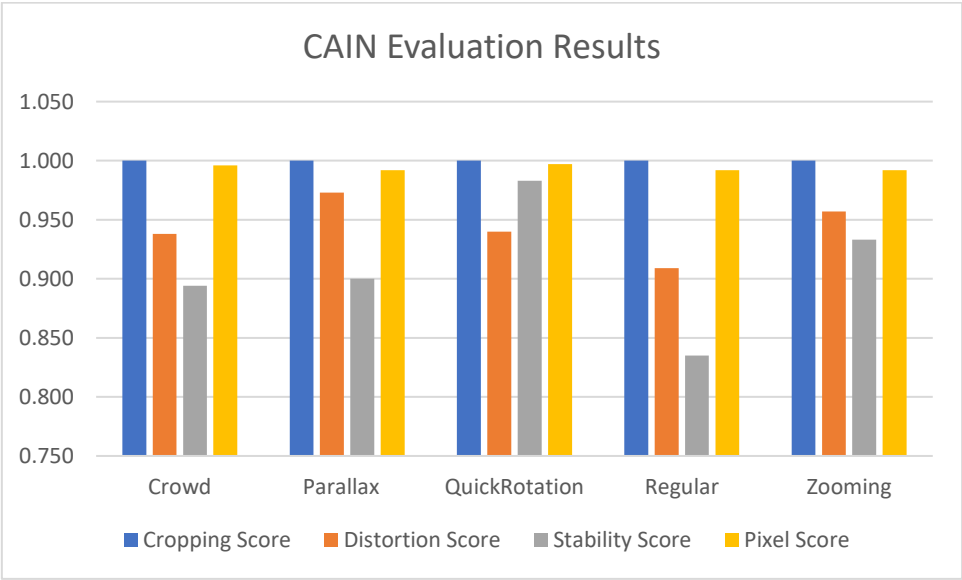
b)

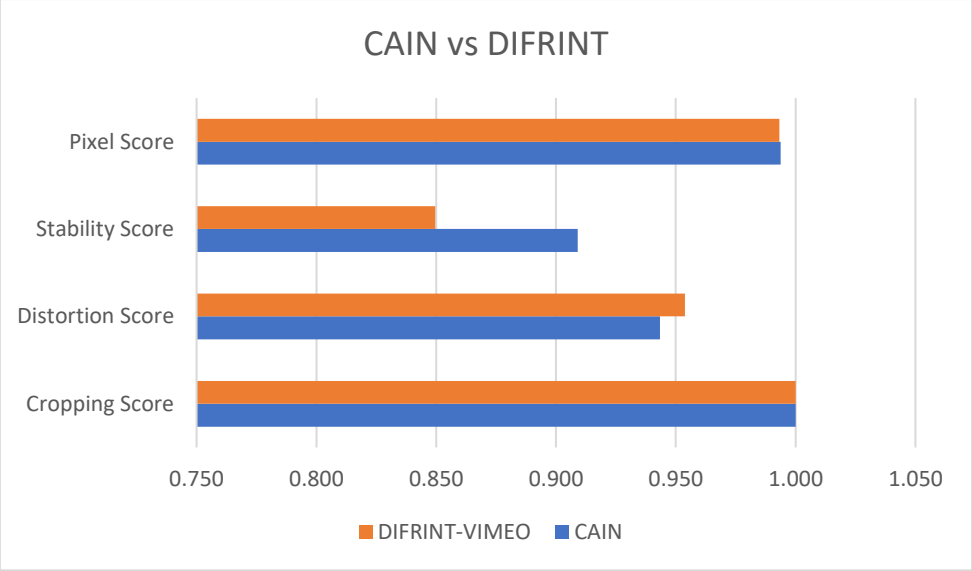
Figure 7.10: In the upper row a resulting interpolated sequence is illustrated, without the use of the refinement ResNet. It is apparent that through iterative interpolation there is blur accumulation. In the lower row the result is refined through our ResNet network, by reintroducing lost information from neighboring original unsteady frames.

7.2.3 Evaluation Results

CAIN

Category	Cropping Score	Distortion Score	Stability Score	Pixel Score
Crowd	1.000	0.938	0.894	0.996
Parallax	1.000	0.973	0.900	0.992
QuickRotation	1.000	0.940	0.983	0.997
Regular	1.000	0.909	0.835	0.992
Zooming	1.000	0.957	0.933	0.992





The capabilities of the CAIN network can be better understood by reviewing its comparative results to DIFRINT. It largely outscores DIFRINT in the stability score, which is the most important. The success of CAIN can be attributed to the use of PixelShuffle which allows it to retrain a large receptive field, without any loss of information, combined with its ingenious use of channel attention. A visual comparison of the generated results is shown in Figure 5.12.

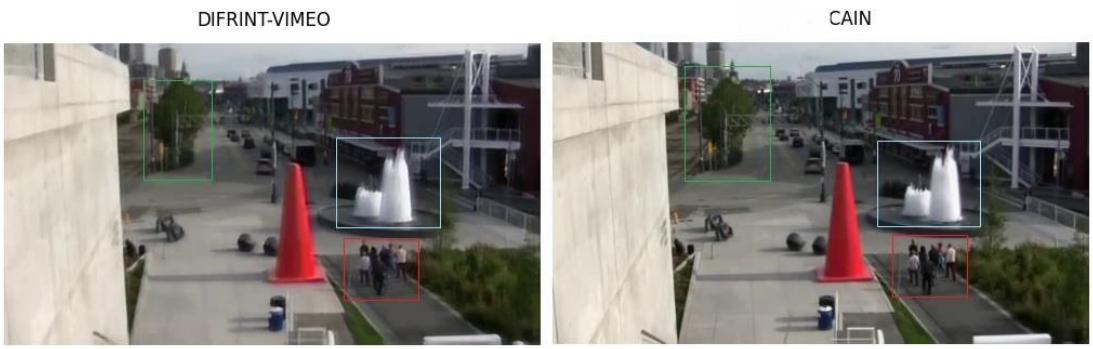


Figure 7.11: Visual comparison of the results of the two models. The results of CAIN show almost no distortion as opposed to the highlighted regions of DIFRINT's result.

While DIFRINT needs to be run for five iterations, the results obtained with CAIN used only 3 iterations. This resulted in an inference time of 4 minutes to stabilize a 40 second video with spatial dimensions 360×640 as opposed to DIFRINT's 6 minutes.

I provide the training code and testing scheme for this algorithm in this repository https://github.com/btxviny/Video-Stabilization-through-Frame-Interpolation-using-CAIN.

Chapter Summary

Using frame interpolation techniques for video stabilization purposes can produce very satisfactory results in most cases. It mitigates the presence of high frequency jerkiness extremely well. However, the algorithms are run iteratively which means long inference times. For example, to stabilize a thirty second video of 360x640 resolution using 3 iterations, approximately six minutes are needed. This limits the applications to be strictly post processing. Lastly, the interpolated results suffer from inevitable distortion artifacts around moving objects. This does not satisfy the basic saliency constraint of an ideal stable video, making the approach not suitable for a variety of videos.

Chapter 8: Synthetic Dataset Generation

Introduction

The availability of training data plagues all aspects of deep learning. Labeled data can help in implementing more straightforward training schemes and demystify various problem domains. For example, in 6.3 we were able to simplify the task of deep learning video stabilization, through the use of an appropriate supervised dataset. However, obtaining such training data is often difficult and time consuming. For the case of video stabilization, we have to record a pair stable/unstable videos which is not as straightforward as it may seem. Previous works, as we already mentioned in chapter 4, achieve this by capturing the video pairs through specialized gear. This device consists of two cameras. The first one is fixed to a stabilizer, producing a video with motion similar to that of the holder of the stabilizer. The second one, however, is attached to a moving platform in the stabilizer, producing stable videos. The two datasets captured in this manner DeepStab [31] and [42], include a combined number of 105 video pairs which is a relatively small number. Despite the careful design of the gear used, the resulting pairs still suffer from perspective mismatch, which is not ideal and can confuse neural network training. Another issue is that the transformations between stable and unstable videos are not provided. This prohibits their straightforward use for transformation learning. In 6.1 we saw how StabNet tried to overcome this issue through complicated loss functions. In this chapter we will go over synthetic dataset generation techniques that aim to mitigate the issues mentioned above, by generating video pairs with no perspective mismatch and provide the corresponding transformations that relate stable to unstable frames. We already reviewed, how 6.3 generates the stable video counterparts through deep learning frame interpolation, however all proposed techniques in this chapter, address the issue by generating unstable videos from already stable ones. These methods include random affine transformation generation and transformation sampling from preexisting motion datasets. I also propose a DNN based approach that is trained to produce realistic noise to perturb stable videos. Finally, I propose a generative method based on principal component analysis. After introducing all methods, we will evaluate them using a deep learning-based metric.

8.1 Random Affine Transformations

The authors of [23] propose creating a synthetic unstable video dataset to address the lack of motion pattern variety and color variation of pre-existing datasets at the time of writing. To produce an unstable video, they propose perturbing its stable counterpart with a random 2D affine transformation. The parameters of this transformation are sampled from uniform distributions with empirically determined ranges. Specifically, the translational components t_x, t_y are sampled from a uniform distribution $U[-5\%, 5\%]$ with respect to the frame's width and height. The horizontal and vertical scale components s_x, s_y are sampled from the uniform distribution $U[0.9,1.1]$. Finally, the rotational and shear parameters θ_r, θ_{sh} are sampled from $U[-5°, 5°]$. The individual transformation matrices are:

Translation: $\boldsymbol{T_r} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$,

Scale: $\boldsymbol{S} = \begin{bmatrix} s_x & 0 & 0 \\ s_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$,

Rotation: $\boldsymbol{R} = \begin{bmatrix} cos(\theta_r) & -sin(\theta_r) & 0 \\ sin(\theta_r) & cos(\theta_r) & 0 \\ 0 & 0 & 1 \end{bmatrix}$,

Shear: $\boldsymbol{S_h} = \begin{bmatrix} 1 & tan(\theta_{sh}) & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

The combined transformation is obtained by:

$$\boldsymbol{T} = \boldsymbol{T_r} \times \boldsymbol{R} \times \boldsymbol{S_h} \times \mathbf{S}$$

However, we are not done yet, because the convention in computer graphics and computer vision is for the origin of the coordinate system to be the top left element. This is not practical when applying transformations, so we have to shift the origin to the center of the image, apply the transformation and then shift the origin back to its original position. The forward and backward shift are achieved using the following translational matrices:

$$\boldsymbol{T1} = \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{bmatrix}, \qquad \boldsymbol{T2} = \begin{bmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

$\boldsymbol{T1}$ shifts the origin to the center where c_x, c_y are half the width and height dimensions respectively. $\boldsymbol{T2}$ shifts the origin back to the top left corner. So, the final transformation is calculated as: $\boldsymbol{T'} = \boldsymbol{T2} \times \boldsymbol{T} \times \boldsymbol{T}$

Using this method, we can collect professionally captured stable videos online, that contain various scene layouts and lighting conditions to expand on pre-existing

datasets. This process is obviously less time consuming than capturing new videos and can also be automated.

The results generated using this algorithm cannot fully represent the intricate noise patterns in real-world captured unstable videos. For example, when the person capturing a video is walking in a straight line there is much less unwanted motion as opposed to when they are walking down a flight of stairs. This method, however, samples the noise for every frame segment from the same distribution, essentially creating frames with equal probabilities of being noisy.

8.2 Sampling Transformations from Noise Dataset

[83] proposes creating a transformations dataset, obtained by fitting a 2D affine transformation that transforms every stable frame in DeepStab to its unstable counterpart. Then, to perturb an N frame stable video we sample N transformations from the dataset we created and apply them to the stable frames.

To calculate the transformations from stable to unstable frames, we fit an affine transformation to the matched features between the frame pair. As a way to ensure robustness I detect features on a 2 by 2 sub-image level and exclude outliers by fitting a homography using RANSAC. From the estimated matrices I extract the translational components and scale them with respect to the image width and height. I also extract the rotational/shear angular component which requires no scaling.

Directly applying these transformations to a frame sequence, results in a very unrealistic unstable video, as the motions can be very abrupt. [83] addresses this issue by passing the transformations through an exponential weighted moving average filter, thus making them less wobbly and more natural. An important final step is to crop both the generated and stable videos to exclude the inevitable black borders that are introduced.

8.3 Unstable Video Synthesis with Supervised Transformation Learning

The key idea is training a neural network that infers realistic unwanted motion noise to destabilize a video, thus enabling us to create stable/unstable video pairs to create a supervised dataset. The first step is acquiring a training dataset. For this purpose, I utilize DeepStab Modded [33] and extract the stable and unstable video trajectories. To extract the trajectories, I use the algorithm described in 6.2 but now we detect matched features between consecutive frames. From all inter-frame transformations, we extract the horizontal, vertical, and angular components and through their cumulative sum we estimate three trajectories. We repeat this process for all stable and unstable videos independently, creating a dataset of stable and corresponding unstable trajectories.

At this point I tried training a network in a supervised manner. Given an input stable trajectory it should infer its unstable version that should be as close as possible to the ground truth unstable trajectory. The input to this network is the three trajectories concatenated resulting in a shape of [batch size, 3, video length]. Using concatenated input instead of three individual inputs was chosen to enable the network to learn the correlation between the three trajectories.

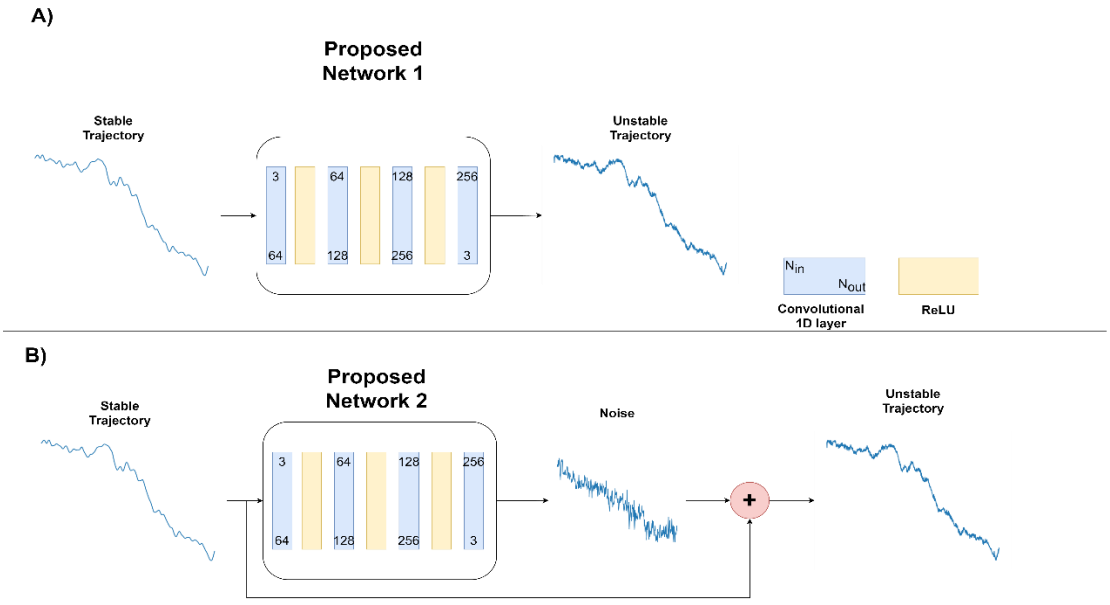


Figure 8.1: Deep Learning based noise generation

The ideal case would be a network, that could take in trajectories of arbitrary length and directly predict their generated unstable counterpart. For this reason, I chose a network based on 1D convolutional layers, as shown in Figure 8.1.a. For the loss function I initially chose the eyclidean distance between the generated and ground truth unstable trajectories. Using this metric, the network was not able to converge. I then changed the role of the model to the one shown in Figure 8.1.b. Instead of

directly predicting an unstable trajectory, now it just predicts the noise necessary to destabilize an input trajectory.

I proceeded by testing different network configurations, with a larger number of hidden channels and embedded LSTM layers [84], with no improvement. I then tried using a different metric, by replacing the strict eyclidean distance which forces the generated result to essentially be the same as the ground truth, with cosine distance. This metric encourages the output to be similar to the ground truth but not necessarily the same. I further introduced a frequency-based loss inspired by the stability metric we used for method evaluation throughout this thesis. This loss is computed exactly as the stability metric and penalizes the energy percentage of low frequencies, thus encouraging the network to generate high frequency content. Using this combination of loss functions the training converged to a value; however, the generated unstable video results were far from satisfactory. Some frame segments contained huge amounts of unwanted motion while others were unaltered.

8.4 Transformation Learning with Adversarial Training

Since the conventional supervised approach did not yield any promising results, I resorted to implementing Generative Adversarial training scheme. Our task perfectly aligns with the capabilities of GANs, as we want to learn the intricate patterns of unwanted motion and generate realistic new samples, that could have been drawn from the original dataset.

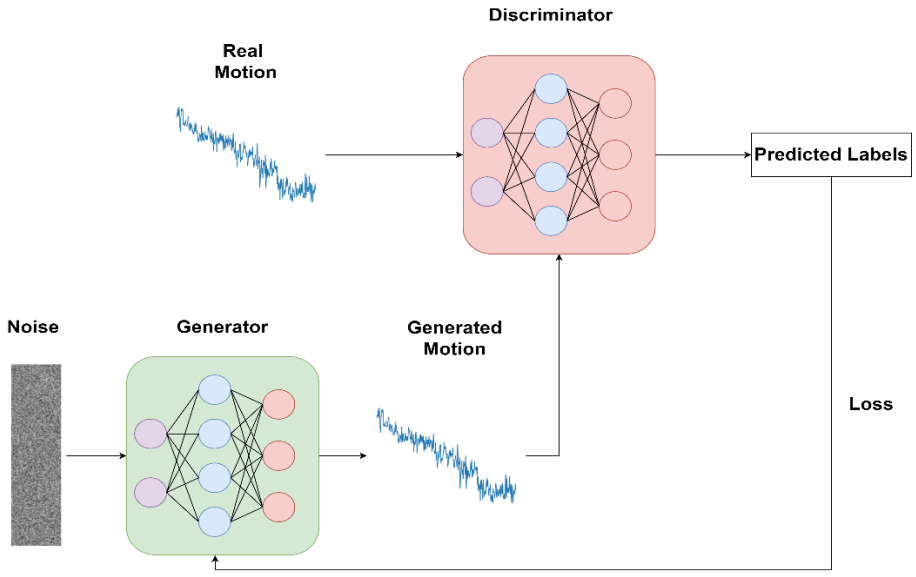


Figure 8.2: Proposed Training Scheme

Both the generator and discriminator are comprised of fully connected layers. The generator consists of 3 linear layers with $90 \times 3 : 2048$, $2048: 1024$, $1024: 90 \times 3$ nodes and each one is followed by a ReLU activation. The discriminator follows a similar architecture, but the last layer has $1024: 1$ nodes.

Using fully connected layers requires the length of the inputs to be fixed. I chose a length of 90 frames, which translates to a 3 second video segment with the frames per second conventionally set to 30. The transformations for the whole video during testing will be inferred using a sliding window technique. The real motion vectors are provided from the dataset we created in 6.2, and since they are already scaled, we do not apply any further pre-processing. The training loss functions for the discriminator and generator respectively are:

$$\mathcal{L}_{\mathcal{D}} = -\frac{1}{N}\sum_{i=1}^{N}\Big[y_i \log\big(D(x_i)\big) + (1-y_i)\log\Big(1 - D\big(G(z_i)\big)\Big)\Big],$$

$$\mathcal{L}_{\mathcal{G}} = -\frac{1}{N}\sum_{i=1}^{N}\log\Big(D\big(G(z_i)\big)\Big)$$

Where:

- N is the number of samples in the batch.
- x_i is a real sample from the dataset.
- z_i is the random noise vector $z \sim \mathcal{N}(0,1)^{1000}$, used as input to the generator.
- y_i is the label indicating whether the sample is real (1) or fake (0).

Using this training scheme, I trained the models for 50 epochs before they reached an equilibrium. The optimizer was ADAM with a learning rate of 1e-4. While the resulting videos created with the generators inferred motion vectors were realistic, implemented a second training stage to further improve their quality.

During this second training stage I discarded the discriminator as it has served its purpose and utilized the contrastive motion loss function explained in 6.3.3. I sampled 90 frame segments from the DeepStab stable videos and warp each one with the inferred transformations. I also sample their corresponding unstable counterparts. We now have three video segments, the input stable segment S_t, the ground truth unstable U_t and the generated unstable segment $\widehat{U_t}$. In contrast with the contrastive motion loss implementation in 6.3.3 we now want to make the generated sequences to resemble the unstable ones instead of the stable. For this reason, the anchor embeddings A are obtained by passing U_t through the R(2+1)D_18 video encoder, while the positive and negative are obtained from $\widehat{U_t}$ and S_t respectively. The loss is defined as:

$$\mathcal{L}_{cml} = max(d(A,P) - d(A,N) + \alpha, 0)$$

Where $d(x,y) = \|x - y\|_2$ is the Euclidean distance of the embeddings and $\alpha = 1$. To load three 90-frame sequences onto memory is demanding, so I converted every image to grayscale reducing the memory requirements by two thirds. However, we must now make an adjustment to the first conv3D layer of the video encoder. The initial weight of this layer in R(2+1)D_18 is [64, 3, 3, 7, 7] where the second dimension

corresponds to the three RGB channels. I compute the mean across this dimension resulting in [64, 1, 3, 7, 7] which is now compatible with our input.

I further trained the generator using this loss function for approximately 30.000 iterations with a reduced learning rate of 1-e5. The inference algorithm using the trained generator is shown in Figure 8.3.

Algorithm 1 Unsteady Video Synthesis

Description: The trained generator infers the transformation parameters to destabilize a `H*W` video of length `num_frames`, using a `sequence_length` sliding window.

```
sequence_length = 90
transforms = zeros(num_frames, 3)
for each idx in range(0, num_frames, 90) do
    Sample noise from normal distribution: noise ~ N(0,1)^1000
    transforms = generator(noise)
    transforms[:,0] *= W
    transforms[:,1] *= H
    if idx > num_frames − sequence_length then
      transforms[idx : idx + (num_frames − sequence_length), :]  =
      transforms[:(num_frames - idx), :]
    else
      transforms[idx : idx + sequence_length, :]  =
      transforms[:(num_frames - idx), :]
    end if
end for
unstable_frames = zeros_like(frames)
for each idx in range(num_frames) do
    frame = frames[idx,...]
    transformation = transformations[idx,...]
    unstable_frames[idx,...]  = warp(frame,transformation)
end for
center_crop(frames)
center_crop(unstable_frames)
Save(frames,unstable_frames)
```

Figure 8.3: Inference Algorithm for Unstable Video Synthesis.

8.5 Unwanted motion generation through PCA

Despite the recent success of deep learning in various tasks and fields, it does not always offer the best or more practical solution. In the previous section I trained a model to infer stable to unstable transformations, through trial and error and cumbersome training schemes. In this section we will use principal component analysis, a statistical procedure which is tailored for extracting the underlying patterns in data.

Specifically in our case, I use the same transformations dataset used in 6.4 but I split each video into 120 frame segments, representing 4-second temporal regions. Each one of these samples has a shape of [120,3] with the last dimension containing t_x, t_y horizontal and vertical transitional components and θ. Since these variables are correlated, I unroll each sample to the shape [360]. The entire segmented dataset has 36159 samples each with 360 variables. Through principal component analysis I calculate 10 principal components of the unwanted motion / stability noise.

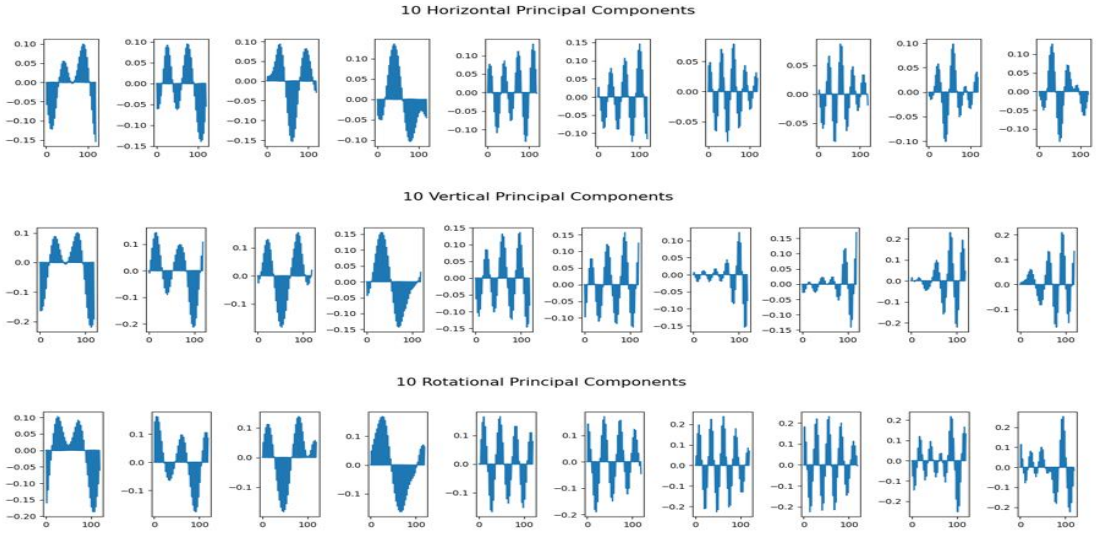


Figure 8.4: First ten translational and rotational principal components

The results are visualized in Figure 8.4, where the translational components X_c, Y_c are expressed in frame width and height percentage respectively, while the rotational components R_c are expressed in radians. Using these components, we will create synthetic 120-frame segments of realistic transformations. Each synthetic sample will be a random linear combination of the first ten principal components. For this purpose, we select 10 random weights $w = [w_1, w_2, \dots, w_{10}]$ with the property $\sum_{i=1}^{10} w_i = 1$ and express a new motion sample as:

$$x' = w^T \cdot \mathrm{X_c}$$

$$y' = w^T \cdot Y_c$$

$$\theta' = w^T \cdot R_c$$

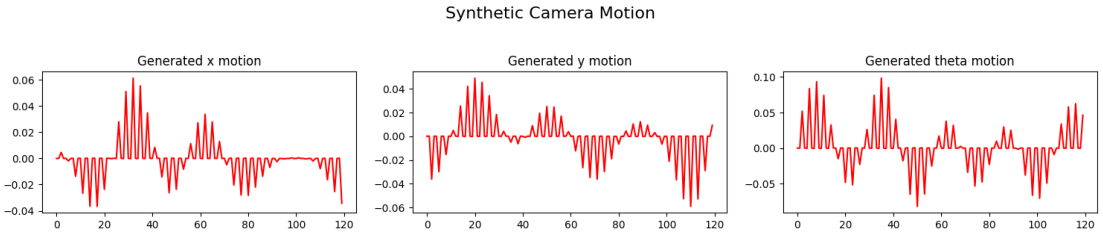
The proposed algorithm for generating realistic camera motion is shown in Figure 8.5.a. A generated sample using the random weights w = [0.057, 0.103, 0.05, 0.163, 0.064, 0.166, 0.281, 0.024, 0.086, 0.007] is shown in Figure 8.5.b.

Algorithm 1 Unsteady Video Synthesis

Description: Generates realistic camera motion using principal components

```
transforms = zeros(num_frames, 3)
sequence_length = 120
for idx in range(0, num_frames, sequence_length) do
    random_weights ~ Dirichlet(1,10)
    x' = random_weights · x_principal_components
    y' = random_weights · y_principal_components
    theta' = random_weights · theta_principal_components
    motion = stack([x', y', theta'],axis=1)
    motion[:,0] *= W
    motion[:,1] *= H
    if idx > num_frames − sequence_length then
        transforms[idx : idx + (num_frames − sequence_length), :] =
        motion[:(num_frames - idx), :]
    else
        transforms[idx : idx + sequence_length, :] =
        motion[:(num_frames - idx), :]
    end if
end for
unstable_frames = zeros_like(frames)
for idx in range(num_frames) do
    frame = frames[idx,...]
    transformation = transformations[idx,...]
    unstable_frames[idx,...] = warp(frame, transformation)
end for
center_crop(unstable_frames)
center_crop(stable_frames)
save(frames, unstable_frames)
```

a)



b)

Figure 8.5: a) Proposed motion algorithm. b) Generated sample.

8.6 Method Evaluation

We introduced four methods of generating realistic camera motion, in order to create an unstable video from its stable counterpart. Comparing these methods is not so straightforward. We could devise a metric which is the inverse of the stability metric we used to evaluate video stabilization methods, which returns the percentage of high frequency energy in the resulting unstable videos. This is not really appropriate here, since we don't just want a shaky video but a realistically unstable video. For this reason, I propose using the deep video encoder and contrastive motion loss exactly as in 6.4.

From [42] dataset I utilize 10 pairs of stable and unstable video pairs, belonging to different types of motion and scene layouts: walking, running, climbing, driving, dark. I generate a synthetic unstable video for every stable video in the evaluation dataset, using all 4 methods: random transformations, sampling transformation, using the generator model, and using the extracted principal components. We now have triplets of videos: stable, ground truth unstable and generated unstable. Similar to the loss described in 6.4, I feed the videos to the R(2D+1) encoder in 90-frame segments. The unstable, generated unstable and stable sequences generate the Anchor **A**, Positive **P** and negative **N** embeddings respectively. After normalizing the embeddings, I compute the loss using the formula we used previously. The losses for all 90-frame segments of the video are averaged to compute the video loss. Then we average all video losses to compute each methods loss. The results for each method are shown in Figure 8.6.

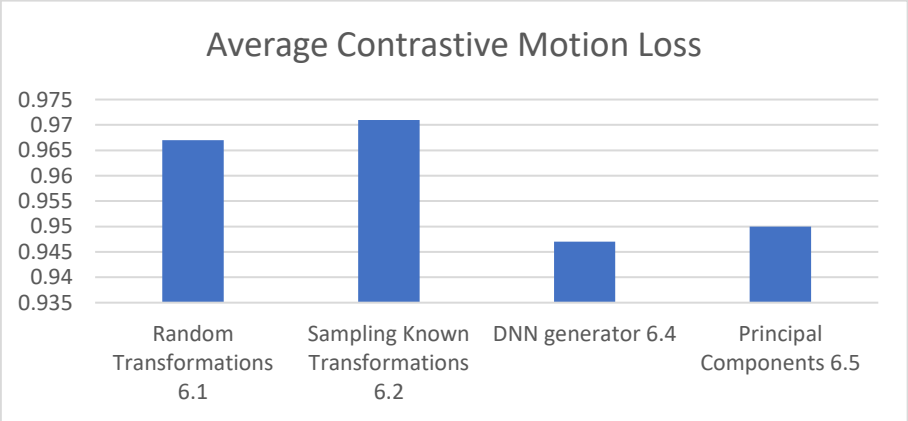


Figure 8.6: Average contrastive motion loss of dataset generation methods

While this metric is far from robust, the results agree with the viewing experience of the generated videos using each method.

Chapter Summary

We reviewed and implemented four different techniques for unstable video generation. Either of these methods can be used to expand on existing datasets. A major advantage of these datasets is that they include the exact transformations needed to stabilize the unstable videos. This can enable simple and effective transformation learning algorithms. I provide a comprehensive implementation of all aforementioned algorithms in this repository https://github.com/btxviny/Synthetic-Dataset-Generation-for-Video-Stabilization.

Chapter 9: Conclusions, Challenges & Future Directions

Through the course of this thesis, we delved into the key concepts behind digital video stabilization. Firstly, I introduced traditional, non-deep learning methods to grasp the core steps behind a video stabilization algorithm. Then through the integration of deep learning, we saw how these methods can be improved, while at the same time introducing unique deep learning approaches. Despite the wide range of approaches, there is still not an approach that performs well across different kinds of video scenes and camera motions, while at the same time achieving good inference speeds.

9.1 Conclusions

In Figure 9.1 I provide the evaluation scores for all the deep learning video stabilization methods I implemented along with some non-deep learning approaches. The main metric we are interested in is the stability score. By observing the results, we can see that the deep learning methods, excluding deep frame interpolation, do not outperform the traditional approaches. These methods optimize an objective function for a specific video input. On the other hand, deep learning methods aim to generalize well on a variety of video scenes and camera motions.

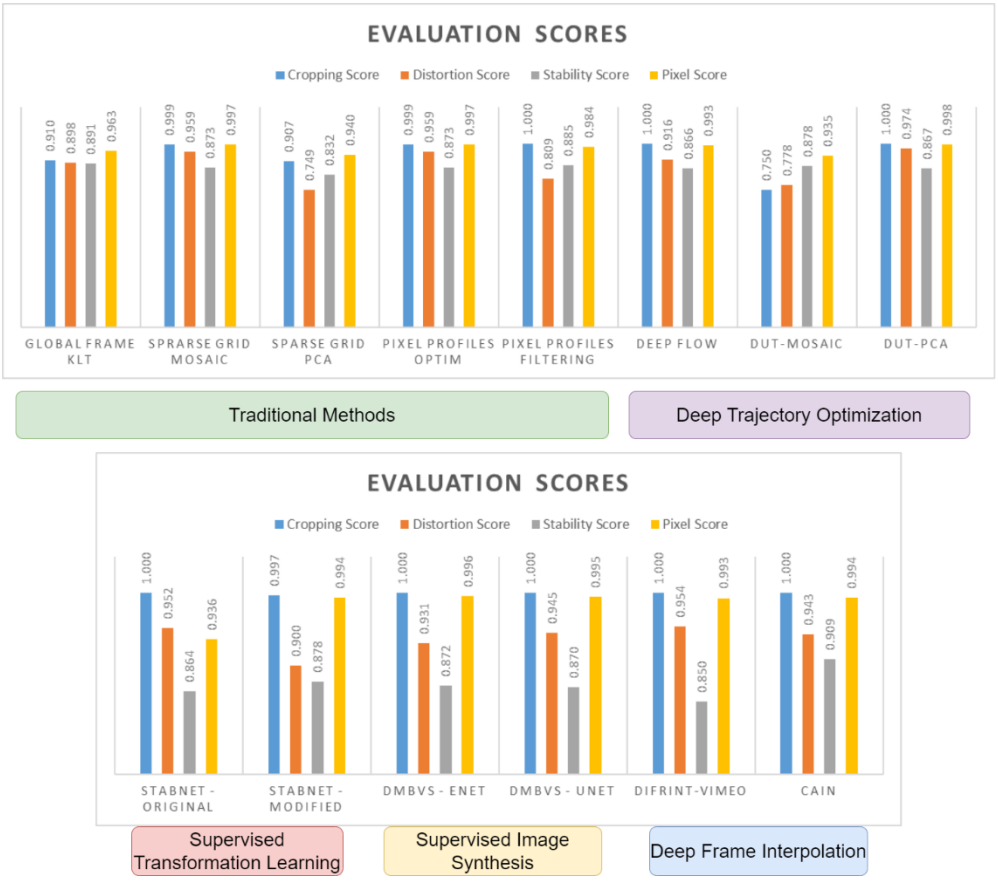


Figure 9.1: Evaluation scores for all introduced methods.

We cannot declare a clear 'winner' as the choice of method of choice strictly depends on the application, we are interested in. If we require a post-processing method that runs fast and generates pleasing results, a non-deep learning approach, such as pixel profiles filtering, would be the most appropriate choice. When time is not an issue using the CAIN network can yield the best results both in terms of stability and cropping. If our application needs a strictly online approach, for example a drone video stream, then StabNet-Original is the appropriate method as it only requires past frames as conditional input. When we are able to use future frames but still require real-time inference speeds, we can use DMBVS-UNet, as it only requires a one second frame buffer. So, in conclusion, the deep learning video methods do not necessarily achieve greater results but expand the range of applications for digital video stabilization.

9.2 Challenges

Lack of Widespread Evaluation Metrics

The first main challenge is the lack of widely accepted evaluation metrics. To this day there is no large scale full-reference dataset containing a large variety of motions and video scenes. For this reason, most researchers use non-reference evaluation to compare video stabilization algorithms. Another issue is that these metrics only consider cropping, distortion and stability and do not reflect the perceptive aspects of video stabilization. The stability score is computed on camera paths constructed using global frame transformations. Due to this fact it is not always representative of the quality of the generated results. Many times, the perceptual stability of a video and its stability score do not coincide.

Lack of Supervised Training Data

The lack of full-reference datasets extends to a lack of supervised training datasets, an important issue which I covered in depth through this thesis. Without appropriate training data, the capabilities of deep learning approaches are limited.

Variety of Camera Motion and Scene Layouts

Issues like missing pixels, occlusion, and parallax in videos are not rare edge cases; instead, they are common occurrences in everyday videos, posing substantial challenges for effective video stabilization. Also, dynamically moving objects and depth variation can prove very challenging, both in the motion estimation and compensation stages. An effective stabilization algorithm must achieve a well-balanced tradeoff between stability and visual distortion to handle the cases mentioned above. Despite the existence of tailored deep learning stabilization approaches designed for specific video scenes, such as selfie videos or UAV footage,

the lack of a universally applicable method persists due to differing stabilization requirements across various scenes.

Long Inference Times

A major issue that plagues many deep learning video stabilization algorithms are long inference times. In trajectory optimization methods the motion estimation stage can take a long time as it requires feature extraction and tracking or dense optical flow estimation. Though these procedures can be implemented using deep learning, they still introduce non-negligible delays. As for frame interpolation methods, they generate better results when run in an iterative manner, this fact restricst their applications to post-processing.

Effectiveness of OIS

In addition to the technical challenges discussed the remarkable performance of Optical Image Stabilization (OIS) mechanisms has made digital video stabilization seem redundant. These mechanisms, employed in many contemporary smartphones, utilize a sophisticated coil-based mechanism where the camera sensor dynamically adjusts its position based on readings from gyroscopes and accelerometers. This ingenious design effectively counteracts camera motion, providing impressive stabilization results. Historically, OIS mechanisms were confined to high-end, expensive handheld devices. However, in recent years, they have become increasingly prevalent in more affordable smartphones, surpassing the performance of digital video stabilization methods. The widespread availability of OIS in budget-friendly devices has, to some extent, dampened the enthusiasm for exploring alternative video stabilization approaches within the research community.

9.3 Future Directions

The effectiveness of video stabilization is a prominent research focus, with a dual emphasis on refining network architectures for extracting inter-frame motion features and exploring the fusion of conventional and learning-based methods. One significant challenge lies in the insufficient availability of datasets dedicated to perceptive Video Stabilization Quality Assessment (VSQA). While current Deep Learning (DL) methods address key aspects such as cropping ratio, distortion value, and stability score, they often lack metrics reflecting perceptive aspects of VS. The future of deep learning video stabilization may hinge on addressing these challenges through a fusion of machine learning approaches like deep learning with user reviews and feedback. Incorporating reinforcement learning techniques for subjective evaluation, rather than relying solely on geometric criteria, could lead to more perceptually accurate and efficient video stabilization solutions. As the field undergoes a deep revolution in computer vision, the integration of perceptual approaches for visual information

processing [85] is poised to enhance the efficacy of DL-based methods, paving the way for innovative solutions in the coming years.

References

[1] J. Shi and others, "Good features to track," in *1994 Proceedings of IEEE conference on computer vision and pattern recognition*, 1994.

[2] H. Bay, A. Ess, T. Tuytelaars and L. Van Gool, "Speeded-up robust features (SURF)," *Computer vision and image understanding,* vol. 110, p. 346–359, 2008.

[3] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision,* vol. 60, p. 91–110, 2004.

[4] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *IJCAI'81: 7th international joint conference on Artificial intelligence*, 1981.

[5] J. Cai and R. Walker, Robust motion estimation for camcorders mounted in mobile platforms, IEEE, 2008.

[6] G. Farnebäck, "Two-frame motion estimation based on polynomial expansion," in *Image Analysis: 13th Scandinavian Conference, SCIA 2003 Halmstad, Sweden, June 29–July 2, 2003 Proceedings 13*, 2003.

[7] C. Ballester, L. Garrido, V. Lazcano and V. Caselles, "A TV-L1 optical flow method with occlusion detection," in *Pattern Recognition: Joint 34th DAGM and 36th OAGM Symposium, Graz, Austria, August 28-31, 2012. Proceedings 34*, 2012.

[8] D. Sun, X. Yang, M.-Y. Liu and J. Kautz, "Pwc-net: Cnns for optical flow using pyramid, warping, and cost volume," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018.

[9] M. Grundmann, V. Kwatra and I. Essa, "Auto-directed video stabilization with robust l1 optimal camera paths," in *CVPR 2011*, 2011.

[10] S. Liu, P. Tan, L. Yuan, J. Sun and B. Zeng, "Meshflow: Minimum latency online video stabilization," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VI 14*, 2016.

[11] D. Sun, S. Roth and M. J. Black, "Secrets of optical flow estimation and their principles," in *2010 IEEE computer society conference on computer vision and pattern recognition*, 2010.

[12] S. Liu, L. Yuan, P. Tan and J. Sun, "Bundled camera paths for video stabilization," *ACM transactions on graphics (TOG),* vol. 32, p. 1–10, 2013.

[13] J. Wulff and M. J. Black, "Efficient sparse-to-dense optical flow estimation using a learned basis and layers," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[14] S. Liu, L. Yuan, P. Tan and J. Sun, "Steadyflow: Spatially smooth optical flow for video stabilization," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014.

[15] S. Liu, Y. Wang, L. Yuan, J. Bu, P. Tan and J. Sun, "Video stabilization with a depth camera," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

[16] C. Liu and others, "Beyond pixels: exploring new representations and applications for motion analysis," 2009.

[17] F. Liu, M. Gleicher, H. Jin and A. Agarwala, "Content-preserving warps for 3D video stabilization," in *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*, 2023, p. 631–639.

[18] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review,* vol. 65, p. 386, 1958.

[19] A. not specified, *Lecture 6e - Overview of mini-batch gradient descent,* Year not specified.

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980,* 2014.

[21] K. He, X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[22] O. Ronneberger, P. Fischer and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, 2015.

[23] J. Yu and R. Ramamoorthi, "Learning video stabilization using optical flow," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020.

[24] R. Ranftl, K. Lasinger, D. Hafner, K. Schindler and V. Koltun, "Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer," *IEEE transactions on pattern analysis and machine intelligence,* vol. 44, p. 1623–1637, 2020.

[25] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai and S. Chintala, *PyTorch: An Imperative Style, High-Performance Deep Learning Library,* 2019.

[26] Y. Xu, J. Zhang, S. J. Maybank and D. Tao, "Dut: Learning video stabilization by simply watching unstable videos," *IEEE Transactions on Image Processing,* vol. 31, p. 4306–4320, 2022.

[27] S. P. Lloyd, "Least squares quantization in PCM," *IEEE transactions on information theory,* vol. 28, p. 129–137, 1982.

[28] J. Yu and R. Ramamoorthi, "Robust video stabilization by optimization in cnn weight space," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019.

[29] Y.-C. Lee, K.-W. Tseng, Y.-T. Chen, C.-C. Chen, C.-S. Chen and Y.-P. Hung, "3d video stabilization with depth estimation by cnn-based optimization," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021.

[30] Z. Shi, F. Shi, W.-S. Lai, C.-K. Liang and Y. Liang, "Deep online fused video stabilization," in *Proceedings of the IEEE/CVF winter conference on applications of computer vision*, 2022.

[31] M. Wang, G.-Y. Yang, J.-K. Lin, S.-H. Zhang, A. Shamir, S.-P. Lu and S.-M. Hu, "Deep online video stabilization with multi-grid warping transformation learning," *IEEE Transactions on Image Processing,* vol. 28, p. 2283–2292, 2018.

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

[33] M. K. Ali, S. Yu and T. H. Kim, "Deep motion blind video stabilization," *arXiv preprint arXiv:2011.09697,* 2020.

[34] J. Yu, Z. Lin, J. Yang, X. Shen, X. Lu and T. S. Huang, "Free-form image inpainting with gated convolution," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019.

[35] S.-Z. Xu, J. Hu, M. Wang, T.-J. Mu and S.-M. Hu, "Deep video stabilization using adversarial networks," in *Computer Graphics Forum*, 2018.

[36] M. Jaderberg, K. Simonyan, A. Zisserman and others, "Spatial transformer networks," *Advances in neural information processing systems,* vol. 28, 2015.

[37] J. Yu, R. Ramamoorthi, K. Cheng, M. Sarkis and N. Bi, "Real-time selfie video stabilization," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021.

[38] Z. Zhang, Z. Liu, P. Tan, B. Zeng and S. Liu, "Minimum latency deep online video stabilization," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023.

[39] S. Niklaus and F. Liu, "Context-aware synthesis for video frame interpolation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018.

[40] J. Choi and I. S. Kweon, "DIFRINT: Deep Iterative Frame Interpolation for Full-Frame Video Stabilization," in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, 2019.

[41] M. Choi, H. Kim, B. Han, N. Xu and K. M. Lee, "Channel attention is all you need for video frame interpolation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020.

[42] L. Zhang, Q.-Z. Zheng, H.-K. Liu and H. Huang, "Full-reference stability assessment of digital video stabilization based on riemannian metric," *IEEE Transactions on Image Processing,* vol. 27, p. 6051–6063, 2018.

[43] T. Zhou, R. Tucker, J. Flynn, G. Fyffe and N. Snavely, "Stereo magnification: Learning view synthesis using multiplane images," *arXiv preprint arXiv:1805.09817,* 2018.

[44] A. Goldstein and R. Fattal, "Video stabilization using epipolar geometry," *ACM Transactions on Graphics (TOG),* vol. 31, p. 1–10, 2012.

[45] Y. J. Koh, C. Lee and C.-S. Kim, "Video stabilization based on feature trajectory augmentation and selection and robust mesh grid warping," *IEEE Transactions on Image Processing,* vol. 24, p. 5260–5273, 2015.

[46] K. Soomro, A. R. Zamir and M. Shah, "UCF101: A dataset of 101 human actions classes from videos in the wild," *arXiv preprint arXiv:1212.0402,* 2012.

[47] S. Su, M. Delbracio, J. Wang, G. Sapiro, W. Heidrich and O. Wang, "Deep video deblurring for hand-held cameras," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[48] W. Kay, J. Carreira, K. Simonyan, B. Zhang, C. Hillier, S. Vijayanarasimhan, F. Viola, T. Green, T. Back, P. Natsev and others, "The kinetics human action video dataset," *arXiv preprint arXiv:1705.06950,* 2017.

[49] J. Pont-Tuset, F. Perazzi, S. Caelles, P. Arbeláez, A. Sorkine-Hornung and L. V. Gool, "The 2017 DAVIS Challenge on Video Object Segmentation," *arXiv:1704.00675,* 2017.

[50] T. Xue, B. Chen, J. Wu, D. Wei and W. T. Freeman, "Video enhancement with task-oriented flow," *International Journal of Computer Vision,* vol. 127, p. 1106–1125, 2019.

[51] R. C. Streijl, S. Winkler and D. S. Hands, "Mean opinion score (MOS) revisited: methods and applications, limitations and alternatives," *Multimedia Systems,* vol. 22, p. 213–227, 2016.

[52] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation,* vol. 19, p. 297–301, 1965.

[53] E. Fix and J. L. Hodges, "Discriminatory analysis. Nonparametric discrimination: Consistency properties," *International Statistical Review/Revue Internationale de Statistique,* vol. 57, p. 238–247, 1989.

[54] E. Ilg, N. Mayer, T. Saikia, M. Keuper, A. Dosovitskiy and T. Brox, "Flownet 2.0: Evolution of optical flow estimation with deep networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[55] B. Zhou, H. Zhao, X. Puig, S. Fidler, A. Barriuso and A. Torralba, "Scene parsing through ade20k dataset," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[56] O. Özyeşil, V. Voroninski, R. Basri and A. Singer, "A survey of structure from motion*.," *Acta Numerica,* vol. 26, p. 305–364, 2017.

[57] X. Shen, C. Wang, X. Li, Z. Yu, J. Li, C. Wen, M. Cheng and Z. He, "RF-Net: An end-to-end image matching network based on receptive field," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019.

[58] R. Jonschkowski, A. Stone, J. T. Barron, A. Gordon, K. Konolige and A. Angelova, "What matters in unsupervised optical flow," in *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part II 16*, 2020.

[59] C. Liu, "Beyond pixels: exploring new representations and applications for motion analysis," Doctoral dissertation, Massachusetts Institute of Technology, 2009.

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

[61] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM,* vol. 24, p. 381–395, 1981.

[62] M. Fatica, "CUDA toolkit and libraries," in *2008 IEEE hot chips 20 symposium (HCS)*, 2008.

[63] Z. Teed and J. Deng, "Raft: Recurrent all-pairs field transforms for optical flow," in *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part II 16*, 2020.

[64] F. Albardi, H. D. Kabir, M. M. I. Bhuiyan, P. M. Kebria, A. Khosravi and S. Nahavandi, "A comprehensive study on torchvision pre-trained models for fine-grained inter-species classification," in *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2021.

[65] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems,* vol. 27, 2014.

[66] S. Kullback and R. A. Leibler, "On information and sufficiency," *The Annals of Mathematical Statistics,* vol. 22, p. 79–86, 1951.

[67] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784,* 2014.

[68] X. Mao, Q. Li, H. Xie, R. Y. K. Lau, Z. Wang and S. Paul Smolley, "Least squares generative adversarial networks," in *Proceedings of the IEEE international conference on computer vision*, 2017.

[69] M. Arjovsky, S. Chintala and L. Bottou, "Wasserstein generative adversarial networks," in *International conference on machine learning*, 2017.

[70] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin and A. C. Courville, "Improved training of wasserstein gans," *Advances in neural information processing systems,* vol. 30, 2017.

[71] J. L. Elman, "Learning and development in neural networks: The importance of starting small," *Cognition,* vol. 48, p. 71–99, 1993.

[72] H. Zhao, O. Gallo, I. Frosio and J. Kautz, "Loss functions for image restoration with neural networks," *IEEE Transactions on computational imaging,* vol. 3, p. 47–57, 2016.

[73] O. Kupyn, V. Budzan, M. Mykhailych, D. Mishkin and J. Matas, "Deblurgan: Blind motion deblurring using conditional adversarial networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018.

[74] G. Bradski, "OpenCV: Open Source Computer Vision Library," *Dr. Dobb's Journal of Software Tools,* vol. 25, p. 120–123, 2000.

[75] K. Hara, H. Kataoka and Y. Satoh, "Learning spatio-temporal features with 3d residual networks for action recognition," in *Proceedings of the IEEE international conference on computer vision workshops*, 2017.

[76] R. Collobert, K. Kavukcuoglu, C. Farabet and et al., "Torch7: A Matlab-like Environment for Machine Learning," *BigLearn, NIPS Workshop,* 2011.

[77] D. Tran, H. Wang, L. Torresani, J. Ray, Y. LeCun and M. Paluri, "A closer look at spatiotemporal convolutions for action recognition," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2018.

[78] C. Feichtenhofer, H. Fan, J. Malik and K. He, "Slowfast networks for video recognition," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019.

[79] R. Mechrez, I. Talmi and L. Zelnik-Manor, "The contextual loss for image transformation with non-aligned data," in *Proceedings of the European conference on computer vision (ECCV)*, 2018.

[80] S. Woo, J. Park, J.-Y. Lee and I. S. Kweon, "CBAM: Convolutional Block Attention Module," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

[81] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert and Z. Wang, "Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[82] I. Sobel and G. Feldman, "An Isotropic 3x3 Image Gradient Operator," *Machine vision for three-dimensional scenes,* p. 376–379, 1968.

[83] M. Ito, "Video Stabilisation Based on Spatial Transformer Networks," 2021.

[84] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation,* vol. 9, p. 1735–1780, 1997.

[85] A. Beghdadi, M.-C. Larabi, A. Bouzerdoum and K. M. Iftekharuddin, "A survey of perceptual image processing methods," *Signal Processing: Image Communication,* vol. 28, p. 811–831, 2013.