THE COOPER UNION
ALBERT NERKEN SCHOOL OF ENGINEERING

Improving Flood Maps by Increasing the Temporal Resolution of Satellites Using Hybrid Sensor Fusion - Video Interpolation Networks

by

Yuval Epstain Ofek

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Engineering

December 2021

Professor Sam Keene, Advisor

THE COOPER UNION FOR THE
ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

Dean, School of Engineering Date

Prof. Sam Keene, Thesis Advisor Date

Acknowledgments

To my family, for the endless love, support, and encouragement.

To my advisor, Sam Keene, for the guidance, inspiration, and support during this work and beyond.

To my friends, at Cooper and beforehand, for shaping me into who I am today and pushing me forward along my way.

To the team at the University of Arizona and Columbia, for opening their research to me and sharing their love of such fascinating topics.

To my professors and teachers, for the knowledge they imparted and the spirit they nurtured within me.

To my mentors, for urging me forward towards better things and showing me the way when I felt lost.

To my peers, for sharing their journey with me and continuing to challenge me to be a better version of myself.

To the hours spent together in lectures, classes, labs, and hallways.

To the meals shared, the jokes, and the positive atmosphere we built together.

To The Cooper Union, for the advancement of science and art.

I humbly say thanks to all of you. Words cannot fully encompass my gratitude. I could not have done it without you, nor would I wish to have done it any other way.

Abstract

Due to the inherent limitations of the satellites used to generate flood inundation maps, a key tool to mitigate damages and deal with floods, the creation of high-spatial-high-temporal resolution flood maps is limited. In recent years, there has been increasing amount of research using satellite sensor fusion to generate synthetic bands [1–3]. Even more recently, studies using video frame interpolation across a single/multiple bands have been used to augment inputs to remote sensing tools [4]. We introduce the synthesis of these two methods through proposing different pipeline architectures of varying complexities to half the revisit rate of Landsat 8 imagery, generating high-spatial-high-temporal imagery through merging insights from the MODIS satellite tools and past and future Landsat 8 imagery. Experiments on a custom dataset show that the combination of video frame interpolation and satellite sensor fusion outperforms standard methods across almost all metrics. Furthermore, experiments show that pipelines composed of as few as a single model perform comparably to the top performing pipelines.

Contents

List of Figures

1 Introduction

Floods are devastating. They are the most destructive natural disaster on our planet. Floods account for half of the world's weather-related disasters, force 26 million people into poverty every year, cause $7.5 billion in damages in the US alone, and lead to economic, urban, and commercial setbacks [5–7].

Understanding floods is the key to helping combat their effects, preventing them from occurring, and mitigating human, commercial, infrastructural, and economic losses. One of the tools best suited for tracking and monitoring floods are flood inundation maps. These maps help understand what areas were affected by a flood and have become critical in assessing and addressing flood risk [8].

Traditionally, mapping flood inundation was done by relying on single-sensor methods. These methods take either optical, radar, or microwave imagery on a daily, monthly, and yearly basis to produce a flood map. A description of the advantages and disadvantages of common one-sensor flood mapping types can be seen in Table 1 [8].

Sensor Type	Advantages	Obstacles	Example Sensors
Coarse resolution optical	High revisit frequency (twice daily)	Cannot map floods through clouds or under vegetation	MODIS
Medium resolution optical	Identify assets such as roads, fields, and major infrastructure	Cannot map floods through clouds or under vegetation, and 3 day revisit for <30m resolution	Landsat, Sentinel-2
Very high resolution commercial (optical or radar)	Identify floods in urban areas and assets including homes, cars, side streets	Cannot map floods through clouds or under vegetation, short history, and data requires purchase	Planetscope, Iceye, TerraSARX, UAVs
Radar (active)	Identify inundation through clouds, and assets such as roads, fields, and major infrastructure	Difficult to map flooded vegetation and water between tall buildings, short history, low revisit (>6 days))	Sentinel-1, ALOS PALSAR
Passive microwave	Identify temporal pattern of inundation through clouds over large areas	Very coarse resolution, degraded signal in urban areas	SMAP, SMOS, AMSR

Table 1: Common single-sensor flood-mapping categories with advantages/disadvantages. Table reproduced from the Tellman NASA THP proposal [8].

One-sensor approaches are inherently limited by the functionalities of the sensor used. Where one sensor exhibits high temporal resolution, its spatial resolution limits the resulting map, or vice versa. Limited spatio-temporal resolution is an inherent problem in remote sensing with satellites, as each sensor design is a trade-off between spatial and temporal resolutions. Specifically in flood events, which are short-lasting, lack of sufficient temporal resolution becomes a challenge in generating high-resolution maps. Low temporal resolution sensors are susceptible to completely missing flood events, or potentially misattributing multiple floods to a single event, as visualized in Figure 1. Given the need for sufficient temporal resolution to detect floods and the limited spatio-temporal resolution of satellites, we see that there is a limit to the spatial resolution of any sensor used to generate flood inundation maps,

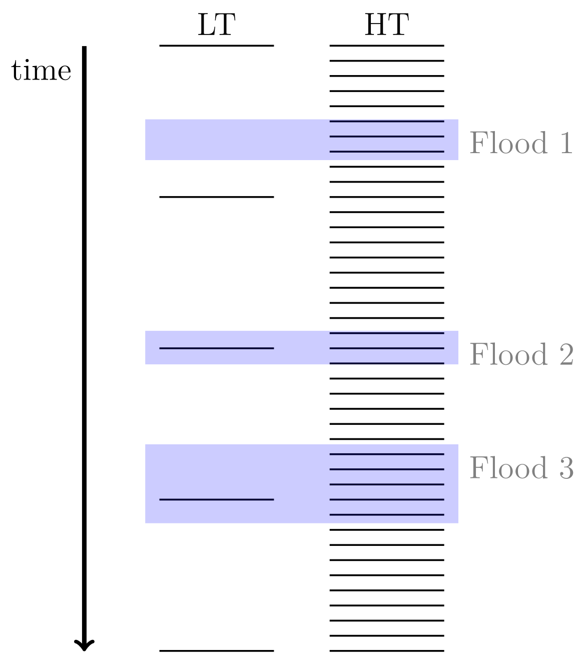


Figure 1: Issues with low temporal resolution during flood events. Low temporal resolution (LT) and high temporal resolution (HT) sensors during three flood events (blue). The LT sensor completely misses Flood 1 and combines Flood 2 and Flood 3 into one event.

which means that flood inundation maps generated by one-sensor techniques are limited in spatial resolution. In other words, one-sensor approaches can only generate flood maps up to a certain spatial resolution.

To combat these limitations, techniques combining multiple sensors have been introduced as means of improving the output flood mapping and are showing much promise [8]. Common approaches used for combining sensor inputs to the mapping system are image-to-image translation, often referred to as sensor-fusion or alternatively satellite-to-satellite translation in the remote sensing field, which allows a low spatial resolution image from an alternate sensor to be converted into the desired sensor's image. Due to the limitations of satellite sensors, this becomes the task of mapping a low-resolution image from one sensor to a high-resolution image of another. This is similar to super-resolution, the task of increasing the spatial resolution of an image, but with the additional

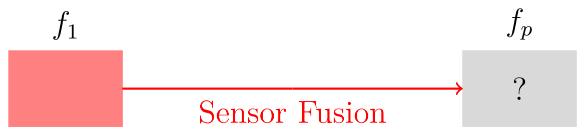


Figure 2: Example of satellite sensor fusion using input from satellite f_1 to predict image in gray for satellite f_p

challenge of switching the domains of the image as a consequence of translating between sensors. While generating images that are temporally consistent with real-life events, these methods are inherently deficient in generating realistic high-resolution imagery and are prone to producing artifacts, all due to a lack of high-resolution inputs.

Another approach is to use video frame interpolation techniques, a method that has only been introduced into the remote sensing field in the last year or so. These techniques rely on known future and past images from the high-resolution sensor and generate an image that 'best' explains how the image changed from the past to the future. An example of video frame interpolation can be seen in Figure 3. It is important to note that as these models rely on 'future' images they cannot be run in real-time and need to wait until the future images are captured, which may take weeks. While this delay is a problem for floods and real-time response to flood events, these techniques are still useful for prolonged recovery from floods or when applied to historical data for better flood-planning. These techniques have also shown promise in weather prediction [4]. While generating high-resolution images that appear incredibly accurate, these methods may fail to predict high temporal resolution events, such as floods, and the predictions are susceptible to not tracking real-life events as they lack high temporal information to make accurate predictions.

In this work, we have attempted to combine video frame interpolation and satellite sensor fusion as a means of generating high spatial resolution - high

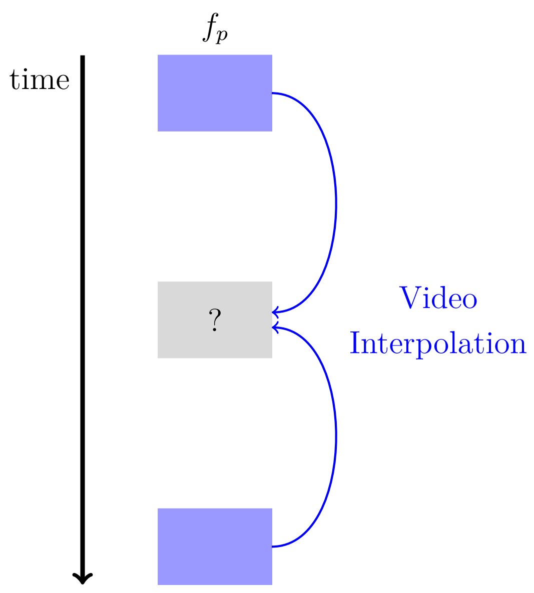


Figure 3: Example of video frame interpolation using inputs in blue to predict intermediate frame in gray

temporal (HSHT) resolution imagery. We visualize this combination in Figure 4. We experiment with providing imagery sensor-fusion methods with contextual spatial information to improve spatial accuracy as well as with providing video interpolation methods with information regarding short-term temporal events. This additional information enables the generation of high-spatial high-temporal imagery with greater accuracy, synthesizing useful sensor inputs for flood inundation mapping systems which lower the need for further synthesis of sensors and mitigate spatio-temporal resolution limitations of the final flood mappings.

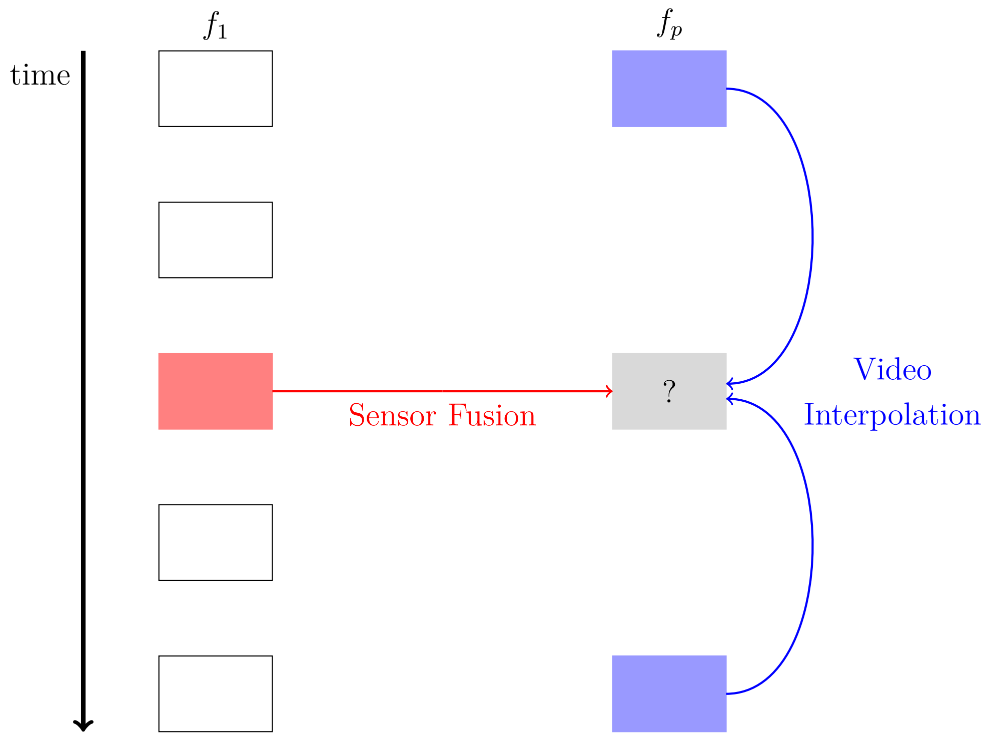


Figure 4: Visualization of satellite sensor fusion (red) and video interpolation (blue) for the same output image (gray).

2 Background

2.1 Remote Sensing

Remote sensing is the practice of using imagery of the Earth's surface to derive information. Such imagery is obtained through electromagnetic radiation in one or many spectral regions that is reflected or emitted from the surface. Besides flood inundation mapping, the field of remote sensing has a sizeable number of applications including, but not limited to, hydrology, urban studies, reconnaissance, surveillance, geology, and agriculture [9].

2.1.1 A Brief History

The field of remote sensing could be said to have begun in 1858, when the first aerial balloon photograph was taken by French photographer Gaspar Félix

Tournachon, known also as 'Nadar'. Said photographs have been lost to time and the earliest surviving photographs are those taken by James Wallace Black on October 13, 1860 over Boston [10]. One such photo can be seen in Figure 5.

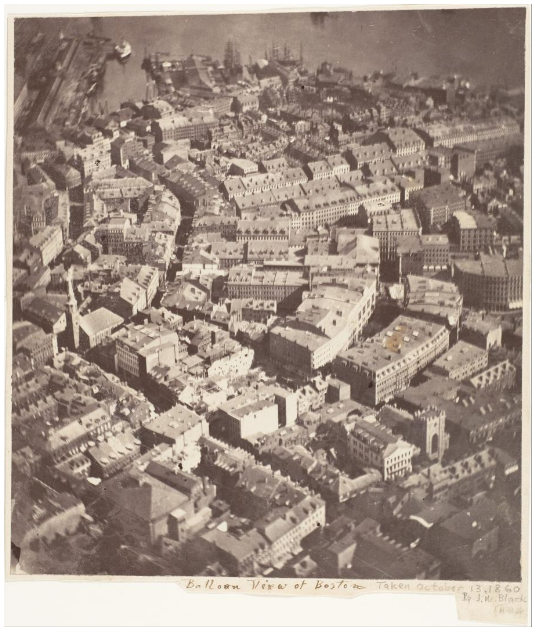


Figure 5: Earliest Surviving Aerial Photo, taken by James Wallace Black in October 13, 1860 [11]

Photography from powered aircrafts was introduced in 1909 when Wilbur Wright flew to photograph a military field at Centocelli, Italy [9, 10]. Aerial photography remained more of an artistic curiosity at the point, lacking a scientific and systematic approach. This all changed with the break of World War I. With the use of aerial photography for reconnaissance, the importance of the field became apparent, spurring two decades of rapid development following the war. New tools and techniques allowed the expansion of the field to include land surveys, geologic mappings, and rural economic monitoring, helping the government monitor the U.S. during the worldwide economic depression.

World War II marks another milestone in the development of the field. The electromagnetic spectrum captured using aerial tools expanded from almost exclusively the visible spectrum to include the infrared and microwave regions. With wartime training also came many experienced pilots and camera operators ready to apply their skills to aerial photography and remote sensing, expanding both the acceptance of remote sensing and developing its uses.

Satellites were introduced into remote sensing at around 1960, during the Cold War. Some were used for strategic reconnaissance, while others were for civil purposes, such as climatology and meteorology. This served as the first use of previously classified military remote sensing instruments for civilian use.

The term *remote sensing* was coined in 1962 following a need to broaden from original 'aerial photography', which did not quite capture the many forms of imagery collected using radiation.

Landsat 1's launch in 1972 was the beginning of the era of using Earth-orbiting satellites to observe Earth's land areas. Landsat 1 provided routine data in a digital format, allowing a wider audience and setting the stage for computerized analysis of imagery.

Following contributions to the field include hyperspectral sensors, which enabled the collection of over 200 spectral regions and the standardization of data using the Global Information System (GIS) and Global Positioning System (GPS).

Towards the end of the 20th century, more countries began to launch satellites for remote sensing. Data became more available and new tools to interact with this data were introduced. One notable tool built for the wide audience available on the World Wide Web, is Google Earth. Google Earth serves as a virtual representation of the Earth's surface created using a composition of varied digital images [9, 12].

2.1.2 Electromagnetic Waves

Electromagnetic radiation is emitted by all objects (except for those at absolute zero temperature) [9]. As one example, what we humans see is electromagnetic radiation, which is a small part of all the electromagnetic radiation around us, aptly named visible light. The study of remote sensing is based directly on electromagnetic waves and how objects interact with them.

Electromagnetic waves can be seen as a consequence of Maxwell's Equations, which are as follows:

$$\nabla\times\vec{E} = -\frac{\partial\vec{B}}{\partial t} \tag{1}$$

$$\nabla\times\vec{H} = \vec{J} + \frac{\partial\vec{D}}{\partial t} \tag{2}$$

$$\nabla\cdot\vec{D} = \rho \tag{3}$$

$$\nabla\cdot\vec{B} = 0 \tag{4}$$

We reproduce the derivation of the wave equations from Maxwell's equations in Appendix A. Further discussion of the equations is beyond the scope of this work, yet we present the key finding that electromagnetic radiation consists of paired electric and magnetic fields perpendicular to the direction of propagation and to each other. These results are visualized in Figure 6.

We also note that in a vacuum, electromagnetic waves obey:

$$c = \lambda\nu \tag{5}$$

where (c) is the speed of light, (λ) is the wavelength, and (ν) is the frequency [9]. The frequency and wavelength of light are incredibly important in

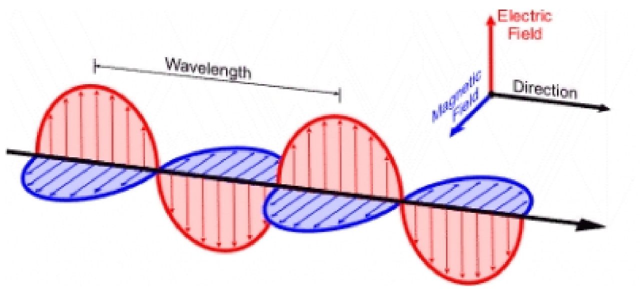


Figure 6: Visualization of electromagnetic waves [13]

Division	Spectral Range
Gamma rays	<0.03 nm
X-rays	0.03 - 300 nm
Ultraviolet radiation	0.30 - 0.38 μm
Visible light	0.38 - 0.72 μm
Infrared region	0.72 - 1000 μm
Microwave region	1 mm - 30 cm
Radio	$\geq$30 cm

Table 2: Electromagnetic spectrum divisions. Table reproduced from Introduction to Remote Sensing, 5th Ed. [9]

remote sensing because different surfaces reflect lights of different frequencies. Through an analysis of the light reflected we are able to distinguish what is on the surface we are observing.

2.1.2.1 Electromagnetic Spectrum

The electromagnetic spectrum refers to the range of all types of electromagnetic radiation. By convention, it is divided into multiple ranges, often referred to as *spectral ranges*, based on wavelength/frequency. Some of the more common divisions can be seen in Table 2 [9].

2.1.3 Image Acquisition

Before data can be used to better understand the land, data needs to be gathered. We describe the technologies required for taking images, which is better referred to as *image acquisition*.

Images can be acquired through *passive* and *active* sensors. Passive sensors measure electromagnetic waves that are naturally available, such as waves from the sun that are reflected by surfaces. An example of passive sensors are hyperspectral sensors. Active sensors provide their own energy and measure the waves that are reflected by the surface and returned to the sensor. An example of an active sensor is the synthetic aperture radar (SAR) [14]. Henceforth, this work focuses solely on passive sensors unless specified otherwise.

Technologies for aerial photography, and photography in general, rely on a set of standard components: (1) a light-sensitive surface to record an image, (2) a lens to gather and redirect light, (3) a shutter to control the entry of light, and (4) a camera body to hold the other components together and block external light. Digital cameras for aerial photography often also include positional and navigational systems to annotate images [9].

We go into some of the more intricate components described above in more depth in the following sections.

2.1.3.1 Recording Surface

There are two main techniques for digital image recording: charged-coupled devices (CCDs) & complementary metal-oxide-semiconductor (CMOS) chips. CCDs are more common in aerial photography, while CMOS chips are more common in consumer products [9].

Both technologies rely on a two-dimensional array of sensors that accumulate charge proportional to local intensity to take pictures. The difference between the two technologies is that CCDs expose all pixels at a time and read the values as the next image is taken, while CMO chips expose a single line at a time and only expose the following line as the data from the previous line

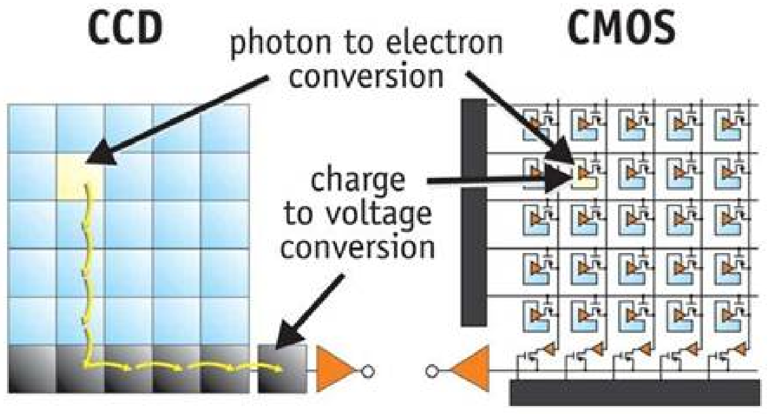


Figure 7: CCD and CMOS Sensor Diagrams [15]

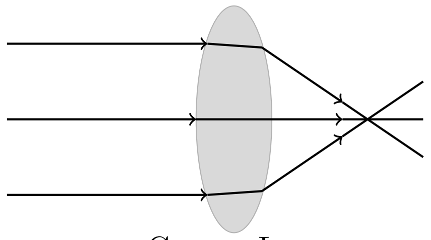


Figure 8: Cross-sectional view of an image passing through a simple convex lens

is transferred. CCDs further transfer the charge sequentially to convert into voltage, while CMOS chips do this on a pixel-by-pixel basis [9]. We present a diagram of the two sensors in Figure 7.

2.1.3.2 Lens

The lens, in its simplest form, is a glass disc with curved faces (see Figure 8). The change in material properties between air and the glass is carefully designed to refract light, maintaining color balance and minimizing optical distortions, and direct it to the recording medium.

One of the drawbacks of a single lens design is that they cause *chromatic aberration.* When incoming white light, which is a superposition of many

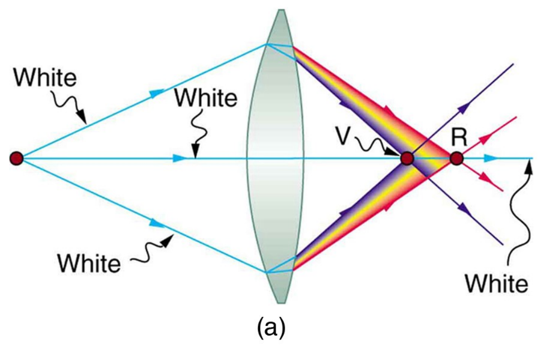


Figure 9: Visualization of white light through a convex lens [16]

electromagnetic waves at different frequencies, passes the lens, each frequency of light has a different focal length. This effect is shown in Figure 9. Most aerial cameras use *compound lenses*, which are formed from many separate lenses of varying sizes, shapes, and optical properties, which aim to, among other things, eliminate this effect [9].

2.1.4 Satellites

Recently, satellites have become one of the more prominent tools for remote sensing. They provide an overhead view of the land at fixed intervals called the *revisit rate*, based on the time it takes the satellite to orbit the Earth and return to the same spot.

The output imagery of satellites comes in the form of *rasters*. The word raster refers to imagery stored as pixel maps, as opposed to *vectors*, which are stored as points and shapes. Typical raster images are size (Im_w, Im_h, N_{bands}), where Im_w & Im_h correspond to the image dimensions, and N_{bands} corresponds to the different frequency radiations, also called bands, the image records (typical images we are used to have $N_{bands} = 3$ for the RGB bands). We opt to call raster images by the more general *imagery* henceforth.

Another important part of satellite images is the coordinate reference system (CRS), which denotes the projection the satellite used to map the 3-D world onto a 2-D image. Storing the appropriate CRS is critical for consistent imagery interpretation, as projections have a significant effect on the final image. These days, mathematical libraries such as GDAL [17] and Rasterio [18] provide tools to transform imagery from one CRS to another.

As noted before, one of the main challenges with satellites is the limited spatio-temporal resolution: satellites that orbit the Earth faster tend to have lower spatial resolution and those that orbit slower tend to have higher spatial resolution. This forms a trade-off between temporal and spatial resolutions.

We proceed to introduce two satellites catalogs, Landsat 8 and MODIS, which will be the two satellite catalogs used in this work.

2.1.4.1 Landsat 8

The Landsat program is the longest continuous global record of Earth's surface and is funded by NASA. The program includes 9 satellites as of December 2021, with the latest satellite, Landsat 9, having launched in late September of 2021.

Landsat 8 was launched in February 11, 2013. The satellite orbits the Earth at around 16-day intervals. The spectral bands of Landsat 8 range from 0.4 μm to 12.5 μm, with resolutions from 15 meters to 100 meters depending on the bands [19]. The band frequency ranges can be seen in Appendix B Table 6.

Landsat 8 data is available online for free using the Google Earth Engine Data Catalog [20].

2.1.4.2 MODIS

Moderate Resolution Imaging Spectroradiometer (MODIS) is a tool on the Terra and Aqua satellites ans is also supervised by NASA. MODIS tools are viewing the entirety of Earth's surface every 1-2 days with 36 spectral bands [21]. These spectral bands are between 0.405 and 14.385 µm, and are acquired at three spatial resolutions – 250m, 500m, and 1,000m, depending on the band [22]. The band frequency ranges are provided in Appendix B Table 7.

MODIS data is also available online for free using the Earth Engine Data Catalog [20].

2.1.4.3 Comparison

Landsat 8 and MODIS are two different satellite data catalogs, intentionally designed for different functionalities. The previous sections described the two in depth, and we summarize the key differences between the two satellites in Table 3.

Satellite	Spectral Range	Spatial Resolution	Revisit Rate
Landsat 8	0.435 μm - 12.5 μm	15 m - 100 m	$\approx$ 16 days
MODIS	405 nm - 14.4 μm	250 m - 1,000 m	1 day

Table 3: Comparison of Landsat 8 and MODIS satellites

We emphasize again that a high spatial resolution sensor comes at the cost of a lower temporal resolution. This can be seen clearly through a comparison of Landsat 8 and MODIS: whereas Landsat 8 has a high spatial resolution, roughly 16 times that of MODIS, it also has 16 times lower temporal resolution by comparison to MODIS.

2.2 Digital Video Processing

This work deals with sequences of imagery data, which are a form of video data. In this section we provide the background and terminology for understanding this type of data and describe some video processing concepts that have been successfully leveraged in state of the art research relating to our topic of interest.

2.2.1 Terminology

Images are pictorial representations of information, which can be still or time-varying. A still image refers to an image that is constant through time, while time-varying images are images that change with time. We denote a time-varying image as $f_o(x, y, t)$ and a still image as $f_o(x, y)$, where x and y are spatial variables while t is temporal. *Image sequences* are time-varying images represented by an ordered set of still images, $\{f_o(x, y)_{n_i} \mid 0 < n_i \leq N, n_i \in \mathbb{Z}\}$ where N is the length of the sequence and the index within the image sequence is represented by n_i [23].

A continuous time varying image is a function of two spatial dimensions and time. The observed spatial window is denoted as $\mathcal{W}$ and the time interval is denoted as $\mathcal{T}$. We combine these and denote the spatio-temporal region $\mathcal{W} \times \mathcal{T}$ as $\mathcal{W}_T$. In other words, $f_o : \mathcal{W}_t \to \mathbb{R}^C$, where C is the number of channels/bands of the image (i.e. for RGB images $C = 3$). The term *channel* is used within the context of digital video processing, while the term *bands* is an equivalent term used within the remote sensing community. The window, $\mathcal{W}$, is of size $w \times h$, where w is the picture width and h is the picture height. Therefore, $\mathcal{W} \sim \mathbb{S}_w \times \mathbb{S}_h$, where $\mathbb{S}_m = \mathbb{R}/m\mathbb{Z}$. Similarly, for a time interval of length k, $\mathcal{T} \sim \mathbb{S}_k$, making $\mathcal{W}_T \sim \mathbb{S}_w \times \mathbb{S}_h \times \mathbb{S}_k$ [24].

A *video* is a representation of information that includes still and time-varying images. Digital video refers to the electronic representation of video and can also be viewed as a discretization of video [23].

2.2.2 Geometric Image Formation

The task of image formation is the task of mapping a 4-D, time-varying 3-D scene, onto a 3-D space, the time-varying image plane [23]. This can be seen as attempting to define a mathematical formulation for image acquisition, and is generally formulated as follows:

$$\mathcal{P} : \mathbb{R}^4 \to \mathbb{R}^3 \tag{6}$$

$$(x_1, x_2, x_3, t) \to (x_1, x_2, t) \tag{7}$$

In other words, image formation is the process of mapping the 3-dimensional and time-varying world around us to the 2-dimensional and time-varying 'video' we see through our eyes. We assume that the time coordinate does not affect image formation, or rather that at a specific time, image formation is only determined by physical coordinates. Thus, we omit notation of the time domain and consider:

$$\mathcal{P} : (x_1, x_2, x_3) \to (x_1, x_2) \tag{8}$$

2.2.2.1 Perspective Projection

Perspective projection is a mapping based on the reflection of an image using the optical principles of an ideal pinhole camera as illustrated in Figures 10 and 11.

All the rays from an object pass through the center of projection, called the *focal point*, which is also the center of the lens. An algebraic mapping can be

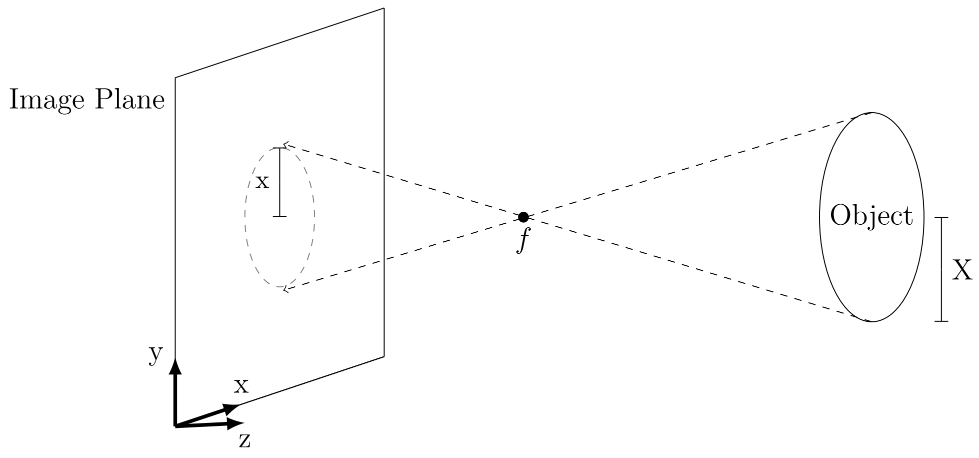


Figure 10: Perspective projection of object onto an image plane through pinhole f.

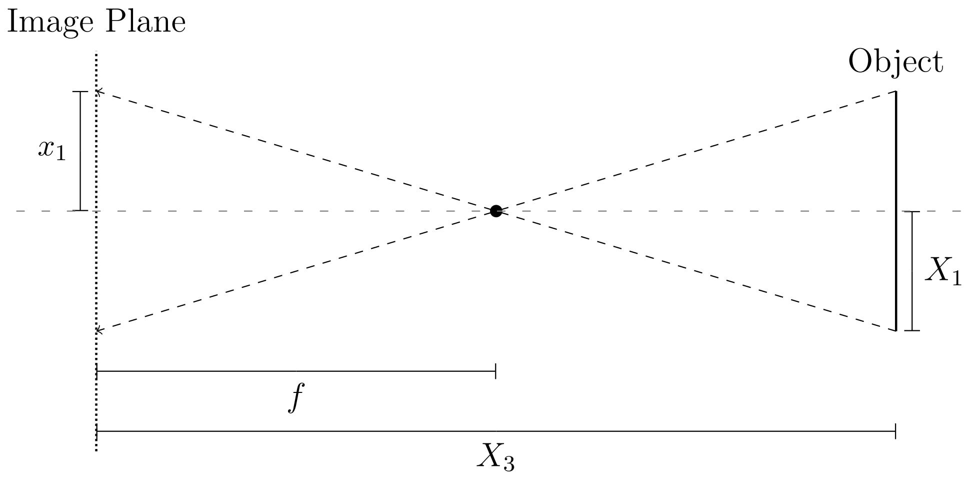


Figure 11: Perspective projection with annotated measurements

derived from this using triangle similarities, resulting in the following:

$$x_1 = -\frac{fX_1}{f - X_3} \tag{9}$$

$$x_2 = -\frac{fX_2}{f - X_3} \tag{10}$$

This is a mapping $(X_1, X_2, X_3) \rightarrow (x_1, x_2, 0)$, where X_i corresponds to the original object, x_i to the resulting image, and f to the focal point.

An additional simplification can be applied when $X_3 >> f$, when the object of the image is far away from the focal point, results in the following:

$$x_1 = \frac{fX_1}{X_3} \tag{11}$$

$$x_2 = \frac{fX_2}{X_3} \tag{12}$$

As a direct result of the algebraic derivation, perspective projection preserves relative lengths across surfaces parallel to the image plane (the plane $X_1 \times X_2$) only [23].

2.2.2.2 Orthographic Projection

Orthographic projection is another mapping, which assumes that all rays from the 3-D object travel parallel to each other, as seen in Figure 12.

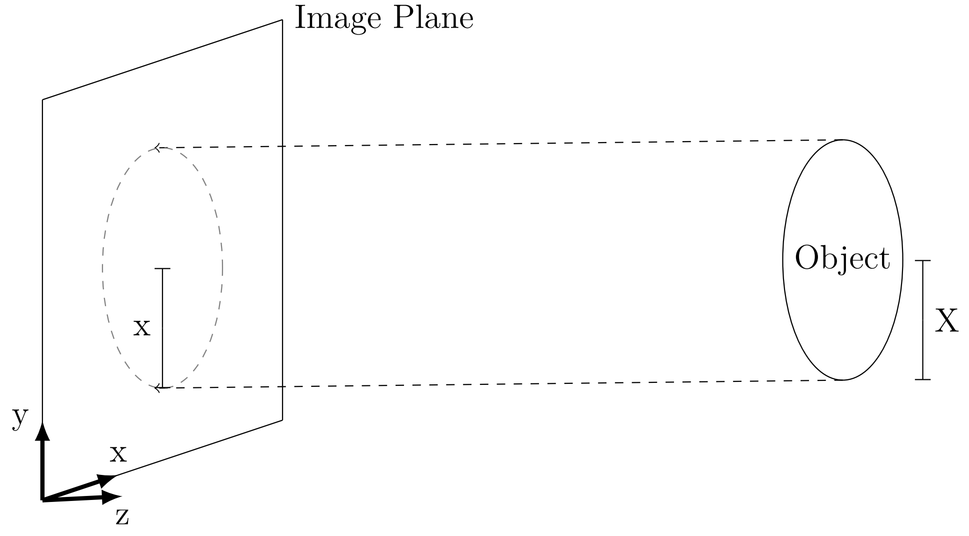


Figure 12: Orthographic projection of an object onto an image plane

When the image plane is parallel to the $X_1 \times X_2$ plane of the world coordinates, the orthographic projection can be described in Cartesian coordinates as:

$$x_1 = X_1 \tag{13}$$

$$x_2 = X_2 \tag{14}$$

The vector notation of this projection is as follows:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix} \tag{15}$$

$$x_h = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} X_h \tag{16}$$

Where x_h denotes the image plane points and X_h denotes the world coordinate points.

As can be seen from the mapping, the orthographic projection results in the same image regardless of how far the object is from the camera. This projection is useful when the scene is much farther than changes in depth of the scene, as is the case with satellites, and is often preferred over more complicated but realistic models because of its linearity and computational simplicity [23].

2.2.3 Photometric Image Formation

It is worthwhile to mention the existence of other, non-geometric models for image formation, a subset of which belonging under the category of *Photometric Image Formation.* Photometric image formation methods are based on image intensities being proportional to the amount of light reflected from the objects in the scene. Scene reflection is assumed to contain Lambertian and specular components (where light reflects diffusely and specularly, respectively), and together they form the resulting images [23].

These models are more accurate representations of image formation and help understand some of the specular noise in imagery yet are often more computationally expensive. This form of image formation is more important when working with active sensors, as active sensors consider surface effects in more detail. We provide a short description of one such photometric based method, the Lambertian Reflectance Model, in the following section, though omit many details for the sake of brevity.

2.2.3.1 Lambertian Reflectance Model

The Lambertian Reflectance Model assumes the imaged scene is composed of Lambertian surfaces, which when illuminated by a single point-source with uniform intensity results in image intensities:

$$f_o(x_1, x_2, t) = \rho N(t)\vec{L} \tag{17}$$

where ρ is the fraction of light reflected by the surface, $\vec{L} = (L_1, L_2, L_3)$ is the unit vector in the mean illuminant direction, and N(t) is the unit surface of the scene at spatial location $(X_1, X_2, X_3(X_1, X_2))$, and at time t. N(t) is given by:

$$N(t) = \frac{(-p, -q, 1)}{p^2 + q^2 + 1}^{1/2} \tag{18}$$

where p = $\frac{\partial X_3}{\partial x_1}$ and q = $\frac{\partial X_3}{\partial x_2}$ [23].

2.2.4 Spatio-Temporal Sampling

The process of taking a time-varying image $f_o(x, y, t)$ and recording it digitally is a form of spatio-temporal sampling. An image can be sampled in one, two, or three dimensions, where one of the sampled dimensions must be the temporal dimension. For digital video, which this work involves, three-dimensional sampling is always used. The subset of $\mathbb{R}^3$ that corresponds to the sampled output image (the finite set of points that are defined in both $\mathbb{R}^3$ and in the final image) is called the *sampling structure*, denoted as Ψ. We further see that $\Psi \subset \mathcal{W}_T \subset \mathbb{R}^3$. We denote the sampled image as $f_s(x, y, t)$, and note that $f_s(x_0, y_0, t_0) = f_o(x_0, y_0, t_0)|_{x_0, y_0, t_0 \in \Psi}$ [24].

2.2.4.1 Lattices

A useful mathematical structure for describing sampling of video is the *lattice*. A lattice Λ in D dimensions is defined as follows:

$$\Lambda = \{n_1\vec{v_1} + n_2\vec{v_2} + \ldots + n_D\vec{v_D} \mid n_i \in \mathbb{Z}\} \tag{19}$$

Or equivalently:

$$\Lambda = \{V \cdot n \mid n \in \mathbb{Z}^d\}, V = \left(\begin{array}{c|c|c|c} \vec{v_1} & \vec{v_2} & \ldots & \vec{v_D} \end{array} \right) \tag{20}$$

where $\{\vec{v_1}, \vec{v_2}, \ldots, \vec{v_D}\}$ is a basis of $\mathbb{R}^D$. Through the construction of Λ, we can immediately see that $\Lambda \sim \mathbb{Z}^{\mathbb{D}}$. We present an example of a 2-D lattice in Figure 13.

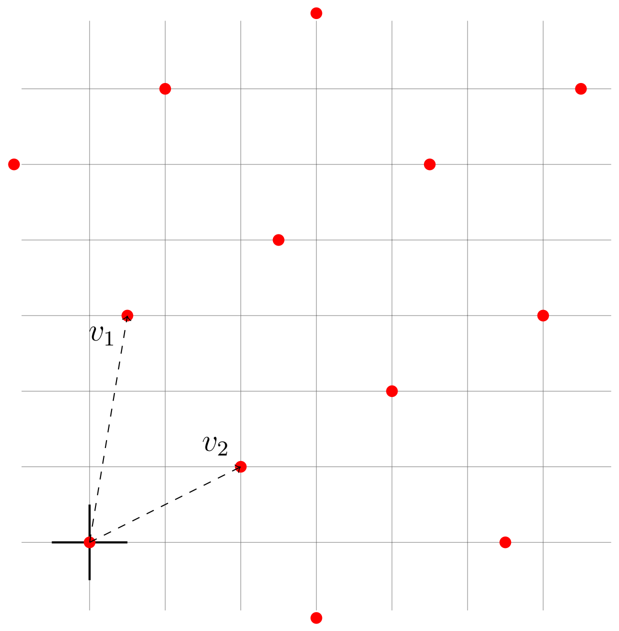


Figure 13: 2-D Lattice Λ shown in red, for $\vec{v_1} = (0.5, 3)$, $\vec{v_2} = (2, 1)$.

The matrix $V = (\vec{v_1} \mid \vec{v_2} \mid \ldots \mid \vec{v_D})$ is called the *sampling matrix* and when lattice Λ is generated from matrix V we write $\Lambda = \text{LAT}(V)$. We note that the sampling matrix for a lattice is not unique as there may be cases where $\text{LAT}(V) = \text{LAT}(U)$ for $V \neq U$. One such case is $\text{LAT}(V) = \text{LAT}(VE)$ where $|\det(E)| = 1$ and E is an integer matrix.

Another useful concept to consider is the *unit cell* of a lattice. A unit cell of lattice Λ is a set $\mathcal{P} \subset \mathbb{R}^D$ such that:

1. $(\mathcal{P} + s_1) \cap (\mathcal{P} + s_2) = \varnothing$ for $s_1, s_2 \in \Lambda, s_1 \neq s_2$
2. $\bigcup_{s \in \Lambda} (\mathcal{P} + s) = \mathbb{R}^D$

While a unit cell $\mathcal{P}$ is not unique for a given lattice Λ, the volume of a unit cell, $d(\Lambda) = |\det(V)|$ is unique (or in other words, it is the same across any unit cell of Λ) and its reciprocal denotes the *sampling density* [24].

2.2.4.2 Sampling on 3-D Structures

In the case of time-varying images, the lattices we deal with are simpler:

$$\Lambda_{xyt} = \{n_1 v_1 + n_2 v_2 + k v_3 \mid n_1, n_2, k \in \mathbb{Z}\} \tag{21}$$

Where the subscripts x, y, & t represent the dimensions the lattice is defined on (two spatial dimensions, x & y, and one temporal dimension, t) and are indexed by n_1, n_2, and k respectively.

The sampling structure, which we may consider as the final output of the sampling, is the sampled observed spatiotemporal window. This is equivalent to the intersection between $\mathcal{W}_T$ and a the sampling lattice Λ_{xyt} (or the union of two or more shifted lattices in more complicated cases):

$$\Psi = \mathcal{W}_T \cap \Lambda_{xyt} \tag{22}$$

We present an example of sampling in Figure 14.

We see that the resulting image, f_s, can be written in a number of ways:

$$f_s(x_1, x_2, t) = \begin{cases} f_o(x_1, x_2, t) & , (x_1, x_2, t) \in \Lambda \\ 0 & , (x_1, x_2, t) \notin \Lambda \end{cases} \tag{23}$$

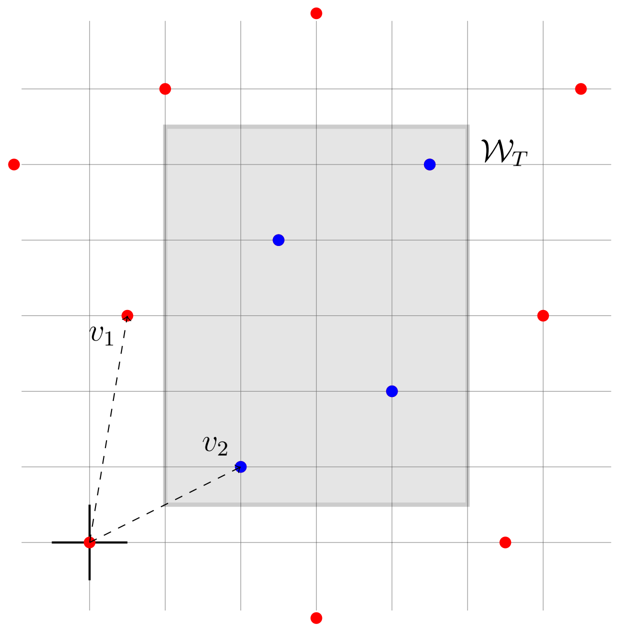


Figure 14: 2-D sampling using a lattice. Sampling lattice $\Lambda = LAT(V)$, where $V = (\vec{v_1} \mid \vec{v_2})$ for $\vec{v_1} = (0.5, 3)$, $\vec{v_2} = (2, 1)$. Spatiotemporal window $\mathcal{W}_T$ in gray. Sampling structure Ψ in blue.

$$= \sum_{(n_1,n_2,k)\in\Lambda} f_o(n_1, n_2, k) \cdot \delta\left((x_1, x_2, t) - (n_1, n_2, k)\right) \tag{24}$$

$$= \sum_{(n_1,n_2,k)\in\mathbb{Z}^3} f_o(n_1, n_2, k) \cdot \delta\left((x_1, x_2, t) - V\begin{pmatrix} n_1 \\ n_2 \\ k \end{pmatrix}\right) \tag{25}$$

2.2.5 Optical Flow

The displacement of the image plane coordinates (x,y) from time t to t‘ is called the correspondence vector. An optical flow vector is defined as the temporal rate of change of the image-plane coordinates $(v_x, v_y) = (\frac{dx}{dt}, \frac{dy}{dt})|_{(x,y,t)\in\mathbb{R}^3}$. An example of optical flow can be seen in Figure 15.

We note that the optical flow is equal to the limit of the correspondence vector as $\Delta t \to 0$, for $\Delta t = t\grave{} - t$:

$$\text{Optical Flow} = \lim_{\Delta t \to 0} \text{Correspondence} \tag{26}$$

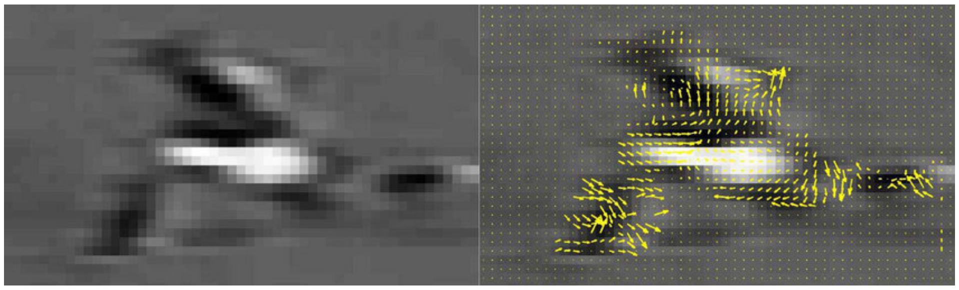


Figure 15: Image (left) and corresponding 2-D optical flow (right) of a footballer [25]

The optical flow is also known as the ”apparent 2-D velocity”. It is important to see how it is different from the actual 2-D velocity [23]. Optical flow has the following problems:

- It needs color variation for motion detection, so it fails to detect projected motion;
- It falsely identifies lighting changes as motion.

2.3 Machine learning

Machine learning is the practice of using statistical learning, calculus, and optimization techniques to let computers analyze data and identify patterns [26].

Machine learning can be used in a variety of different tasks including, but not limited to, speech synthesis, image classification, video classification, movie recommendation, and image super-resolution.

Machine learning has three major disciplines:

1. Supervised learning - deals with input pairs (x, y), with the task of finding some function $f(x) = \hat{y}$ such that $\hat{y} \approx y$.

2. Unsupervised learning - deals with data X with the task of finding subsets $X_1, X_2, \ldots, X_N \subset X$, with common characteristics.

3. Reinforcement learning - deals with environments, actions, and rewards. The task of a reinforcement learning algorithm, or *agent*, is to interact with the environment to determine an ideal set of actions to maximize the final reward.

This work focuses on supervised learning, though related works also include unsupervised learning.

2.3.1 Supervised Learning

As a general example, consider the subset of N input-output pairs drawn from a larger population $\mathbb{P}$:

$$\mathcal{S} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\} \tag{27}$$

$$\mathcal{S} \subset \mathbb{P} \tag{28}$$

The x_i's are referred to as the features, or as feature vectors, while the y_i's are called labels or classes.

A supervised learning algorithm attempts to find mapping $f : X \to Y$, where X is the input space and Y is the output space.

To find such function, the algorithm attempts to minimize a loss function $\ell(\hat{y_i}, y_i)$, $\ell : Y \times Y \to \mathbb{R}^{\geq 0}$, which serves as an estimate of how the model will perform on the population $\mathbb{P}$. Because loss functions are constructed such that $\lim\limits_{\hat{y_i} \to y_i} l(y_i, \hat{y_i}) = 0$, we see that minimizing the loss is equivalent to making our prediction $\hat{y_i}$ as close to the actual value y_i as possible, for all possible points in our sample set $\mathcal{S}$.

2.4 Deep Learning

Deep learning is a subset of machine learning that deals with neural networks. In recent years, the field has received wide recognition for its ability to outperform other machine learning algorithms in tasks relating to computer vision, natural language processing, graph processing, and more.

Neural networks are trained using gradient descent coupled with the chain rule, optimizing network parameters to minimize a loss function. Tools like TensorFlow [27] and PyTorch [28], which allow for automatic differentiation, make the creation and use of neural networks much easier than ever before, thus contributing significantly to the volume of works in the field.

2.4.1 Neural networks operations

2.4.1.1 Fully connected neural networks

One of the ways neural networks are constructed is using fully connected neural networks (FCNN), also referred to as dense layers.

FCNNs are considered networks because they are a composition of different functions:

$$f_{network}(x) = f^{(N)}(\ldots(f^{(2)}(f^{(1)}(x)))\ldots) \tag{29}$$

Any one of these transformations, $f(x)$, is called a *layer*. Due to the composition of linear functions being linear and therefore reducible to a single linear transformation, to obtain a meaningful formulation $f^{(i)}$ should be nonlinear.

A convenient way to create such nonlinear functions for multidimensional inputs is using matrix multiplication by a vector followed by a non-linear transformation. A matrix multiplication of $\mathbf{x}$ by a weight matrix $\mathbf{W}$ can be formulated as:

$$(\cdot,\cdot) : \mathbb{R}^{m\times n} \times \mathbb{R}^{n} \rightarrow \mathbb{R}^{m} \tag{30}$$

$$(\mathbf{W},\mathbf{x})_j = \sum_{i=1}^{n} w_{ji}x_i, \forall j \tag{31}$$

With a nonlinear transformation $f(\cdot)$ (known as an activation function) at the output of the matrix multiply, the functional form is as follows:

$$\mathbf{y} = f(\mathbf{W}_n \cdots f(\mathbf{W}_3 f(\mathbf{W}_2 f(\mathbf{W}_1\mathbf{x}))) \cdots) \tag{32}$$

As long as $f(\cdot)$ is differentiable we are able to train the now nonlinear model through automatic differentiation and gradient based optimization. We note that the nonlinear transformation may change between layers (i.e. $f_1()$ for layer 1, $f_2()$ for layer 2, and so on), but for convenience and clarity of notation we keep the transformation consistent across layers.

We add that the formulation:$f_{layer} = f(\mathbf{Wx})$ inherently limits the interaction between input parameters within a layer, due to the matrix multiplication, but is still expressible enough to capture a wide space of nonlinear functions once a composition of layers is created.

Finally, we add a bias $\mathbf{b}$ so that each transformation in a layer is not constrained to intercept zero. Therefore, the full form of a single FCNN layer is:

$$\mathbf{y} = f\left(\mathbf{Wx} + \mathbf{b}\right) \tag{33}$$

Figure 16 shows a FCNN with one hidden layer. The outputs of the hidden layer are called *latent variables.* Each hidden layer added to the network increases the complexity of data representations in the latent variables [29].

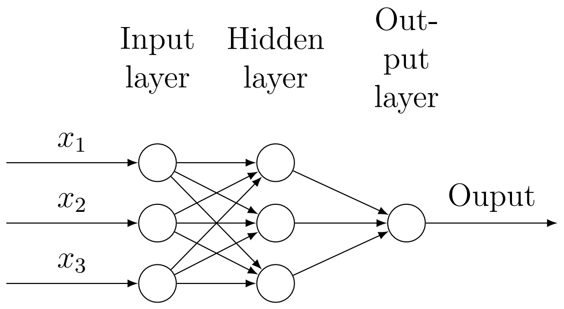


Figure 16: A graphical representation of an FCNN. Each arrow has a scaling factor w_{ij} associated with it. Each circle, known as a *neuron*, performs the operation $f(\mathbf{w}_i\mathbf{x} + \mathbf{b})$, where x refers to the layer inputs and w_i refer to the weights associated with the inputs for the given neuron. The transformation from one layer to the next, as described in Equation 33, is thus represented locally as inner products.

2.4.1.2 Convolutional neural networks

A convolutional neural network, also known as ConvNet or CNN, is a type of deep learning architecture specialized at capturing spatial relations between neighboring pixels. The idea for CNNs comes from filtering multidimensional signals, which uses convolutions, and what the network essentially does is sweep a small kernel across an input. Instead of learning a weight matrix $\mathbf{W}$ for every input-output pair, all that is needed is a set of kernels of much smaller sizes and convolutions.

A convolution of image I and kernel K is defined as follows:

$$\mathsf{S}_{i,j} = \sum_{i=m}\sum_{j=n} \mathsf{I}(m,n)\mathsf{K}(i-m, j-n) \tag{34}$$

$$= \sum_{i=m}\sum_{j=n} \mathsf{I}(i-m, j-n)\mathsf{K}(m,n) \tag{35}$$

This is equivalent to element-wise multiplication of a subset of the image with a flipped kernel, followed by a summation.

Because of the application, most implementation omit the flipping of the kernel in the process called cross-correlation:

$$\mathsf{S}_{i,j} = \sum_{i=m}\sum_{j=n} \mathsf{I}(m+i, n+j)\mathsf{K}(m,n) \tag{36}$$

The intuition behind this is that there is no need to flip the kernel when convolving, as we can simply learn the post-flip kernel [29].

Consider an image f, represented by tensor F and with shape $c \times h \times w$. The output of a 'convolution' of F with kernel K is:

$$\mathsf{Z}_{i,j,k} = \sum_{l}\sum_{m}\sum_{n} \mathsf{F}_{l,j+m-1,k+n-1}\mathsf{K}_{i,l,m,n} \tag{37}$$

over all indices l, m, n where the indexing for the summation is valid [29]. The output map Z has shape $d_o \times (h - k_h + 1) \times (w - k_w + 1)$, where $k_h \times k_w$ are the height and width of K, respectively.

Now we consider strided convolutions, where we skip over s input features each time the kernel is moved:

$$\mathsf{Z}_{i,j,k} = c(\mathsf{K}, \mathsf{X}, s)_{i,j,k} = \sum_{l}\sum_{m}\sum_{n} \mathsf{X}_{l,(j-1)\times s+m,(k-1)\times s+n}\mathsf{K}_{i,l,m,n} \tag{38}$$

We see that the output shape is now:

$$w_o = \lfloor (w_i - k_w)/s + 1 \rfloor \tag{39}$$

$$h_o = \lfloor (h_i - k_h)/s + 1 \rfloor \tag{40}$$

Striding a convolution is inherently a trade-off, as it decreases computational costs at the expense of reducing the output resolution.

Another important concept for CNNs is padding. When a convolution is padded, the input image I is surrounded by layers of zeros:

$$\begin{pmatrix} 0 & \cdots & & \mathbf{0} & & \cdots & 0 \\ \vdots & \ddots & & \vdots & & \iddots & \vdots \\ & & 0 & \mathbf{0} & 0 & & \\ \mathbf{0} & \cdots & \mathbf{0} & \mathsf{I} & \mathbf{0} & \cdots & \mathbf{0} \\ & & 0 & \mathbf{0} & 0 & & \\ \vdots & \iddots & & \vdots & & \ddots & \vdots \\ 0 & \cdots & & \mathbf{0} & & \cdots & 0 \end{pmatrix} \tag{41}$$

We note that the bolded 0's are vectors to match the size of the input image I.

Typically, a hyper-parameter p is used to denote how many layers of zeros to pad the input, so for $p = 2$:

$$\begin{pmatrix} 0 & 0 & \mathbf{0} & 0 & 0 \\ 0 & 0 & \mathbf{0} & 0 & 0 \\ \mathbf{0} & \mathbf{0} & \mathsf{I} & \mathbf{0} & \mathbf{0} \\ 0 & 0 & \mathbf{0} & 0 & 0 \\ 0 & 0 & \mathbf{0} & 0 & 0 \end{pmatrix} \tag{42}$$

One can immediately see that the added padding effectively increases the image size by $2p$, so the output shape is now:

$$w_o = \lfloor (w_i - k_w + 2p)/s + 1 \rfloor \tag{43}$$

$$h_o = \lfloor (h_i - k_h + 2p)/s + 1 \rfloor \tag{44}$$

2.4.2 Layer operations

2.4.2.1 Max-Pooling

The max-pooling operation refers to taking windows of the input, the size of which is determined by hyper-parameters, and returning the highest value within the window. The window may have any number of dimensions, again determined by external hyper-parameters.

2.4.2.2 Batch normalization

Batch normalization was proposed to combat *internal covariate shift*, which is when the internal distributions at a layer's input drift from zero mean, unit variance. To do this, batch normalization normalizes the input to each layer at each mini-batch during training and using accumulated statistics from the training period during inference [30].

The normalization that takes place for mini-batch $\mathfrak{B}$ during the training period is as follows:

$$\mu_{\mathfrak{B}} \leftarrow \frac{1}{m}\sum_{i\in\mathfrak{B}} x_i \tag{45}$$

$$\sigma^2_{\mathfrak{B}} \leftarrow \frac{1}{m}\sum_{i\in\mathfrak{B}} (x_i - \mu_{\mathfrak{B}})^2 \tag{46}$$

$$y_i \leftarrow \gamma\frac{x_i - \mu_{\mathfrak{B}}}{\sqrt{\sigma^2_{\mathfrak{B}} + \epsilon}} + \beta \tag{47}$$

where γ and β are learnable parameters, and ϵ is a small constant value for numerical stability.

At inference time, rather than normalizing using batch mean and variance, batch normalization uses an average of the mean and variance across all batches from the training set.

2.4.2.3 Transposed Convolution

While traditional upsampling uses zero-insertion or nearest-neighbor to up-sample an input, transposed convolutions are layers that use convolutions for up-sampling. What a transposed convolution attempts is to reimagine the input as being the direct result of the output and the kernel.

For example, take a convolution of an input image I of size 4×4 and a kernel K of size 2×2. The output image would be of size 3×3. A transposed convolution associated with the described convolution will have an output of size 4×4 when applied to an input of size 3×3.

To do this, a transposed convolution first inserts zeros into the image based on the stride s, then pads the image based on the padding parameter p, and finally performs a convolution like a CNN to get the output. It is critical to note that the parameters used for zero-insertion and padding are not the parameters s and p as the input.

For an image I, kernel K of size k, stride s, and padding p, the associated transposed convolution is described by kernel size $k' = k$, $s' = 1$, and $p = k - p - 1$ [31].

We see that for the previous example, $p' = 2 - 0 - 1 = 1$, so the input image would have an effective size of 5×5. Therefore, the output of a convolution with a kernel of size 2×2 is of size 4×4 as desired.

The challenge is the zero insertion, as the *image-size* $+2p - k$ may not be a multiple of s. If it is, the image used for the input has $s - 1$ zeros inserted between each input pixel. Therefore, the output image size is:

$$w_o = \lfloor (w_i - 1)s + k_w - 2p \rfloor \tag{48}$$

$$h_o = \lfloor (h_i - 1)s + k_h - 2p \rfloor \tag{49}$$

If it is not, in addition to the zero insertion of $s-1$ zeros at every pixel, zero padding is added to the bottom and right sides of the images. The added number of layers is:

$$a_{right} = \mathrm{mod}_s(w_i - k_w + 2p) \tag{50}$$

$$a_{bottom} = \mathrm{mod}_s(h_i - k_h + 2p) \tag{51}$$

This makes the output size:

$$w_o = \lfloor (w_i - 1)s + a_{right} + k_w - 2p \rfloor \tag{52}$$

$$h_o = \lfloor (h_i - 1)s + a_{bottom} + k_h - 2p \rfloor \tag{53}$$

2.4.3 Activation functions

Here we present several common activation functions, each of which is pertinent to the present work. A visualization of each of them can be seen in Figure 17.

2.4.3.1 Sigmoid

The sigmoid activation function is as follows:

$$\sigma : \mathbb{R} \to (0,1) \tag{54}$$

$$\sigma(x) = \frac{1}{1+\exp(-x)} \tag{55}$$

and gradient:

$$\sigma'(x) : \mathbb{R} \to (0,1/4] \tag{56}$$

$$\sigma'(x) = \frac{\exp(x)}{(\exp(x)+1)^2} \tag{57}$$

Due to the gradient being < 1 for all inputs, sigmoid-based networks are prone to having vanishing gradient problems, where the gradients become incredibly small, slowing or stalling learning.

2.4.3.2 Rectified linear units

The rectified linear unit (ReLU) has demonstrated significant performance increases for network generalization and increased training speed [32,33]. The ReLU activation is defined as:

$$\text{ReLU} : \mathbb{R} \to \mathbb{R}^{\geq 0} \tag{58}$$

$$\text{ReLU}(x) = \max(0, x) \tag{59}$$

with gradient:

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{60}$$

We see that the gradient of the ReLU is undefined at 0, though this is no concern in practice.

We note that training may halt if the input to a node is consistently < 0, as no gradient information will be passed down the network. A node where this occurs is referred to as a *dead ReLU*.

2.4.3.3 Leaky rectified linear units

Leaky ReLUs (LReLUs) are a modification to the ReLU designed to have some non-zero gradient when the input is negative, thus avoiding dead nodes [34]. The LeakyReLU with hyperparameter $\alpha > 0$ is:

$$\text{LReLU} : \mathbb{R} \to \mathbb{R} \tag{61}$$

$$\text{LReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \tag{62}$$

with gradient:

$$\text{LReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \leq 0 \end{cases} \tag{63}$$

2.4.4 Higher level architectures

2.4.4.1 Residual networks

Residual networks are a class of neural networks developed by He et al. in 2015 [35]. The architecture eases the training of deep neural networks by intentionally reformulating the network to learn *residuals*, instead of direct mappings.

A residual is a value r such that:

$$x = y + r \tag{64}$$

where y is the desired output and x is the input.

In learning the residual, the overall network learns mapping $\mathcal{F}(\mathbf{x})$ in order to perform another $\mathcal{H}(\mathbf{x})$, related as follows:

$$\mathcal{H}(\mathbf{x}) = \mathcal{F}(\mathbf{x}) + \mathbf{x} \tag{65}$$

A residual building block can be seen in Figure 18. We see that at the beginning of the block the input is sent down two paths: one where it is transformed by the learned network and the other where it is left untouched. The weighted-path performs $\mathcal{F}(\cdot)$ on the input $\mathbf{x}$ and then the output and the untouched input are added, resulting in mapping $\mathcal{H}(\mathbf{x}) = \mathcal{F}(\mathbf{x}) + \mathbf{x}$.

Residual networks have been shown to train faster and with higher accuracy than plain neural networks, and have also allowed for training of very deep networks of over 1000 layers.

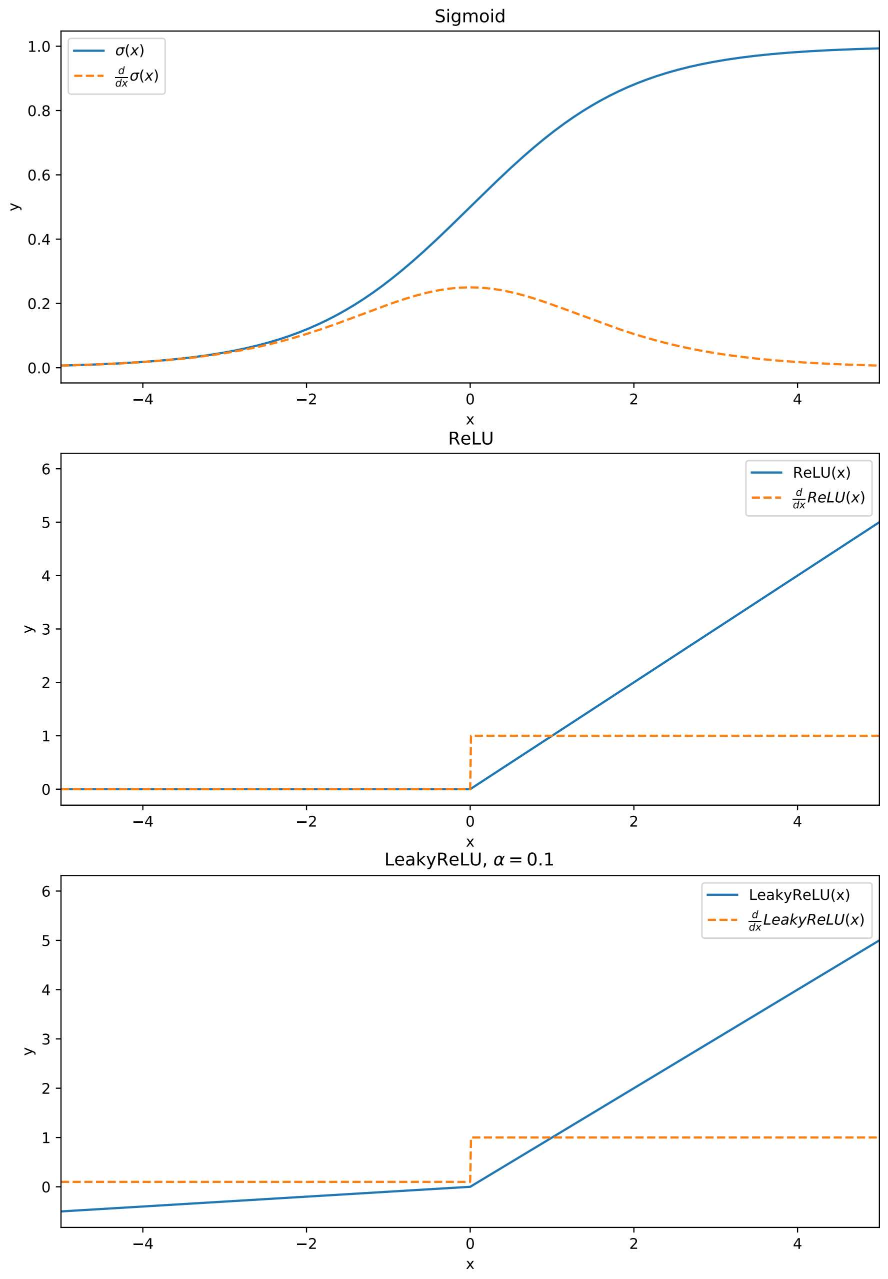
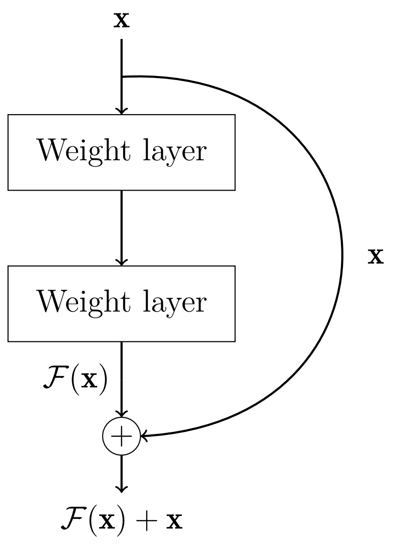


Figure 17: Visualization of various activation functions



$\mathbf{x}$
Weight layer
Weight layer
$\mathcal{F}(\mathbf{x})$
$\mathbf{x}$
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$

Figure 18: Residual building block

2.4.4.2 Encoder-Decoder Architectures

Encoder-decoder architecture is an umbrella term for networks that take an input, encode it into a latent space of lower dimension, and decode it back into an output, as seen in Figure 19.

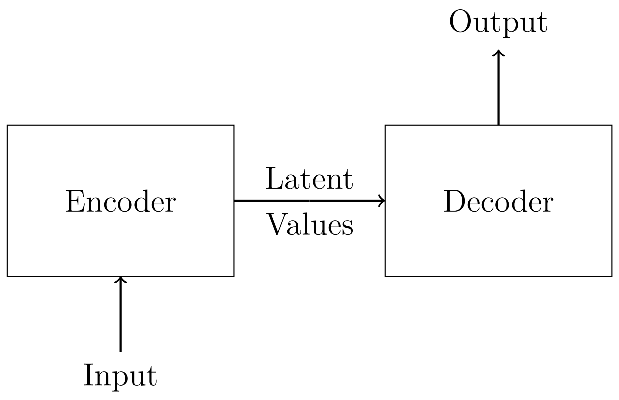


Figure 19: Diagram of encoder-decoder networks

The intuition behind these networks is like Occam's razor: simple is better. This works in the sense that the encoder learns to condense the relevant information for decoding in an efficient representation, the latent values, which it then passes to the decoder. Encoder-decoder networks have been used in

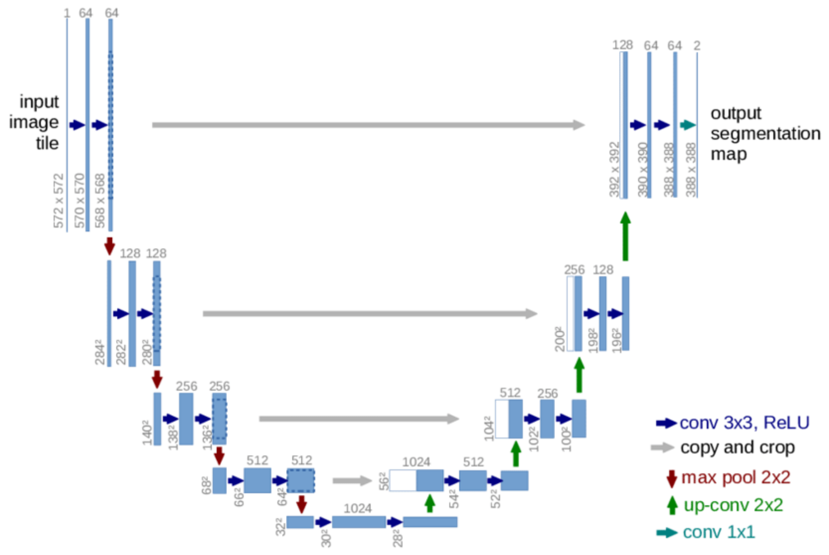


Figure 20: U-Net architecture for 32 × 32 sized image. Blue boxes denote feature maps and white boxes denote the copied residual feature maps [43]

a variety of tasks to achieve state of the art results in tasks such as language translation [36–38] and dance generation [39–42].

2.4.4.3 U-Net

The U-Net was proposed by Ronneberger et al. for biomedical image segmentation [43]. The architecture combines the ideas of encoder-decoder architectures and residual networks to produce a U-shaped network befitting the name. An example network can be seen in Figure 20.

The U-Net architecture can be divided into two parts: the encoder left side with the max pooling layers and the decoder with the up-convolutions. The U-Net encoder takes the input images and gradually down-samples it into several latent spaces, each corresponding to different image size. The smallest sized latent vector is passed to the decoder to use, followed by the second smallest, and so on, until the entire set of latent features are passed, and the decoder produces an output image the size of the input image. In other words, the encoder provides the decoder information at each image resolution, allowing the decoder to gradually build an output image by incorporating higher and higher resolution information until the output is produced.

2.4.4.4 Generative Adversarial Networks

Generative Adversarial Networks, or GANs, were proposed by Goodfellow et al. [44] as a form of generative networks for imagery. A generative adversarial network aims to model the distribution of the training set using two adversarial networks, called the generator and discriminator. Generator $\mathcal{G}(\cdot)$ aims to create data of the same distribution as the input data-set $p_{\text{data}}(x)$ using a random input $\mathbf{z}$ generated from a user defined distribution $p_{\mathbf{z}}(\mathbf{z})$, while discriminator $\mathcal{D}(\cdot)$ aims to uncover images created by the generator from a set of real and fake images. This effectively corresponds to a two-player minimax game:

$$\min_{\mathcal{G}} \max_{\mathcal{D}} V(\mathcal{G}, \mathcal{D}) = \mathbb{E}_{x \sim p_{\text{data}}(\mathbf{x})} [\log \mathcal{D}(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - \mathcal{D}(\mathcal{G}(\mathbf{z})))] \tag{66}$$

This has been proven to create a generator that is able to generate samples from the training-set distribution when the network converges.

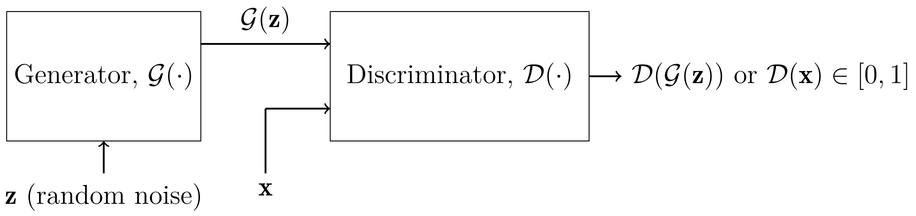

$\mathcal{G}(\mathbf{z})$
Generator, $\mathcal{G}(\cdot)$
Discriminator, $\mathcal{D}(\cdot)$
$\mathcal{D}(\mathcal{G}(\mathbf{z}))$ or $\mathcal{D}(\mathbf{x}) \in [0, 1]$
$\mathbf{z}$ (random noise)
$\mathbf{x}$

Figure 21: Outline of generative adversarial networks

2.4.4.5 Conditional Generative Adversarial Nets

Conditional Generative Adversarial Nets, or Conditional GANs, were proposed by Mirza et al. [45] as an extension of GANs. The method involves feeding the generator and discriminator data y to condition them. This corresponds to the two-player minimax game:

$$\min_{\mathcal{G}} \max_{\mathcal{D}} V(\mathcal{G}, \mathcal{D}) = \mathbb{E}_{x \sim p_{\text{data}}(\mathbf{x})} [\log \mathcal{D}(\mathbf{x}|\mathbf{y})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - \mathcal{D}(\mathcal{G}(\mathbf{z}|\mathbf{y})))] \quad (67)$$

The network is outlined in Figure 22.

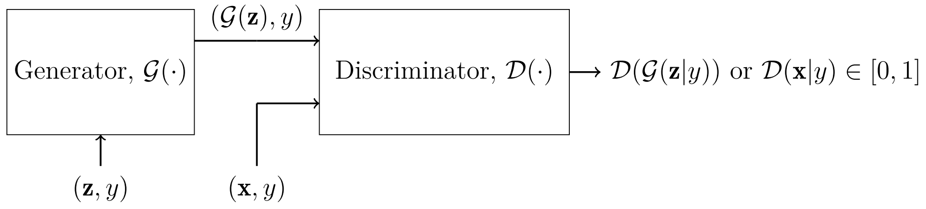

Generator, $\mathcal{G}(\cdot)$
$(\mathcal{G}(\mathbf{z}), y)$
Discriminator, $\mathcal{D}(\cdot)$
$\mathcal{D}(\mathcal{G}(\mathbf{z}|y))$ or $\mathcal{D}(\mathbf{x}|y) \in [0,1]$
$(\mathbf{z}, y)$
$(\mathbf{x}, y)$

Figure 22: Outline of conditional generative adversarial networks

2.5 Digital Multi-Sensor Temporal-Based Image Synthesis

We generalize imagery sensor-fusion and video interpolation techniques introduced in the introduction and describe what we call *digital multi-sensor temporal-based image synthesis* (DMST-IS). To understand what we mean by that, we begin with a non-digital form, *multi-sensor temporal-based image synthesis* (MST-IS). MST-IS is the task of taking a set of input time-varying images of the same spatial region $\mathcal{W}$ and synthesizing another time varying image based on these inputs. Equivalently, it is a mapping $\mathcal{F}$ such that:

$$\{f_1(x,y,t), f_2(x,y,t), \dots, f_D(x,y,t)\} \overset{\mathcal{F}}{\longmapsto} f_p(x,y,t) \quad (68)$$

where a number D of input time-varying images defined on $\mathcal{W}_T$ for $\mathcal{T} = [t_1, t_f]$ are used to synthesize time-varying image f_p. We refer to the set of input images $\{f_1, f_2, \dots, f_D\}$ as F_{input} henceforth. The inputs F_{input} can be loosely interpreted as a cross-sensor/band and cross-time window, a window of D sensors by T time range, though ordering for the sensors is not enforced (we note the term *cross-band* as the spatial region of all the images is consistent, permitting the interpretation of different sensors as bands of a single sensor). A more accurate interpretation is that F_{input} is a set of D temporal windows across the same range T.

Due to restrictions of digital processes, we sample the input in time, formulating the task of DMST-IS:

$$\begin{aligned}&\{f_1(x,y,t_{1,1}), f_1(x,y,t_{1,2}), \dots, f_1(x,y,t_{1,K_1}),\\ &\ f_2(x,y,t_{2,1}), f_2(x,y,t_{2,2}), \dots, f_2(x,y,t_{2,K_2}), \dots,\\ &\ f_D(x,y,t_{D,1}), f_D(x,y,t_{D,2}), \dots, f_D(x,y,t_{D,K_D})\} \overset{\mathcal{F}}{\mapsto}\\ &\ f_p(x,y,t) \qquad (69)\end{aligned}$$

for $t_1 \leq t_{i,J} \leq t_f, \forall i \in \{1,2,\dots,D\}, \forall j \in \{1,2,\dots,K_j\}$, where D is the number of input image sequences and K_j number of images in image sequence j. In words, it is a mapping of a set of image sequences to a single time-varying image. We emphasize through the subscript notation that the sampling is not necessarily performed in the same manner across all input images, nor does it need to be uniform. It is important to note a subtle difference between DMST-IS and MST-IS, that in DMST-IS having $f_p \in F_{input}$ is a non-trivial task, whereas in MST-IS it is.

Obtaining a continuous output on purely digital machines is tricky as the output the user receives must be digital. To work around this, some methods construct mappings that are able to generate images at any specified time [46]. This can be thought of as a continuous output that is inaccessible due to limitations of digital systems, yet a user can sample this output to access a single non-time varying image at any desired time.

When we further enforce constant sampling rate on each input image sequence, as is often the case with conventional sensors with fixed frame rates, Equation

69 reduces to:

$$
\begin{aligned}
&\{f_1(x,y,t_1), f_1(x,y,t_1+\Delta t_1), \ldots, f_1(x,y,t_1+K_1\Delta t_1),\\
&\ f_2(x,y,t_2), f_2(x,y,t_2+\Delta t_2), \ldots, f_2(x,y,t_2+K_2\Delta t_2), \ldots,\\
&\ f_D(x,y,t_D), f_D(x,y,t_D+\Delta t_D), \ldots, f_D(x,y,t_D+K_D\Delta t_D)\} \overset{\mathcal{F}}{\mapsto}\\
&\ f_p(x,y,t) \qquad (70)
\end{aligned}
$$

for time step Δt_i corresponding to the inverse of the sampling rate of image i. An example of this is shown in Figure 23.

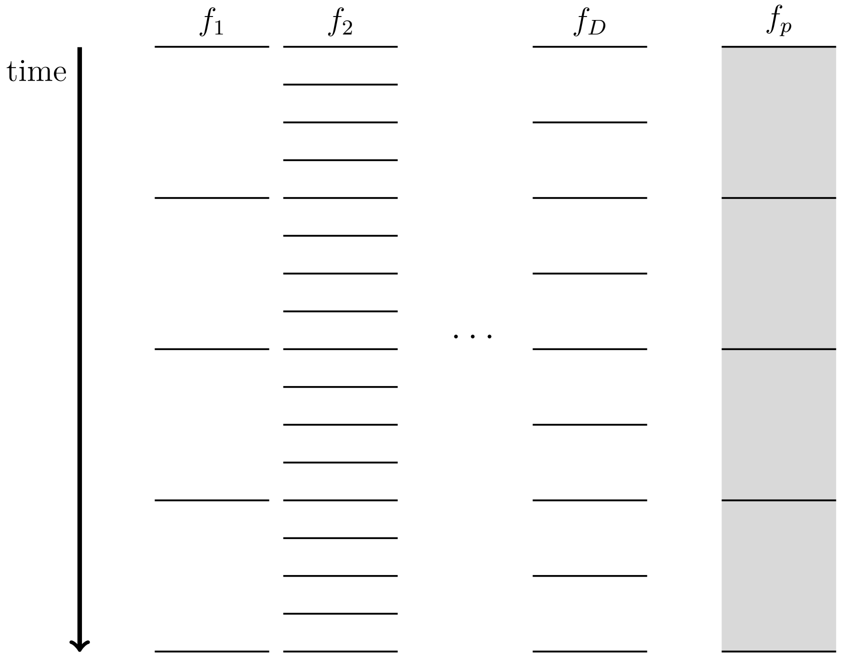


Figure 23: Example of multi-sensor temporal-based image synthesisas described in Equation 70 with $f_p \in F_{input}$. The black horizontal lines represent input information and the gray represents the output $f_p(x,y,t)$.

We now see that the formulation of video interpolation and satellite sensor fusion are cases of multi-sensor temporal-based image synthesis. Video interpolation is the case when $f_p \in F_{input}$ and $D = 1$ (which consequently forces $f_p = f_1$). Satellite sensor fusion is the case where $f_p \notin F_{input}$ and $t = t_i$ for some $t_i \in \mathcal{T}$. We discuss these cases more in depth in the following sections.

2.5.1 Deep Video Interpolation

Video interpolation is a form of image-to-image generation tasked with recreating what temporally occurs in between existing images for a single sensor. Video interpolation is used to overcome the temporal limitations of sensors, such as through the creation of slow-motion video from existing video.

The task of video interpolation has historically been viewed in different forms, which led to a variety of sub-tasks within the field. We proceed to justify these different video interpretation formulations and provide a set of prominent prior works in the field.

2.5.1.1 Continuous Output

Given a single input sensor, or $D = 1$, and the restriction that the predicted images are temporally between the input images, $f_p \in F_{input}$, Equation 69 reduces to:

$$\{f(x, y, t_1), f(x, y, t_2), \ldots, f(x, y, t_K)\} \overset{\mathcal{F}}{\mapsto} f(x, y, t) \tag{71}$$

where $t_1 < t < t_K$. This serves as the most general formulation of image interpolation and is shown in Figure 24.

We make the assumption that image generation only depends on the closest image neighbors (the images temporally closest to the image being generated), which allows us to reduce the mapping to predicting the time varying image between exactly two time-steps:

$$\{f(x, y, t_1), f(x, y, t_2)\} \overset{\mathcal{F}}{\mapsto} f(x, y, t) \tag{72}$$

where $t_1 < t < t_2$. This is visualized in Figure 25.

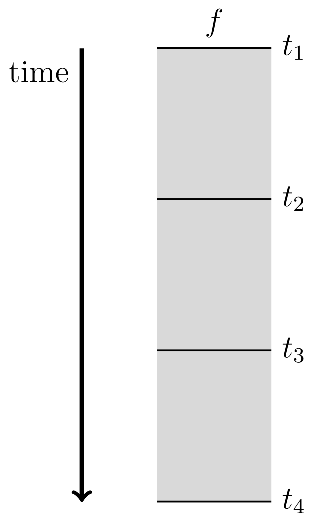


Figure 24: Visualization of video interpolation as described in Equation 71 for $K = 4$. The black horizontal lines represent known information $\{f(x,y,t_1), f(x,y,t_2), \ldots, f(x,y,t_4)\}$ whereas the gray represents the output $f(x,y,t)$.

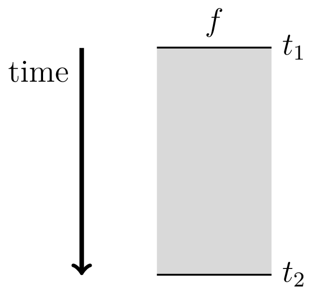


Figure 25: Visualization of video interpolation as described in Equation 72. The black horizontal lines represent known information $\{f(x,y,t_1), f(x,y,t_2)\}$ whereas the gray represents the output $f(x,y,t)$.

2.5.1.2 Discrete Output

Additional assumptions allow us to simplify our formulation further. If we choose to predict a fixed number n of output images, equivalent to sampling the output image at some time points between the times of the input images, Equation 73 reduces to:

$$\{f(x,y,t_1), f(x,y,t_2)\} \overset{\mathcal{F}}{\mapsto} \{ \quad f(x,y,t_i), f(x,y,t_{i+1}), \ldots, f(x,y,t_{i+n})\} \quad (73)$$

where $t_1 < t_i, t_{i+n} < t_2$. An example of this is shown in Figure 26.

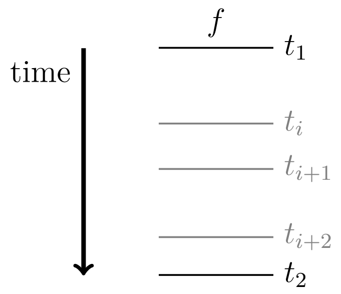


Figure 26: Example of video interpolation as described in Equation 73. The black horizontal lines represent known information $\{f(x, y, t_1), f(x, y, t_2)\}$ whereas the gray lines represents the outputs $\{f(x, y, t_i), f(x, y, t_{i+1}), f(x, y, t_{i+2})\}$.

2.5.1.3 N-Image Interpolation

When we further choose to predict these n images at fixed time-steps, as one would get from a fixed frame rate camera, we get:

$$\begin{aligned}&\{f(x, y, t_1), f(x, y, t_2)\} \overset{\mathcal{F}}{\mapsto} \\ &\{f(x, y, t_1 + \Delta t), f(x, y, t_1 + 2\Delta t), \ldots, f(x, y, t_1 + n\Delta t)\} \qquad (74)\end{aligned}$$

where $\Delta t = \frac{t_2 - t_1}{n+1}$ is the video time-step for an output of n images (see that $t_1 + n\Delta t = t_2 - \Delta t$). This is called n-image interpolation. An example of n-image interpolation is shown in Figure 27.

2.5.1.4 Single Image Interpolation

When we set $n = 1$, we get single image interpolation:

$$\mathcal{F} : \{f(x, y, t_1), f(x, y, t_2)\} \mapsto f_1(x, y, t_{\frac{1}{2}}) \qquad (75)$$

where $t_{\frac{1}{2}} = \frac{t_2 - t_1}{2}$.

Though this last formulation may appear lacking compared to the prior formulations, one must consider iteratively applying single image interpolation using the generated images. After one iteration we generate 1 intermediate image,

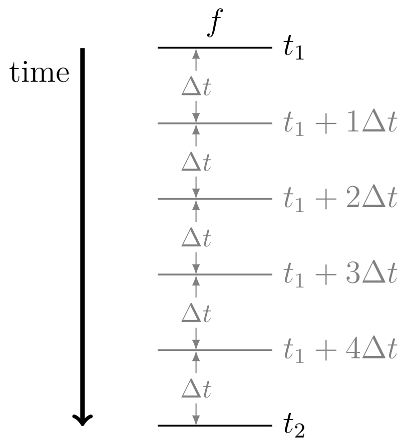


Figure 27: Example of n-image interpolation as described in Equation 74 with $n = 4$. The black horizontal lines represent known information $\{f(x, y, t_1), f(x, y, t_2)\}$ whereas the gray lines represents the outputs.

after 2 iterations we generate 3 images, and after 3 iterations we generate 7 images. After n iterations we generate we generate 2^{n-1} more intermediate images than iteration n-1, thus allowing us to create many intermediate images with relative ease. On the other hand, a critical drawback from this approach is the accumulation of error at each successive iteration.

2.5.2 Image Sensor Fusion

Satellite sensor fusion, in remote sensing, is the task of taking in input images from differing satellites at time t_i to generate a realistic prediction of what a different satellite output would look like at the same time. Such a task is useful for recovering sensor information due to sensor failure or when sensor information is not available.

The term *sensor fusion* may be deceiving for those not in the field, as it traditionally encompasses much more than the description above. *Sensor fusion* refers to all tasks taking a group of sensors, procuring any type of data, and using them together to synthesize information, which is not necessarily the output of any sensor. We see that even with the restriction of the pre-pended word 'satellite' this does not directly result in the description we proposed above. The reasons for the use of this term are historical, as prior works in the field were more in line with the general definition [47, 48], and the term has stuck to this day. The terms *image translation* and *image-to-image generation* are perhaps more accurate representations of our task, though are typically limited to having a single image sequence/satellite as an input, and are the terms used for similar objective as ours in fields outside of remote sensing. We use this terminology interchangeably.

We proceed to introduce multi- and single-input image translation formulations, though this work focuses on single-input image translation.

2.5.2.1 Multi-input image translation

Given the conditions described, we see that Equation 68 reduces to:

$$\{f_1(x, y, t_i), f_2(x, y, t_i), \ldots, f_D(x, y, t_i)\} \overset{\mathcal{F}}{\longmapsto} f_p(x, y, t_i), \forall t_i \in \mathcal{T} \tag{76}$$

or

$$\{f_1(x, y), f_2(x, y), \ldots, f_D(x, y)\} \overset{\mathcal{F}}{\longmapsto} f_p(x, y) \tag{77}$$

where $\mathcal{T}$ is the period of time the mapping is valid, D is the number of input sensors, and f_p denotes a sensor different from the input sensors ($f_p \notin \{f_1, f_2, \ldots, f_D\}$). We present a visualization of sensor fusion for D input sensors in Figure 28.

Due to the discrepancy in sampling rates of the different input sensors and the lacking benefit from adding more inputs, imagery sensor fusion is often done with one input sensor mapping to a single output one.

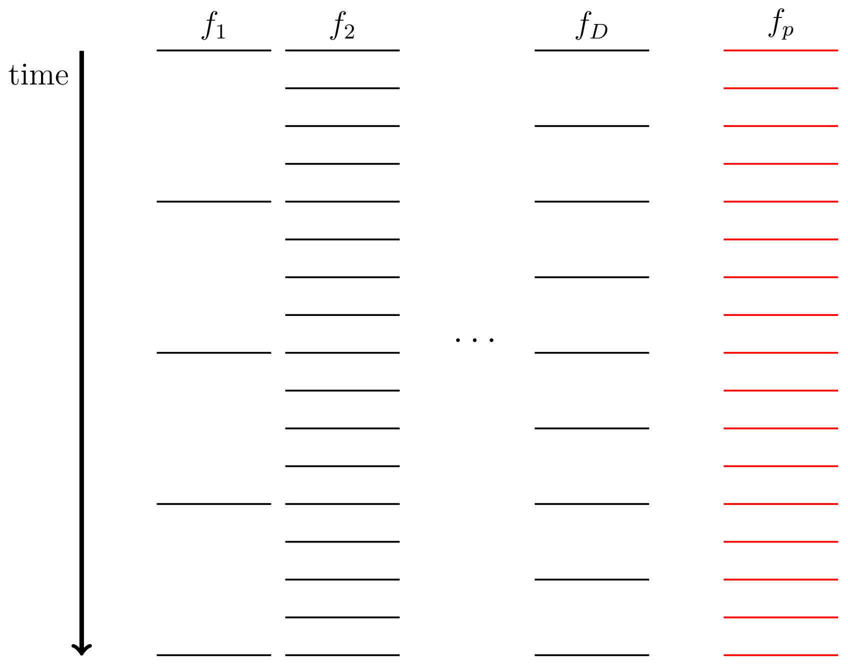


Figure 28: Example satellite sensor fusion for D inputs across multiple time steps. The black horizontal lines at each time point represent input information and the red represents the output $f_p(x, y, t)$. See that at some time points not all inputs are used in generating the prediction.

2.5.2.2 Single-input image translation

When there is a single input image sequence, the problem reduces to single-input image synthesis, also known as image-to-image generation and image translation. The numerical formulation is as follows:

$$f_1(x, y, t_i) \overset{\mathcal{F}}{\longmapsto} f_p(x, y, t_i), \forall t_i \in \mathcal{T} \tag{78}$$

We henceforth refer to single-input image translation wherever we mention *sensor fusion*, unless otherwise specified.

3 Related Works

3.1 Deep Video Interpolation

Work for image interpolation is varied, namely due to the variety of tasks that the title encompasses and the various techniques that have shown promise in producing accurate results. We divide prior work into **phase-based**, which characterized early works in the field, **optical flow-based**, which contain a majority of recent state of the art work, and **other**, which includes works that do not fit the prior categories.

3.1.0.1 Phase-based

One of the early methods for video frame interpolation is to use phases [49,50]. In general, these techniques use per-pixel phase-shift predictions to create an output image. Meyer et al. [49] proposed a method that uses phase information across a multi-scale pyramid coupled with a phase-based synthesis as a means of image interpolation, without using deep learning. First, wavelet-like pyramids are used to decompose the input images into several frequency bands and phase is extracted. The amplitude of the input images is also extracted. Using the phase difference between the inputs the phase is corrected at each scale of the pyramid, and the final output is used to adjust the phase difference found earlier. This adjusted value is used to interpolate the phase of the intermediate image, which is then combined with the blended amplitudes from the input images to generate the intermediate image. PhaseNet [50] extends this into a neural network using PhaseNet blocks composed of convolutional layers. The system takes the decomposition of two consecutive frames is obtained by applying the steerable pyramid filters (Ψ), resulting in R1 and R2, as the inputs to PhaseNet. The number of layers and their dimensions within

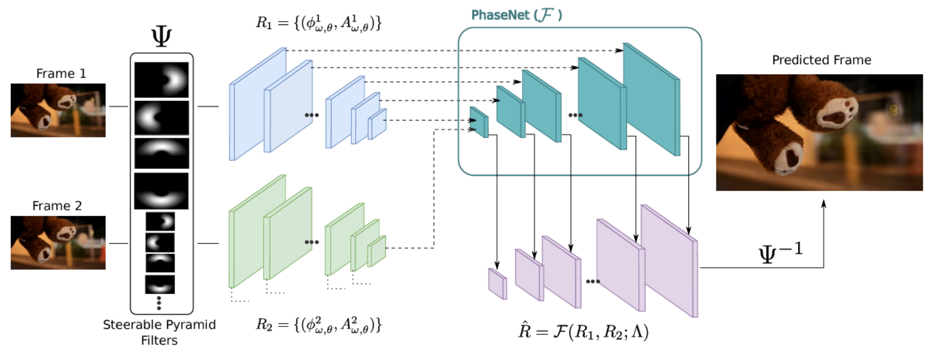

Ψ
$R_1 = \{(\phi^1_{\omega,\theta}, A^1_{\omega,\theta})\}$
PhaseNet ($\mathcal{F}$)
Predicted Frame
Frame 1
Frame 2
Steerable Pyramid Filters
$R_2 = \{(\phi^2_{\omega,\theta}, A^2_{\omega,\theta})\}$
$\hat{R} = \mathcal{F}(R_1, R_2; \Lambda)$
Ψ^{-1}

Figure 29: PhaseNet Architecture with only the first frame decomposition is displayed to avoid cluttering the image [50].

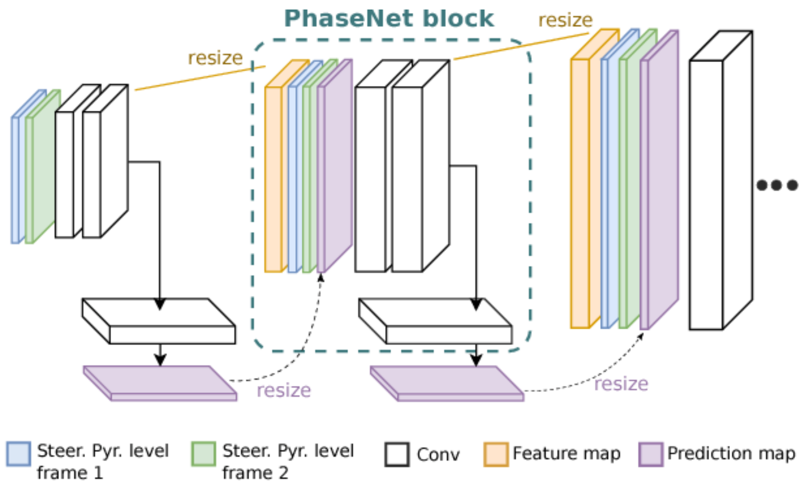


Figure 30: PhaseNet block. Each block of the PhaseNet takes the decompositions of the input frames at current level (shown in blue and green), performs two successive convolutions with batch normalization and ReLU, and predicts the response (amplitude and phase) with a convolutional layer with a hyperbolic tangent activation function [50].

the network mirror the input frame decompositions (i.e. is determined by Ψ). Each layer is composed of PhaseNet blocks, where each block takes the decomposition values from the corresponding level as input. The predicted filter responses ($\hat{R}$) are then used to reconstruct the middle frame. The overall architecture of PhaseNet is shown in Figure 29 and a PhaseNet block is shown in Figure 30.

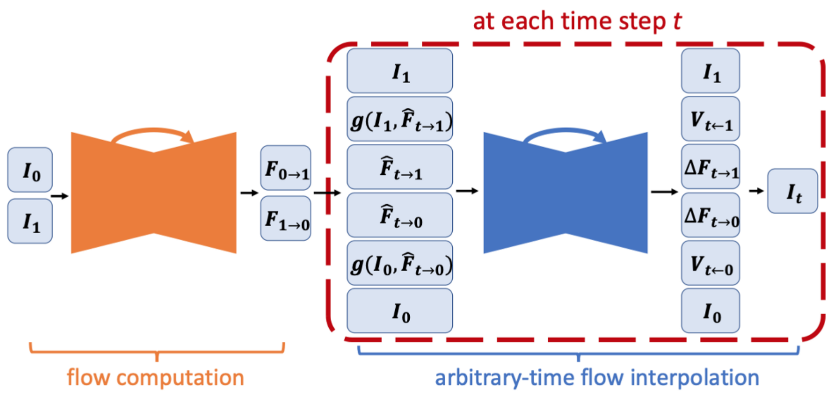


Figure 31: Super SloMo architecture [51].

3.1.0.2 Optical flow-based

The most common technique for image interpolation is to use optical flow. Image interpolation using optical flow typically has a mechanism of predicting optical flow between the input images and proceeds to warp the input images using the optical flow and combine them to synthesize the intermediate image/s.

Super SloMo [51] is one of the most prevalent optical flow models and was proposed to perform the task outlined in Equation 73, n-image prediction at variable times. Super SloMo uses bi-directional optical flows (one going forward in time and another backward) between the input images generated using a U-Net. The flows are linearly combined at each time step and passed to a second U-Net to refine the flow and predict a visibility map, the latter of which gives the model information on occlusion. Finally, the two images are warped using the refined optical flow, the visibility map is applied, and the results are linearly fused to form the output image(s). The architecture of Super SloMo is shown in Figure 31. The approach produces an arbitrary number of intermediate images as none of the network parameters are time-dependent.

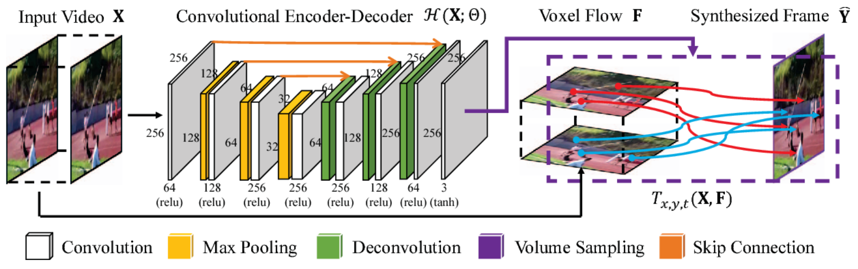

Input Video X
Convolutional Encoder-Decoder $\mathcal{H}(\mathbf{X};\Theta)$
Voxel Flow F
Synthesized Frame $\hat{\mathbf{Y}}$
$T_{x,y,t}(\mathbf{X},\mathbf{F})$
Convolution
Max Pooling
Deconvolution
Volume Sampling
Skip Connection

Figure 32: Pipeline for Deep Voxel Flow [54].

Vandal et al. [4] proposed the use of the Super SloMo architecture in remote sensing for weather tracking. The process involves the creation of task-specific Super SloMo networks for each spectral channel. The work shows the effectiveness of using task-specific optical flow for interpolating high-temporal weather events, and specifically for precipitation.

Some architectures propose creating an end-to-end system based on optical flow. Niklaus et al. [52] proposed a method that combines the estimation of optical flow and synthesis of the intermediate image using local convolutions for single image interpolation. Due to the large kernels of the prior method, Niklaus et al. [53] further proposed a similar method using 1D kernels, significantly reducing the number of parameters. Liu et al. [54] proposed another method, called Deep Voxel Flow, which uses *voxel layers* to predict intermediate images. Voxel layers are per-pixel, 3D optical flow vectors across time and space, which is like optical flow yet is only used as intermediate values. Due to this latter fact, the correctness of the voxel flow is never evaluated and thus requires no supervision. The voxel layers provide an intermediate optical flow-like information to the network, allowing it to predict intermediate frames using solely image triplets. The pipeline of Deep Voxel Flow can be seen in Figure 32.

Another approach is to extract additional information from images on top of optical flow. Niklaus et al. [55] proposed a method for using context-aware

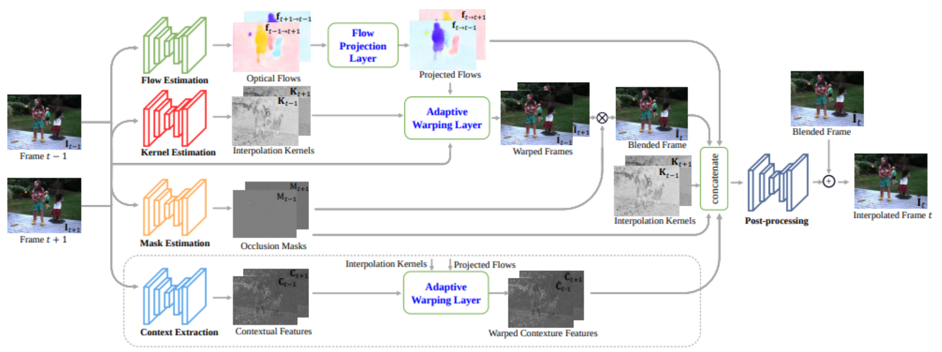


Fig. 4. **Network architecture of the proposed MEMC-Net and MEMC-Net*.** The context extraction module and its generated contextual features and warped contextual features are for MEMC-Net*.

Figure 33: Network architecture MEMC-Nett and MEMC-Net*. The context extraction module and its generatedcontextual features and warped contextual features are for MEMC-Net* [57].

synthesis. The method used a pre-trained neural network to extract per-pixel context maps, which was then used during warping to inject local context information during synthesis. Giving the network context information allows it to better handle challenging scenarios such as occlusion and motion. Bao et al. [56] proposed a method of using depth information, which can explicitly detect occlusion. In the proposed network, depth was used to refine optical flow predictions and as an input to the frame synthesis network, thus injecting information on occlusion into the model. Another similarly themed work, proposed by Bao et al. [57] is MEMC-Net, which combines motion estimation (ME) and motion compensation (MC). MEMC-Net extracts occlusion masks, optical flow, and interpolation kernels from each input and uses them together in a novel adaptive warping layer to obtain intermediate images. An additional network, called MEMC-Net* also uses context features. Both network architectures can be seen in Figure 33.

Additional methods attempt to remove assumptions on flow progression. Most state-of-the-art video interpolation methods assume uniform motion between frames, yet assumption does not necessarily hold [58]. One extremely common example of a scenario where the assumption fails is acceleration. To combat

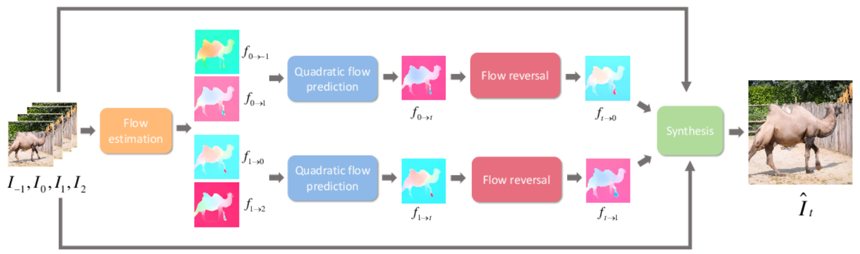


Figure 34: Pipeline for Quadratic Video Interpolation [58].

this, Xu et al. proposed quadratic video interpolation (QVI) [58]. It uses four input images (I_{-1}, I_0, I_1, I_2) to predict a single intermediate image. First, the network extracts forward and backward optical flow from images I_0 and I_1, and then it uses flow prediction with quadratic coefficients to obtain forward flow from I_0 to the intermediate image and backward flow from I_1 to the intermediate image. The flows are then reversed using a custom flow reversal layer and the intermediate image is synthesized. The pipeline for QVI is shown in Figure 34. Liu et al. [59] expand on this idea and introduce enhanced quadratic video interpolation (EQVI) to combat artifacts, inaccurate motion, and ghosting. This network uses the same input but generates three flow maps to estimate the quadratic model. It also relies on residual contextual synthesis. In this case, this means that spatially down-sampled images are fed into the network in conjunction with the original images, and then the low-resolution outputs are upsampled and joined with the original size using a multi-scale fusion network. The last improvement proposed is the multi-scale fusion network itself, which is used to create a pixel-weighted map for the low- and high-resolution inputs and then combines the two.

Other methods adapt techniques used in other fields to the image interpolation domain. Liu et al. [60] propose a method using *cycle consistency loss*, a loss typically used in unsupervised methods when mapping between domains. In general, cycle consistency loss is calculated using a known value and the result

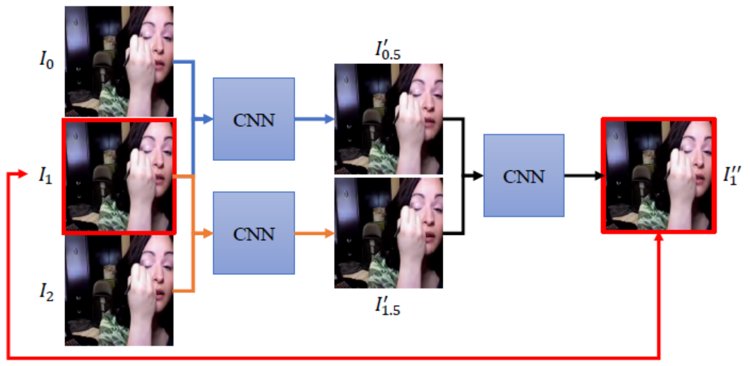


Figure 35: Cycle consistency loss for a triplet of images [60].

of mapping values to the desired domain (of which no information is available) and then back to the known domain to compare the predictions to known information. In this network, the cycle-consistency loss can be summarized as follows: take a triplet of input frames (I_0, I_1, I_2), pair them off into pairs $((I_0, I_1)$ and $(I_1, I_2))$, generate intermediate frames $\hat{I}_{0.5}$ and $\hat{I}_{1.5}$, and finally generate predicted frame $\hat{I}_1$ from the pair of predictions. The final prediction $\hat{I}_1$ is compared with known value I_1 to calculate the loss. This is visualized in Figure 35. Instead of generating images, the proposed network generates optical flow at each time step and uses warping to obtain the intermediate images. It also compares the flow from I_0 to I_2 to the flow from $\hat{I}_{0.5}$ to $\hat{I}_{1.5}$ as an additional loss term.

It is valuable to note that work in the field is not limited to supervised learning and extends to unsupervised machine learning. Reda et al. [46] proposed an unsupervised video interpolation technique using cycle consistency. The network trains on triplets of consecutive frames and can predict an arbitrary number of intermediate frames, as outlined in Equation 73. The network first predicts intermediate images $\hat{I}_t$ and $\hat{I}_{1+t}$ from image pairs (I_0, I_1) and (I_1, I_2), respectively. Then it uses the predicted pair $(\hat{I}_t, \hat{I}_{1+t})$ to generate image $\hat{I}_1$. This is visualized in Figure 36. The resulting prediction $\hat{I}_1$ is then compared

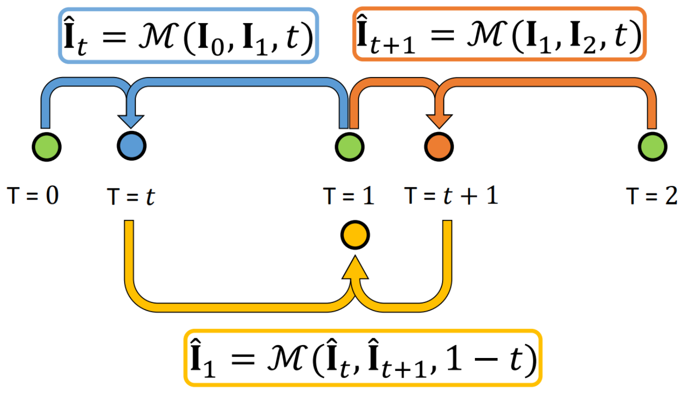

$\hat{\mathbf{I}}_t = \mathcal{M}(\mathbf{I}_0, \mathbf{I}_1, t)$
$\hat{\mathbf{I}}_{t+1} = \mathcal{M}(\mathbf{I}_1, \mathbf{I}_2, t)$
T = 0
T = t
T = 1
T = $t+1$
T = 2
$\hat{\mathbf{I}}_1 = \mathcal{M}(\hat{\mathbf{I}}_t, \hat{\mathbf{I}}_{t+1}, 1-t)$

Figure 36: Unsupervised video interpolation to generate sample at any time t [46].

with the true value I_1 to produce the final loss. If frames I_t and I_{1+t} are known, the network also includes loss terms comparing $\hat{I}_t$ to I_t and $\hat{I}_{1+t}$ to I_{1+t}. In the background the network uses Super SloMo [51] to approximate flows and generate intermediate images.

3.1.0.3 Other

Where a lot of contributions to the field have used optical flow, recent work shows that this is not a necessity. FLAVR, by Kalluri et al. [61], is a model that does not use optical flow. It uses 3D space-time convolutions to efficiently learn to reason about non-linear motion, occlusion, and temporal abstractions, without the need to deliberately extract values from the input. The model, seen in Figure 37, is a composition of U-Net and residual network architectures and is trained using only L1 loss. This design allows FLAVR to predict multiple frames in one inference forward pass. The model produces high accuracy inferences at a much faster speed than prior models with comparable or higher accuracy to optical flow models.

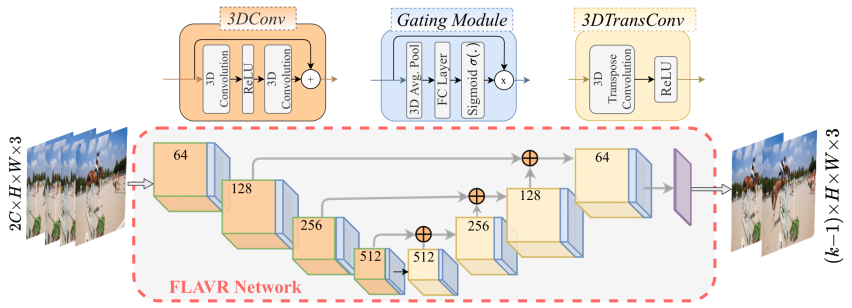


Figure 37: FLAVR Architecture. FLAVR is U-Net style architecture with 3D space-time convolutions (orange blocks) and deconvolutions (yellow blocks). It uses channel gating after all (de-)convolution layers (blue blocks). The final prediction layer (the purple block) is implemented as a convolution layer to project the 3D feature maps into (k−1) frame predictions [61].

3.2 Imagery Sensor Fusion

Satellite, or imagery, sensor fusion is a topic relevant to several different fields, thus prior work in the field is varied. For convenience, we divide prior works into those outside the field of remote sensing (external) and those within.

3.2.0.1 External

One of the critical advances in image-to-image generation is the advent of the conditional GANs [45]. Based on this work, Isola et al. [62] proposed pix2pix, a conditional image-to-image generator which conditions on an input image from one domain to generate an image from a different domain. Some domain examples include street view maps to labels for autonomous vehicles, black and white images to color, and aerial imagery to maps. The network architecture is strongly based on U-Net [43].

Another supervised method for image-to-image generation is proposed by Wang et al. [63]. The system generates realistic high-resolution images using a novel adversarial loss. The loss improves upon existing GAN loss by

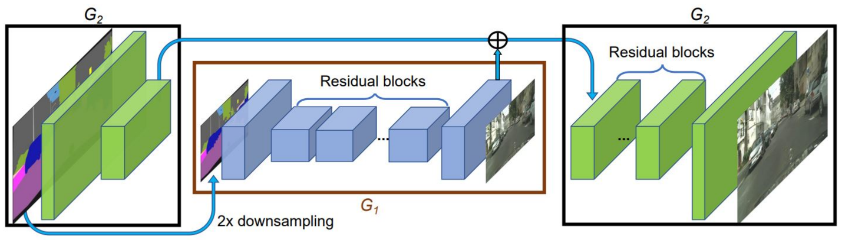


Figure 38: Generator architecture proposed by Wang et al. [63]. Generator is trained in parts, first G1, then G1 appended with G2 and finally the entire network together.

incorporating a feature matching loss, where features are extracted in multiple layers of the discriminator. The work also proposes a new generator architecture, shown in Figure 38.

A significant portion of work in this field is in unsupervised learning, justified by the difficulty of having accurate labels across domains. Taigman et al. proposed the Domain Transfer Network (DTN) [64] as a means of asymmetric unsupervised domain transfer from one domain to the other. The model employs a compound loss functions to encourage a GAN to sample from the desired output domain. The compound loss involves a GAN loss, a loss to minimize the difference between generated images with inputs from the output domain and the actual inputs ($|f(x) - f(G(x))|$), and a loss to encourage the mapping to be an identity mapping. Zhu et al. proposed CycleGAN [65], a cycle consistent adversarial network, as a means of unsupervised bi-directional image-to-image generation. CycleGAN is a system composed of a pair of conditional GANs mapping between input domains (one from domain X to domain Y and the other from domain Y to domain X) and a pair of discriminators, one for each domain. During training, the system maps an input from one domain to the other, uses the latter domain's discriminator to generate a loss, and then remaps the generated image back to its original domain to compose a cycle-consistency loss. The two cycle consistency losses capture the intuition

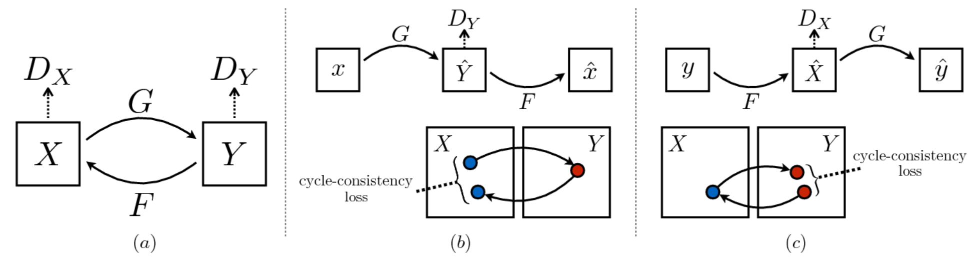


Figure 39: (a) CycleGAN architecture. (b) forward cycle-consistency loss: $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$, and (c) backward cycle-consistency *loss* : $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$ [65].

that translating from one domain to the other and back again should result with the original input. CycleGAN and the cycle-consistency losses are visualized in Figure 39. The architecture is based on the pix2pix model. Discovery GAN, dubbed DiscoGAN, by Kim et al. [66] learns the cross-domain relations across unpaired data. The model follows a similar procedure as CycleGAN. For one, the loss construction is different, CycleGAN provides a hyperparameter to tune the contribution of the cycle-consistency loss whereas DiscoGAN uses a simple sum. DiscoGAN also uses a slightly different generator architecture. The big difference is that DiscoGAN couples the generators, G_{AB} and G_{BA} share parameters. An overview of this model can be seen in Figure 40.

Multimodal image-to-image translation is also important to mention, as it aims to diversify the domains of the network outputs [67–69]. Zhu et al. [67] proposed a method for modeling a distribution of possible outputs, distilling ambiguity, and also encouraged the inevitable transformation, thus encouraging more diverse outputs. Huang et al. [68] proposed Multimodal Unsupervised Image-to-image Translation (MUNIT) to address failure prior methods to generate a diverse set of outputs. The framework breaks inputs into content and style and recombines the content with a random style to produce an output. By doing this, MUNIT forces output images to have a diverse content, thus varying the resulting images. We present a more detailed overview of MU-

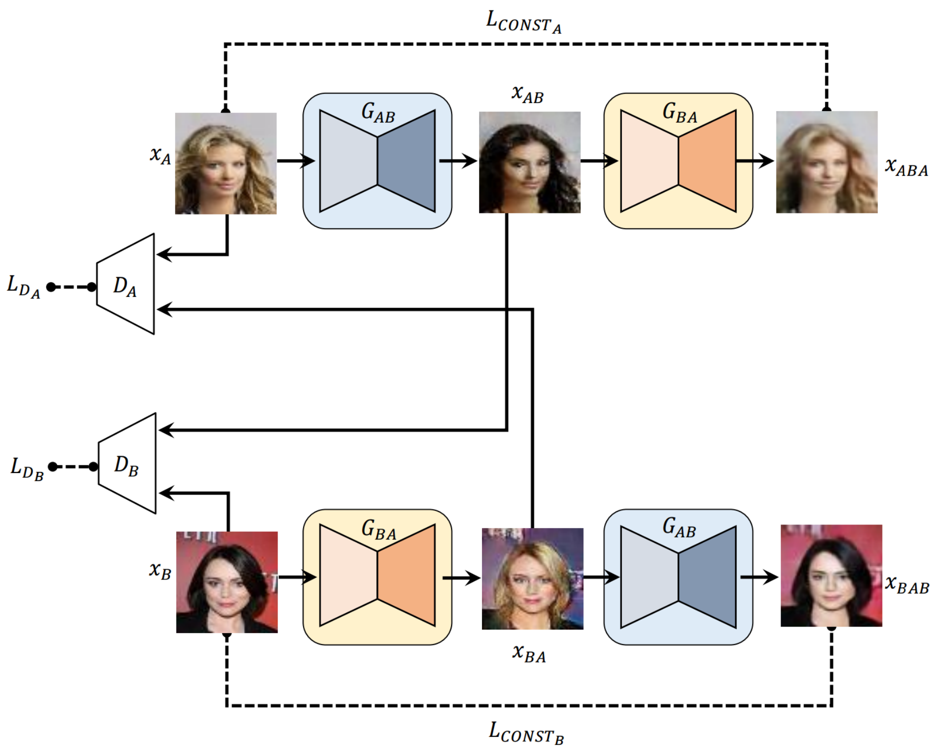


Figure 40: DiscoGAN architecture [66].

NIT architecture in Figure 41.Choi et al. [69] propose StarGAN, a generative network that can perform image-to-image translations from multiple domains using a single model. The model uses several different datasets while training, which makes the labels incomplete (where one dataset may label emotions it might not label gender and vice versa). To deal with this, the model ignores the unknown labels using a mask. To generate images, StarGAN employs a generator - discriminator pair. The discriminator is tasked with distinguishing between real and fake images, as well as with classifying the real images to the right domain. The generator takes an input image and a label of the target domain, generates a new fake image, and proceeds to use that image as an input to re-generate the input image with a label of the original domain. A visualization of how StarGAN performs image-to-image translation can be seen in Figure 42.

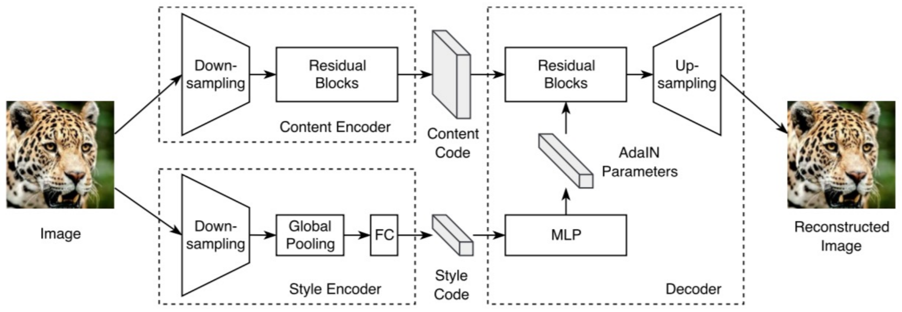


Figure 41: MUNIT architecture [68].

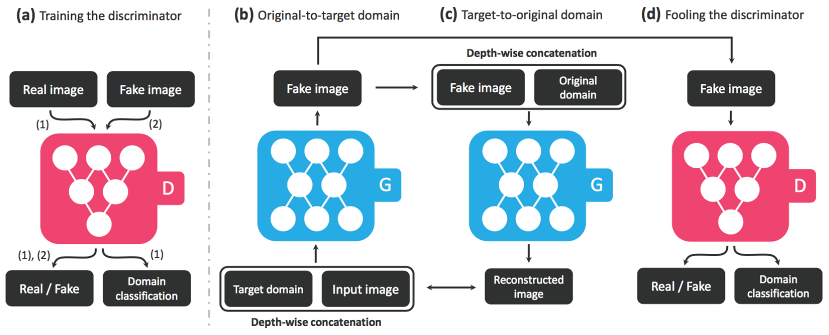


Figure 42: StarGAN architecture [69].

3.2.0.2 Remote sensing-based

Techniques from image-to-image translation have been used successfully in a remote sensing context.

Recent work by Vandal et al. [2] investigated the use of Variational Autoencoder GANs (VAE-GANs) for spectral satellite to satellite translation. The term variational autoencoder refers to the generator design in this work. *Autoencoders* are machine learning systems tasked with reducing the input into an encoding, or a latent space, and the term *variational* refers to how the encoding is constructed, a composition of means and standard of deviation metrics for the input, characterizing a set of normal distributions [70]. The work focused on generating synthetic bands for existing satellites using skip

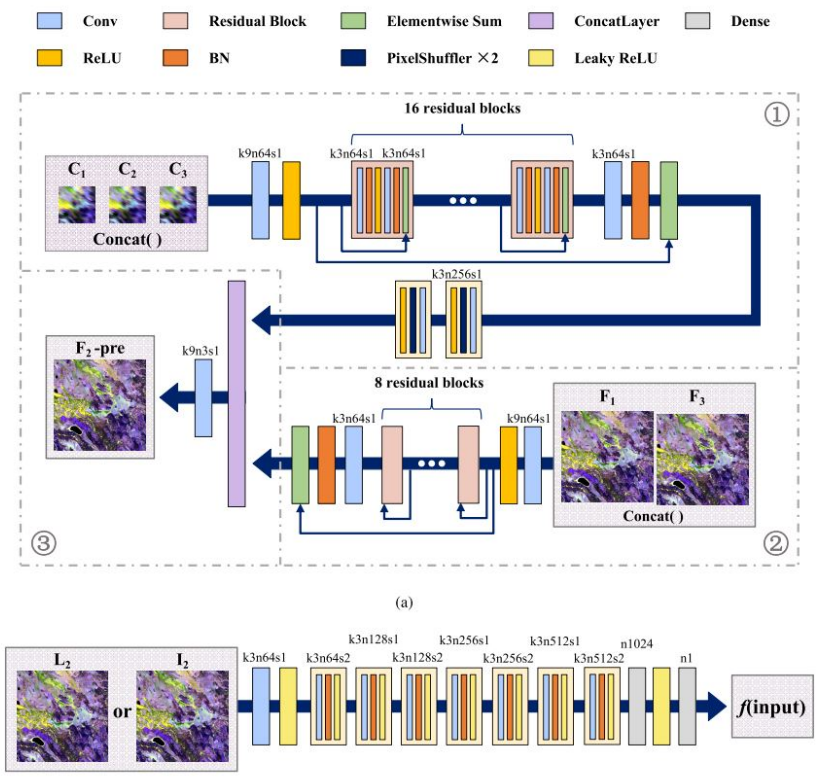


Figure 43: STFGAN architecture [3].

connections, a shared latent space, and a compound loss term composed of cycle-consistency, spectral reconstruction, and GAN loss terms.

STFGAN, proposed by Zhang et al. [3], introduces a GAN-based system for spatiotemporal fusion between MODIS and Landsat. The network first super-resolves MODIS images and combines the results with high frequency features from Landsat to generate Landsat-like images. The generator for this model uses a sequence of overlapping residual blocks as a means of creating accurate predictions. The model architecture can be seen in Figure 43.

CycleGAN-STF, by Chen et al. [1], combines synthesized images from high-resolution sensors and information from low-resolution sensors to generate higher accuracy synthesized images. In more detail, the model used Cycle-GAN to predict images both forward and backward in time. These predicted images are compared with low-resolution images for accuracy and enhanced using wavelets. The resulting images from this are then passed into the Flexible Spatiotemporal Data Fusion framework to improve the fusion performance.

4 Hybrid Video Frame Interpolation - Sensor Fusion Networks

4.1 Architectures

We attempt to improve the accuracy of synthesized imagery using a combination of image-to-image and video frame interpolation networks. To simplify the process, we base our work on two existing architectures, FLAVR [61] and pix2pix [62], and attempt to create meaningful methods for *synthesis*, as opposed to designing complete architectures from scratch. FLAVR was selected due to its simplicity, outstanding overall performance, and ability to work with noise. Pix2pix was selected because it is a widely referenced supervised learning model. Henceforth we refer to these two models as our *base models*. The outputs and inputs of these two base models within the context of this work are visualized in Figures 44 and 45.

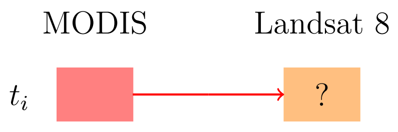


Figure 44: Pix2pix model inputs and output

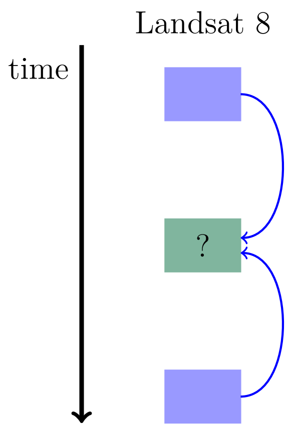


Figure 45: FLAVR model inputs and output

We propose four architectures for fusing these video frame interpolation and image-to-image translation networks:

1. Naive - averaging the outputs of the image translation and video interpolation models.
2. Secondary sensor-fusion - use the outputs of the image interpolation and satellite sensor fusion networks as inputs to a secondary sensor fusion network.
3. High-resolution information boosting - feeding the output of the video interpolation to the image translation model to boost its understanding of the high-resolution space.
4. High-temporal information injection - injecting low-resolution high temporal information to the video interpolation model.

We proceed to go into more depth into each of these hybrid architectures.

4.1.1 Naive

Since the predictions of both models at a given time step correspond to the same output image, a simple approach to combine the two networks is to train the networks independently and take an average of their predictions at each time step. This serves as a form of model ensemble, though the inputs for each of the models are different. We visualize the pipeline to conduct a naive prediction, within the context of this work, in Figure 46.

4.1.2 Secondary Fusion

A step beyond a simple average is to train a separate network to produce a more accurate synthesis of the two predictions. This thus serves as our

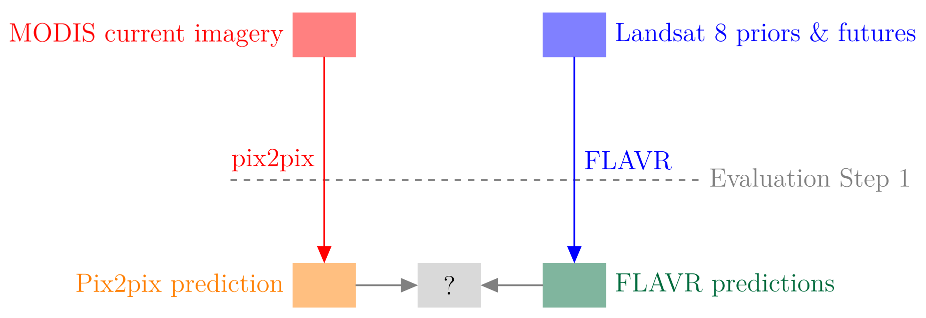


Figure 46: Naive architecture pipeline

next model, where both predictions are concatenated together into a 2-channel input and fed into an image-to-image network, specifically another instance of pix2pix. The pipeline for secondary fusion is shown in Figure 47.

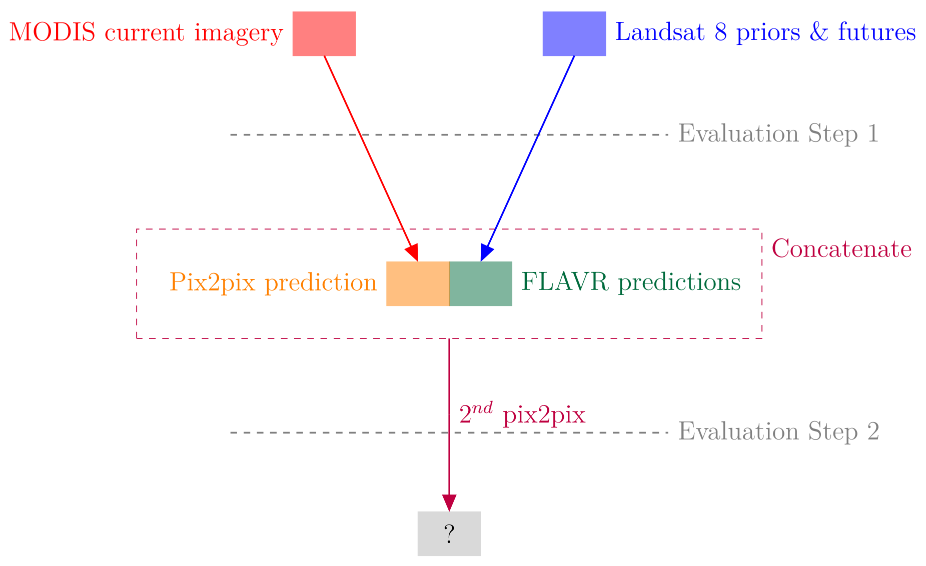


Figure 47: Secondary fusion pipeline

We then ask if there is a need to train three separate networks, which is inherently an expensive process. The following two architectures attempt to take raw inputs from one sensors and predictions from model to synthesize images, thus reducing the number of models trained to two.

4.1.3 High Spatial Resolution Information Boosting

One of the main challenges with image-to-image translation between MODIS and Landsat 8 is the difference in resolution. The high spatial resolution information boosting (HR-boosting) architecture attempts to take interpolated images from FLAVR and feed them into pix2pix alongside the original MODIS imagery to give pix2pix high-resolution information about the landscape. We do this similarly to the secondary fusion method, by concatenating the two images and training a 2-to-1 pix2pix model, though this serves to replace the original pix2pix model used for the secondary fusion architecture. We visualize the inputs and outputs for this model in Figure 48.

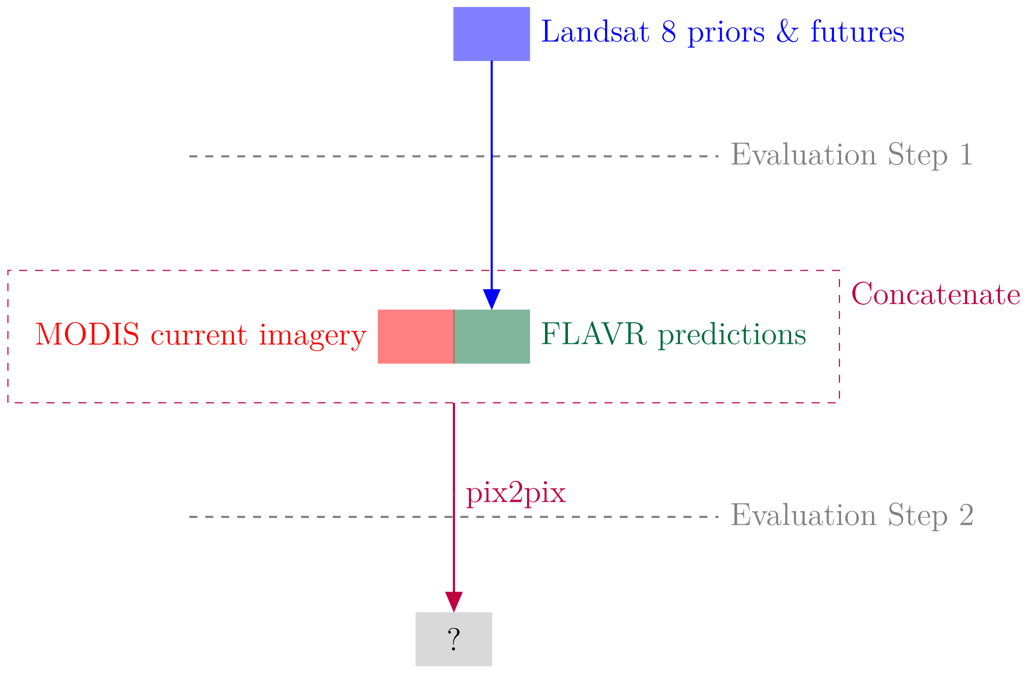


Figure 48: High-resolution information boosting pipeline

We provide a more physical interpretation of this architecture. Since MODIS imagery is taken in the current time step and over the same place, the action of concatenating the FLAVR prediction and the MODIS imagery is like adding bands to either image. Through this we add a high temporally accurate, low-

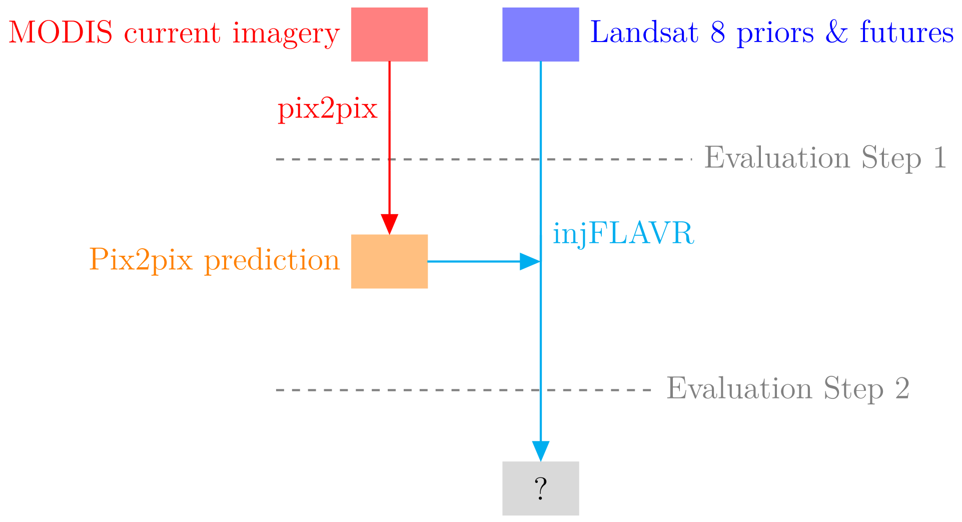


Figure 49: High temporal resolution information injection architecture's pipeline, injecting pix2pix predictions

resolution band to the predicted imagery from FLAVR or equivalently a high spatial resolution mapping of the area (that is somewhat temporally accurate) to the MODIS imagery. Through the following pix2pix model we combine the high spatial and high temporal information to synthesize the final image.

4.1.4 High Temporal Resolution Information Injection

The main concern with video frame interpolation techniques is the lack of high temporal information while generating imagery, permitting the model to veer off and not accurately track real-time events. We propose an architecture feeding a low-resolution version of the pix2pix prediction after the FLAVR encoder, thus injecting high-temporal resolution information during evaluation, and aptly call this method High Temporal Resolution Information Injection (HT-injection). We visualize the pipeline of the architecture in Figure 49 and further display the modified FLAVR architecture to allow for the injection of low spatial resolution high temporal information inputs, dubbed injFLAVR, in Figure 50.

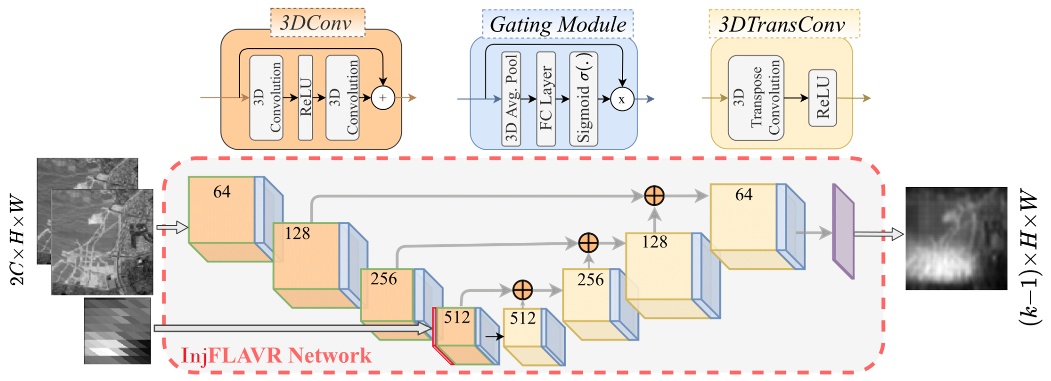


Figure 50: injFLAVR Architecture

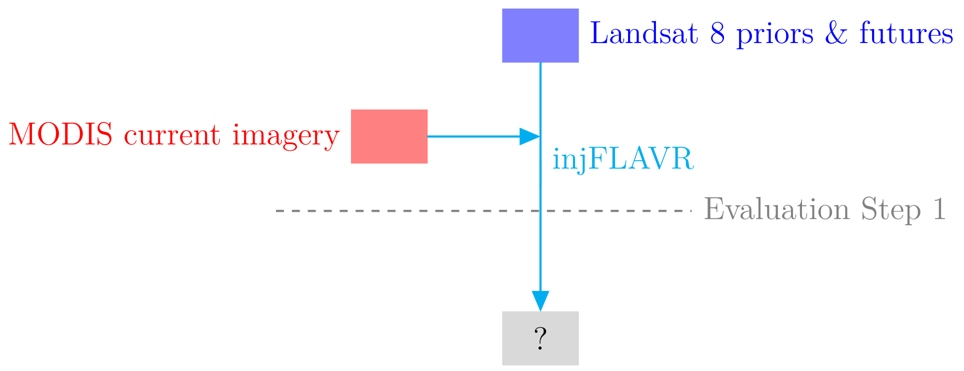


Figure 51: End-to-end High temporal resolution information injection architecture's, injecting MODIS imagery

We also ascertain how injFLAVR performs as an end-to-end method. This corresponds to the pipeline shown in Figure 51.

4.1.5 Comparisons

We provide a table to summarize the differences of the four proposed architectures:

Architecture	Trained Models	Evaluation steps
Naive	1 pix2pix & 1 FLAVR	1
Secondary Fusion	2 pix2pix & 1 FLAVR	2
HR-Boosting	1 pix2pix & 1 FLAVR	2
HT-Injection	1 pix2pix & 1 injFLAVR / 1 injFLAVR	2/1

We see that secondary fusion is the most expensive method to train due to the number of models it needs. We also note that the HT-injection, if using MODIS and Landsat 8 imagery only, requires the least amount of models. We further see that naive & HT-injection (with MODIS inputs) need a single evaluation step while the secondary fusion, HR-boosting, & HT-injection (with pix2pix predictions as inputs) need 2 evaluation steps. This metric can give us an estimation of how long it will take to evaluate an input given to each of these methodologies.

5 Experiments

5.1 Custom Dataset

Due to not finding public datasets that include both video frame interpolation and satellite sensor fusion, we gathered a custom MODIS and Landsat 8 dataset using Google Earth Engine [20]. The dataset was taken at one spatial location over Las Vegas from 2015 to 2020. We chose to focus on a single location as opposed to many locations to simplify the data gathering process, reduce the complexity of the code, and decrease the storage costs of the data. We note that a drawback of this is that the models may not generalize well to other locations, yet as a proof of concept the choice of a single location suffices.

The data was downloaded with a consistent spatial resolution of 15 meters across all bands and sensors, at the highest temporal resolution available for each satellite tool. For simplicity and due to time constraints, we chose to run experiments on a single band, band 1 from MODIS and band 1 from Landsat 8. The dataset was then split temporally using an 80-20% split into training and testing sets, respectively, without randomization. The lack of randomization is justified by the need for temporal windows for the input of video frame interpolation and the desire to not have the model introduced to the test set in any way.

Clouds were ignored throughout the experimentation. Images with clouds were left in the dataset as their removal will directly influence the temporal resolution of the inputs (making it inconsistent) and as the video interpolation model used is said to work well with noise [61]. To accommodate for this, Las Vegas was picked for a location to reduce the number of cloudy days. We further note that by achieving good results with the clouds as part of the dataset we create more robust models.

5.2 Bench-marking

Due to using a custom dataset, the performance of FLAVR and pix2pix needs to be evaluated on the newly created datasets to serve as a benchmark for the proposed architectures. The FLAVR model was trained on triplets of images from the high-resolution Landsat 8 data, taking the first and last frames to predict the intermediate frame, corresponding to the task of single image interpolation. Pix2pix was trained using pairs of MODIS and Landsat 8 data to predict Landsat 8 imagery from MODIS imagery.

We validated both models throughout training. For the FLAVR model, we run a 3-image window across the training set, index the resulting image sequences, and randomly separate the indices into training and validation sets. The pix2pix model is validated by randomly selecting image pairs from the two sensors.

5.3 Hybrid Models

Once we understood the benchmark level of accuracy, we proceeded to test the hybrid models. Due to the need of the base model predictions in multiple models, we first generated synthesized image datasets using each of the two base models and then used a combination of synthesized and non-synthesized imagery to train each of the hybrid architectures, as described in Section 4.1. We used Optuna [71] to tune model hyperparameters for each of the hybrid models but the naive architecture (which does not have additional hyperparameters) based on PSNR. The tuning was performed without data augmentation, as data augmentation is expensive and would significantly prolong the duration of tuning. Finally, we evaluated how similar the predictions of each of the architectures are to the true values on the test set. Testing employed the use of the Sewar package [72] and scikit-image metrics [73] for different image quality assessments. We used the metrics PSNR, MSE, SSIM, and PSNR-B [74] to evaluate our results. For HR-boosting, the highest PSNR network when fully tuned and trained had a significantly low performance and did not resemble the true images at all when trained with augmentation, so the second highest PSNR model based on the tuning was used instead.

6 Results and Discussion

6.0.1 Test Set Evaluation

We present the results of our benchmark and hybrid model tests in Table 5.

Architecture	PSNR	MSE	SSIM	PSNR-B
Pix2pix	17.655	0.024	0.404	1.97
FLAVR	14.202	0.052	**0.623**	13.69
Naive	13.380	0.056	0.435	1.50
Secondary Fusion	20.868	**0.014**	0.441	**24.01**
HR-boosting	21.0469	0.020	0.493	21.725
HT-injection (with pix2pix)	**21.501**	0.015	0.564	22.75
End-to-end HT-injection	20.618	0.022	0.510	22.08

Table 5: Model results on test set

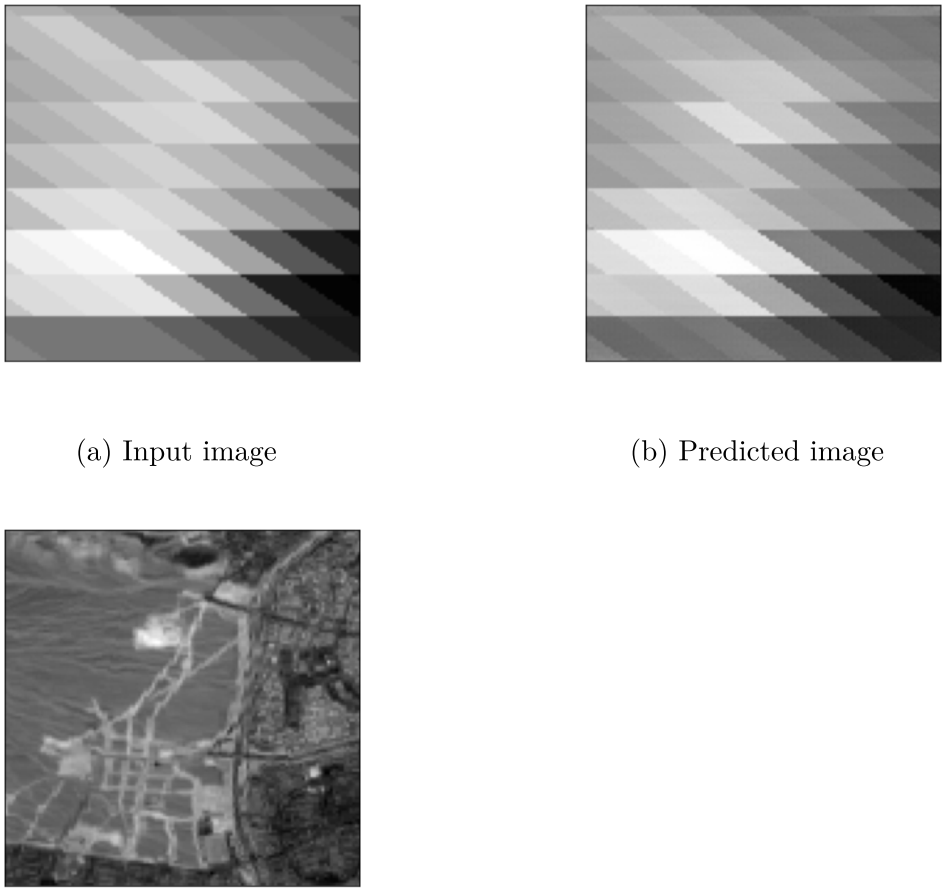
We see that all the hybrid methods but the naive architecture outperformed the base models across all metrics but SSIM and that the hybrid models had relatively similar performance (again except the naive architecture). The naive approach was the lowest performer across all metrics,

6.0.2 Predicted Imagery Samples

We further provide sample predictions from each of the base and hybrid models.

6.0.2.1 Pix2pix

We see that the pix2pix model had a hard time predicting high resolution imagery, as noted previously to be the challenge with satellite sensor fusion. The model was not able to learn how to accurately synthesize high resolution imagery and the predictions are visually similar to the MODIS data instead of the Landsat 8 data.



(a) Input image

(b) Predicted image

(c) True image

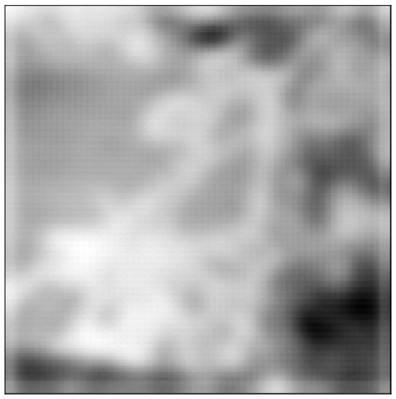
Figure 52: Sample pix2pix evaluation

6.0.2.2 FLAVR

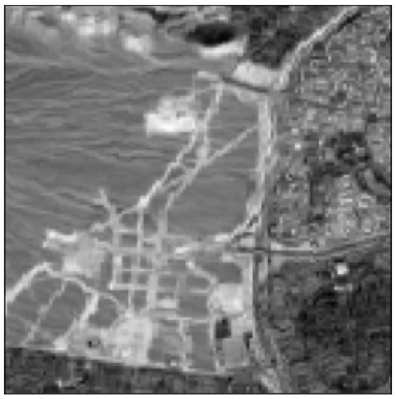
The FLAVR base model also had difficulty in generating crisp imagery, but we see that its prediction is much more visually similar to the true value in comparison to the pix2pix, which is supports the results from Table 5. We further notice artifacts on the edges of the predicted image in the form of light highlights.

(a) Prior image

(b) Future image



(c) Predicted image

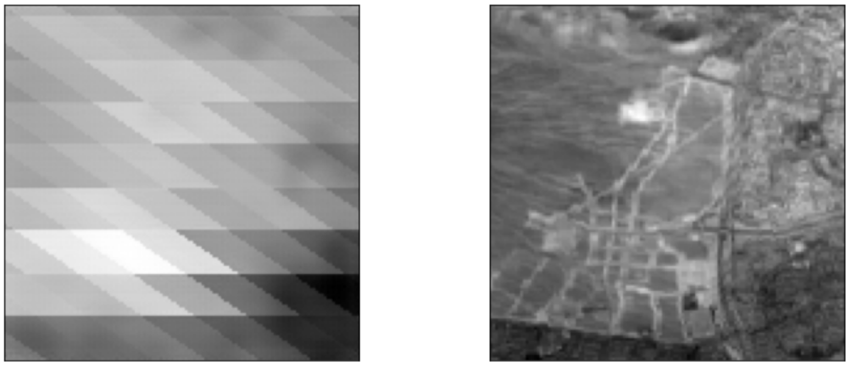


(d) True image

Figure 53: Sample FLAVR evaluation

6.0.2.3 Naive

The naive approach is a simple combination of the base models. Its prediction is like that of the pix2pix model, but with an additional blur from the FLAVR prediction. We see that the averaging of the two methods did not create clearer images than either base method.

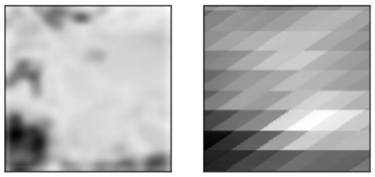


(a) Predicted image (b) True image

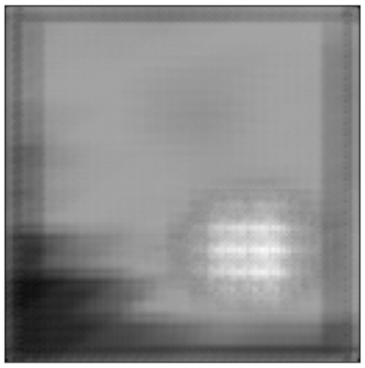
Figure 54: Sample naive evaluation

6.0.2.4 Secondary Fusion

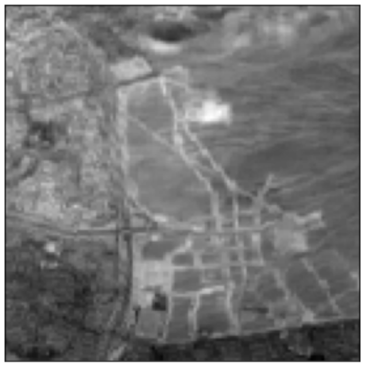
The prediction from the secondary model is surprising in that it does not really resemble anything seen so far. The images are blurry, somewhat pixelated, and have light and dark blotches on them. These justify the relatively low SSIM performance for the model in comparison to the other ones.



(a) Input image, where each sub-image is a channel



(b) Predicted image

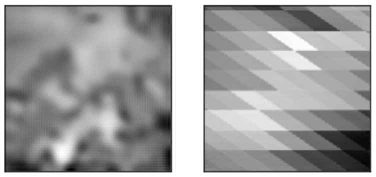


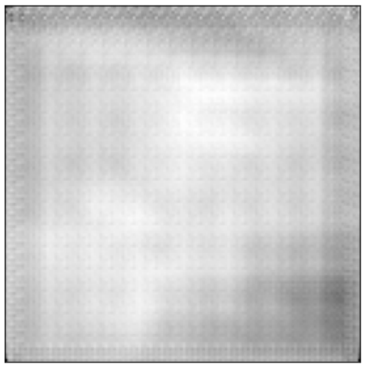
(c) True image

Figure 55: Sample secondary fusion evaluation

6.0.2.5 HR-Boosting

The predictions of the HR-boosting model exhibit similar blurring and pixelation to the secondary fusion method. The images are less patchy then the secondary fusion and have rough outlines of like the structures in the true image.


(a) Input image, where each sub-image is a channel


(b) Predicted image

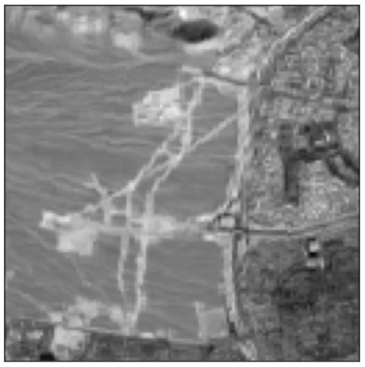

(c) True image

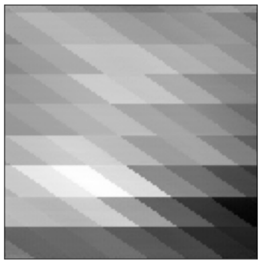
Figure 56: Sample HR-boosting evaluation

6.0.2.6 HT-Injection (with pix2pix)

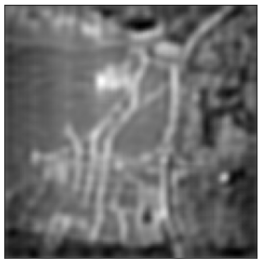
Images from the HT-Injection with the pix2pix prediction used as the injected input resemble a blurry image that seems to resemble the underlying physicality of the land well. They are less blurry than the FLAVR predictions, yet the predictions appear much darker than the true images.

(a) Prior image

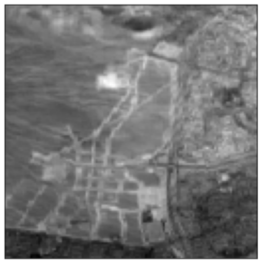
(b) Future image



(c) Injected image



(d) Predicted image



(e) True image

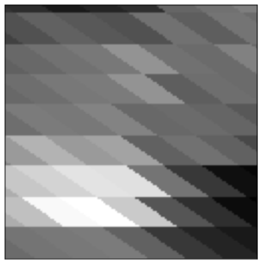
Figure 57: Sample HT-injection with injected pix2pix evaluation

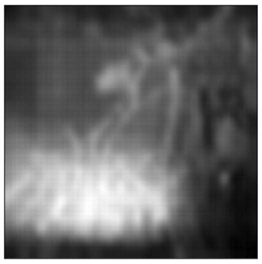
6.0.2.7 End-to-End HR-Injection

Images from the End-to-End HT-Injection that injected MODIS imagery into the injFLAVR resulted in what appears to be the combination of the Secondary fusion method and the HT-Injection with the pix2pix prediction injected, we see an outline of the land and physical structures as well as white blobs and dark areas.

(a) Prior image

(b) Future image


(c) Injected image


(d) Predicted image

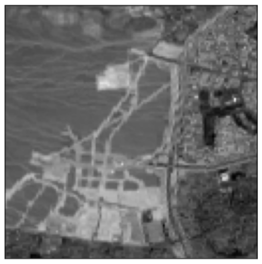

(e) True image

Figure 58: Sample end-to-end HT-injection evaluation

7 Conclusion

Current remote sensing is in dire need of high spatial and high temporal resolution imagery to generate flood inundation maps with sufficient spatial and temporal resolution. The field mainly utilizes two methodologies for synthesizing such imagery, satellite sensor fusion and video frame interpolation. We have presented a variety of architectures to combine the two methodologies and demonstrated the benefits of simultaneously using both temporal and cross-sensor information while synthesizing satellite imagery. Furthermore, we have made progress in reducing the complexity of such combined architectures, showing that pipelines with as few as a single model achieve comparable results to pipelines that separately perform video frame interpolation, satellite sensor fusion, and the synthesis of the predictions of the two.

8 Future Work

We propose several approaches to extend this work:

- Use within a flood-inundation mapping tool - while this work has shown promise in synthesizing high spatial high temporal satellite imagery, the overall aim is the production of high accuracy flood inundation maps. The predictions from the model are meant to be used as an input to a flood-mapping tool.

- Expand the custom dataset to incorporate more spatial regions - this will enable models to generalize better to new environments and provide more robust predictions.

- Remove clouds - while leaving clouds within the dataset allows for the creation of models that are less susceptible to clouds, doing so also effectively adds noise to the data and may reduce the effectiveness of models trained using it.

- Evaluate different normalization techniques - while this work used the division of an image by its maximum value for normalization, other techniques could be better suited for remote sensing. Clouds and other noise may influence the maximum value of the image and thus the resulting normalized image. We hypothesize that using a form of clipping by percentile to remove outlier pixel values followed by a normalization technique may result in better performance.

- Test out different architectures for the base models - while the choices for baselines were justified, the references for the justifications lie in the domain of 3-band (RGB) images as opposed to multi-spectral images. It is possible that some of the methods that performed worse on RGB images would perform better on multi-spectral images, so an investigation should be performed to evaluate that.

- Tune base models more - we approached the composition of the hybrid models with little tuning to the hyperparameters of the base models. More comprehensive tuning of these models will improve the quality of the inputs to some of the hybrid architectures (all models but the end-to-end injFLAVR) and may improve the overall performance of these hybrid architectures.

- Use more of the capabilities of the FLAVR model - in this work we focused on single image interpolation, while FLAVR can do much more. It could be worthwhile to use FLAVR with different window sizes to get more images per evaluation, though this could potentially lead to interesting challenges with the current injection technique.

- Swap the base models with unsupervised learning-based models - the field of image-to-image translation has a large count of unsupervised learning-based methods [64–66]. Since multi-sensor temporal-based image synthesisis a similar task, there is high potential for unsupervised learning in multi-sensor temporal-based image synthesisas well.

- Add a GAN loss to the injFLAVR - it is well known that GANs do extremely well in creating imagery that is visually similar to the training set. Due to the artifacts in the injFLAVR predictions, we hypothesize that the addition of a GAN loss may further improve predictions.

- Perform satellite sensor fusion from MODIS to Landsat 8 at low resolution and inject the result into injFLAVR - we showed promising results for injecting high temporal resolution imagery into injFLAVR, and specifically when these images have been first transformed into the correct context (as the pix2pix model does). Since the injFLAVR first downsamples the injected images, there is no benefit to training a pix2pix model to generate high spatial resolution Landsat 8 images, low spatial resolution images should suffice.

References

[1] J. Chen, L. Wang, R. Feng, P. Liu, W. Han, and X. Chen, "Cyclegan-stf: Spatiotemporal fusion via cyclegan-based image generation," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 59, no. 7, pp. 5851–5865, 2021.

[2] T. Vandal, D. McDuff, W. Wang, A. Michaelis, and R. Nemani, "Spectral synthesis for satellite-to-satellite translation," 2020.

[3] H. Zhang, Y. Song, C. Han, and L. Zhang, "Remote sensing image spatiotemporal fusion using a generative adversarial network," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 59, no. 5, pp. 4273–4286, 2021.

[4] T. Vandal and R. Nemani, "Temporal interpolation of geostationary satellite imagery with task specific optical flow," 2020.

[5] UNISDR, "The human cost of weather related disasters 1995-2015," https://www.unisdr.org/2015/docs/climatechange/COP21_WeatherDisastersReport_2015_FINAL.pdf, 2015, accessed: 2021-08-17.

[6] A. V.-S. M. B. Hallegatte, S. and J. Rozenberg, *Unbreakable: Building the Resilience of the Poor in the Face of Natural Disasters.* The World Bank, 2016.

[7] N. G. Society, "flood," Oct 2012. [Online]. Available: https://www.nationalgeographic.org/encyclopedia/flood/

[8] B. Tellmab, "Nasa thp," accessed: 2021-08-18.

[9] J. B. Campbell and R. H. Wynne, *Introduction to Remote Sensing, 5th Ed.*, 2011.

[10] "Bird's eye viewfinder: 160 years of aerial photography," https://airandspace.si.edu/stories/editorial/birds-eye-viewfinder-160-years-aerial-photography, accessed: 2021-07-27.

[11] J. W. Black, "Boston, as the eagle and the wild goose see it," 1860, [Online; accessed Oct 1, 2021]. [Online]. Available: https://collectionapi.metmuseum.org/api/collection/v1/iiif/283189/604165/main-image

[12] "Milestones in the history of remote sensing," https://nature.berkeley.edu/~penggong/textbook/chapter1/html/sect12.htm, accessed: 2021-07-30.

[13] "Lesson : The electromagnetic spectrum." [Online]. Available: https://dashboard.dublinschools.net/lessons/?id=d00226c67d475300b9afc56a5ff7a9fc&amp;v=1

[14] N. R. Canada, "Government of canada," Nov 2015. [Online]. Available: https://www.nrcan.gc.ca/maps-tools-publications/satellite-imagery-air-photos/remote-sensing-tutorials/introduction/passive-vs-active-sensing/14639

[15] M. Stefano, "Active pixel sensor vs ccd. who is the clear winner?" [Online]. Available: https://meroli.web.cern.ch/lecture_cmos_vs_ccd_pixel_sensor.html

[16] "Vision and optical instruments," https://courses.lumenlearning.com/physics/chapter/26-6-aberrations/, accessed: 2021-08-10.

[17] GDAL/OGR contributors, *GDAL/OGR Geospatial Data Abstraction software Library*, Open Source Geospatial Foundation, 2021. [Online]. Available: https://gdal.org

[18] S. Gillies *et al.*, "Rasterio: geospatial raster i/o for Python programmers," Mapbox, 2013–. [Online]. Available: https://github.com/mapbox/rasterio

[19] "Landsat science," https://landsat.gsfc.nasa.gov/, accessed: 2021-08-17.

[20] N. Gorelick, M. Hancher, M. Dixon, S. Ilyushchenko, D. Thau, and R. Moore, "Google earth engine: Planetary-scale geospatial analysis for everyone," *Remote Sensing of Environment*, 2017. [Online]. Available: https://doi.org/10.1016/j.rse.2017.06.031

[21] "Modis about," https://modis.gsfc.nasa.gov/about/, accessed: 2021-08-09.

[22] "Modis data," https://modis.gsfc.nasa.gov/data/, accessed: 2021-08-09.

[23] A. M. Tekalp, *Digital Video Processing*, 1995.

[24] E. Dubois, "Video sampling and interpolation," https://www.site.uottawa.ca/~edubois/courses/ELG5378/interp.pdf, accessed: 2021-08-10.

[25] M. Khoury, "A fuzzy probabilistic inference methodology for constrained 3d human motion classification," 01 2010.

[26] D. M. Tamir, "What is machine learning?" https://ischoolonline.berkeley.edu/blog/what-is-machine-learning/, 2015, accessed: 2021-08-23.

[27] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[28] 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," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[29] I. G. Y. Bengio and A. Courville, "Deep learning," 2016, book in preparation for MIT Press. [Online]. Available: http://www.deeplearningbook.org

[30] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[31] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," 2018.

[32] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier networks," in *Proceedings of the 14th International Conference on Artificial*

Intelligence and Statistics. JMLR W&CP Volume, vol. 15, 2011, pp. 315–323.

[33] G. E. Dahl, T. N. Sainath, and G. E. Hinton, “Improving deep neural networks for lvcsr using rectified linear units and dropout,” in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on.* IEEE, 2013, pp. 8609–8613.

[34] A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier nonlinearities improve neural network acoustic models,” in *Proc. ICML*, vol. 30, no. 1, 2013.

[35] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[36] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, “On the properties of neural machine translation: Encoder-decoder approaches,” 2014.

[37] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” 2014.

[38] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” 2014.

[39] Z. Ye, H. Wu, J. Jia, Y. Bu, W. Chen, F. Meng, and Y. Wang, “Choreonet: Towards music to dance synthesis with choreographic action unit,” *Proceedings of the 28th ACM International Conference on Multimedia*, Oct 2020. [Online]. Available: http://dx.doi.org/10.1145/3394171.3414005

[40] Y. Duan, T. Shi, Z. Zou, J. Qin, Y. Zhao, Y. Yuan, J. Hou, X. Wen, and C. Fan, “Semi-supervised learning for in-game expert-level music-to-dance translation,” 2020.

[41] N. Yalta, S. Watanabe, K. Nakadai, and T. Ogata, “Weakly supervised deep recurrent neural networks for basic dance step generation,” 2019.

[42] J. Lee, S. Kim, and K. Lee, “Listen to dance: Music-driven choreography generation using autoregressive encoder-decoder network,” *ArXiv*, vol. abs/1811.00818, 2018.

[43] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” 2015.

[44] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial networks,” 2014.

[45] M. Mirza and S. Osindero, “Conditional generative adversarial nets,” 2014.

[46] F. A. Reda, D. Sun, A. Dundar, M. Shoeybi, G. Liu, K. J. Shih, A. Tao, J. Kautz, and B. Catanzaro, “Unsupervised video interpolation using cycle consistency,” 2021.

[47] Moreno–Martínez, M. Moneta, G. C. Valls, L. Martino, N. Robinson, B. Allred, and S. W. Running, “Interpolation and gap filling of landsat reflectance time series,” in *IGARSS 2018 - 2018 IEEE International Geoscience and Remote Sensing Symposium*, 2018, pp. 349–352.

[48] F. Gao, J. Masek, M. Schwaller, and F. Hall, “On the blending of the landsat and modis surface reflectance: predicting daily landsat surface reflectance,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. 44, no. 8, pp. 2207–2218, 2006.

[49] S. Meyer, O. Wang, H. Zimmer, M. Grosse, and A. Sorkine-Hornung, “Phase-based frame interpolation for video,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1410–1418.

[50] S. Meyer, A. Djelouah, B. McWilliams, A. Sorkine-Hornung, M. Gross, and C. Schroers, “Phasenet for video frame interpolation,” 2018.

[51] H. Jiang, D. Sun, V. Jampani, M.-H. Yang, E. Learned-Miller, and J. Kautz, “Super slomo: High quality estimation of multiple intermediate frames for video interpolation,” 2018.

[52] S. Niklaus, L. Mai, and F. Liu, “Video frame interpolation via adaptive convolution,” 2017.

[53] ——, “Video frame interpolation via adaptive separable convolution,” 2017.

[54] Z. Liu, R. A. Yeh, X. Tang, Y. Liu, and A. Agarwala, “Video frame synthesis using deep voxel flow,” 2017.

[55] S. Niklaus and F. Liu, “Context-aware synthesis for video frame interpolation,” 2018.

[56] W. Bao, W.-S. Lai, C. Ma, X. Zhang, Z. Gao, and M.-H. Yang, “Depth-aware video frame interpolation,” 2019.

[57] W. Bao, W.-S. Lai, X. Zhang, Z. Gao, and M.-H. Yang, “Memc-net: Motion estimation and motion compensation driven neural network for video interpolation and enhancement,” 2019.

[58] X. Xu, L. Siyao, W. Sun, Q. Yin, and M.-H. Yang, “Quadratic video interpolation,” 2019.

[59] Y. Liu, L. Xie, L. Siyao, W. Sun, Y. Qiao, and C. Dong, “Enhanced quadratic video interpolation,” 2020.

[60] Y.-L. Liu, Y.-T. Liao, Y.-Y. Lin, and Y.-Y. Chuang, “Deep video frame interpolation using cyclic frame generation,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 8794–8802, Jul. 2019. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/4905

[61] T. Kalluri, D. Pathak, M. Chandraker, and D. Tran, “Flavr: Flow-agnostic video representations for fast frame interpolation,” 2021.

[62] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, “Image-to-image translation with conditional adversarial networks,” 2018.

[63] T.-C. Wang, M.-Y. Liu, J.-Y. Zhu, A. Tao, J. Kautz, and B. Catanzaro, “High-resolution image synthesis and semantic manipulation with conditional gans,” 2018.

[64] Y. Taigman, A. Polyak, and L. Wolf, “Unsupervised cross-domain image generation,” 2016.

[65] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, “Unpaired image-to-image translation using cycle-consistent adversarial networks,” 2020.

[66] T. Kim, M. Cha, H. Kim, J. K. Lee, and J. Kim, “Learning to discover cross-domain relations with generative adversarial networks,” 2017.

[67] J.-Y. Zhu, R. Zhang, D. Pathak, T. Darrell, A. A. Efros, O. Wang, and E. Shechtman, “Toward multimodal image-to-image translation,” 2018.

[68] X. Huang, M.-Y. Liu, S. Belongie, and J. Kautz, “Multimodal unsupervised image-to-image translation,” 2018.

[69] Y. Choi, M. Choi, M. Kim, J.-W. Ha, S. Kim, and J. Choo, "Stargan: Unified generative adversarial networks for multi-domain image-to-image translation," 2018.

[70] A. B. L. Larsen, S. K. Sønderby, H. Larochelle, and O. Winther, "Autoencoding beyond pixels using a learned similarity metric," 2016.

[71] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[72] A. Khalel, "Sewar," https://github.com/andrewekhalel/sewar, 2021.

[73] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors, "scikit-image: image processing in Python," *PeerJ*, vol. 2, p. e453, 6 2014. [Online]. Available: https://doi.org/10.7717/peerj.453

[74] C. Yim and A. C. Bovik, "Quality assessment of deblocked images," *IEEE Transactions on Image Processing*, vol. 20, no. 1, pp. 88–98, 2011.

[75] F. Fontaine, "Lecture notes: Waves and phasors," January 2020.

[76] "Modis specifications," https://modis.gsfc.nasa.gov/about/specifications.php#1, accessed: 2021-08-09.

A Electromagnetic Wave Equation

To obtain wave equations from Maxwell's equations, we follow [75] and first assume linear homogeneous isotropic material, of permeability μ and permittivity ϵ. This implies:

$$\vec{D} = \epsilon\vec{E},\ \vec{B} = \mu\vec{H} \tag{79}$$

Using this with Equations 1 - 4 results in the following:

$$\nabla\times\vec{E} = -\mu\frac{\partial\vec{H}}{\partial t} \tag{80}$$

$$\nabla\times\vec{H} = \vec{J} + \epsilon\frac{\partial\vec{E}}{\partial t} \tag{81}$$

$$\nabla\cdot\vec{E} = \rho/\epsilon \tag{82}$$

$$\nabla\cdot\vec{H} = 0 \tag{83}$$

Then:

$$\nabla\times(\nabla\times\vec{E}) = -\mu\frac{\partial}{\partial t}\nabla\times\vec{H}$$

Applying identities leads to:

$$\nabla\cdot(\nabla\cdot\vec{E}) - (\nabla^2\vec{E}) = -\mu\frac{\partial}{\partial t}\nabla\times\vec{H}$$

Substituting Equation 81 for $\nabla\times\vec{H}$ and Equation 82 for $\nabla\cdot\vec{E}$ leads to:

$$\nabla\rho/\epsilon - (\nabla^2\vec{E}) = -\mu\frac{\partial\vec{J}}{\partial t} - \mu\epsilon\frac{\partial^2\vec{E}}{\partial t^2}$$

Rearranging the equation and with $v = \frac{1}{\sqrt{\mu\epsilon}}$ we get an inhomogeneous wave equation:

$$\nabla^2\vec{E} - \frac{1}{v^2}\frac{\partial^2\vec{E}}{\partial t^2} = \nabla\rho/\epsilon - \mu\frac{\partial\vec{J}}{\partial t} \tag{84}$$

We note that similar derivation using equation 81 leads to another wave equation:

$$\nabla^2\vec{H} - \frac{1}{v^2}\frac{\partial^2\vec{H}}{\partial t^2} = -\nabla\times\vec{J} \tag{85}$$

With the additional assumption that there are no sources (namely $\rho = 0$ and $\vec{J} = 0$) we get homogeneous wave equations:

$$\nabla^2 \vec{E} - \frac{1}{v^2} \frac{\partial^2 \vec{E}}{\partial t^2} = 0 \tag{86}$$

$$\nabla^2 \vec{H} - \frac{1}{v^2} \frac{\partial^2 \vec{H}}{\partial t^2} = 0 \tag{87}$$

B Satellite Band Specifications

The following pages contain the band specifications for the satellites used throughout this work.

Band	Bandwidth
1	0.435-0.451 μm
2	0.452-0.512 μm
3	0.533-0.590 μm
4	0.636-0.673 μm
5	0.851-0.879 μm
6	1.566-1.651 μm
7	2.107-2.294 μm
8	0.500–0.680 μm
9	1.360–1.390 μm
10	10.6-11.2 μm
11	11.5-12.5 μm

Table 6: Landsat 8 Band Specifications [19]

Band	Bandwidth
1	620 - 670 nm
2	841 - 876 nm

Band	Bandwidth
3	459 - 479 nm
4	545 - 565 nm
5	1230 - 1250 nm
6	1628 - 1652 nm
7	2105 - 2155 nm
8	405 - 420 nm
9	438 - 448 nm
10	483 - 493 nm
11	526 - 536 nm
12	546 - 556 nm
13	662 - 672 nm
14	673 - 683 nm
15	743 - 753 nm
16	862 - 877 nm
17	890 - 920 nm
18	931 - 941 nm
19	915 - 965 nm
20	3.660 - 3.840 μm
21	3.929 - 3.989 μm
22	3.929 - 3.989 μm
23	4.020 - 4.080 μm
24	4.433 - 4.498 μm
25	4.482 - 4.549 μm
26	1.360 - 1.390 μm
27	6.535 - 6.895 μm
28	7.175 - 7.475 μm

Band	Bandwidth
29	8.400 - 8.700 μm
30	9.580 - 9.880 μm
31	10.780 - 11.280 μm
32	11.770 - 12.270 μm
33	13.185 - 13.485 μm
34	13.485 - 13.785 μm
35	13.785 - 14.085 μm
36	14.085 - 14.385 μm

Table 7: MODIS Band Specifications [76]

C Code Sample

The remaining pages contain a representative sample of the code used for the presented experiments. We note that this work is extensively based on two open-sourced implementations of the base models. These were `https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix` and `https://github.com/tarun005/FLAVR` for pix2pix and FLAVR respectively. We further emphasize our extensive use rasterio [18] to modify these implementations to accept .tiff files, the format used to store the satellite imagery.

C.1 Pix2pix Modifications

C.1.1 Dataset Loader

```python
import os
from data.base_dataset import BaseDataset, get_params, get_transform
from data.image_folder import make_dataset
from PIL import Image
import rasterio
import torch
import numpy as np
from torchvision import transforms
import random

class AlignedDataset(BaseDataset):
    """A dataset class for paired image dataset.

    It assumes that the directory '/path/to/data/train' contains image pairs in the form of {A,B}.
    During test time, you need to prepare a directory '/path/to/data/test'.
    """

    def __init__(self, opt):
        """Initialize this dataset class.

        Parameters:
            opt (Option class) -- stores all the experiment flags; needs to be a subclass of BaseOptions
        """
        BaseDataset.__init__(self, opt)
        self.dir_AB = os.path.join(opt.dataroot, opt.phase)  # get the image directory
        self.AB_paths = sorted(make_dataset(self.dir_AB, opt.max_dataset_size))  # get image paths
        assert(self.opt.load_size >= self.opt.crop_size)   # crop_size should be smaller than the size of loaded image
        self.input_nc = self.opt.output_nc if self.opt.direction == 'BtoA' else self.opt.input_nc
        self.output_nc = self.opt.input_nc if self.opt.direction == 'BtoA' else self.opt.output_nc

    def __getitem__(self, index):
        """Return a data point and its metadata information.

        Parameters:
            index - - a random integer for data indexing

        Returns a dictionary that contains A, B, A_paths and B_paths
            A (tensor) - - an image in the input domain
            B (tensor) - - its corresponding image in the target domain
            A_paths (str) - - image paths
            B_paths (str) - - image paths (same as A_paths)
        """
        # read a image given a random integer index
        AB_path = self.AB_paths[index]
        with rasterio.open(AB_path) as src:
          AB = src.read()

        # split AB image into A and B
        c, h, w = AB.shape
        w2 = int(w / 2)
        A = AB[:, :, :w2]
        B = AB[:, :, w2:]

        A = torch.from_numpy(A.copy()).type(torch.FloatTensor)
        B = torch.from_numpy(B.copy()).type(torch.FloatTensor)

        return {'A': A, 'B': B,
 'A_paths': AB_path, 'B_paths': AB_path}

    def __len__(self):
        """Return the total number of images in the dataset."""
        return len(self.AB_paths)
```

C.1.2 Evaluation

```python
"""General-purpose test script for image-to-image translation.

Once you have trained your model with train.py, you can use this script to test the model.
It will load a saved model from '--checkpoints_dir' and save the results to '--results_dir'.

It first creates model and dataset given the option. It will hard-code some parameters.
It then runs inference for '--num_test' images and save results to an HTML file.

Example (You need to train models first or download pre-trained models from our website):
    Test a CycleGAN model (both sides):
        python test.py --dataroot ./datasets/maps --name maps_cyclegan --model cycle_gan

    Test a CycleGAN model (one side only):
        python test.py --dataroot datasets/horse2zebra/testA --name horse2zebra_pretrained --model test --no_dropout

    The option '--model test' is used for generating CycleGAN results only for one side.
    This option will automatically set '--dataset_mode single', which only loads the images from one set.
    On the contrary, using '--model cycle_gan' requires loading and generating results in both directions,
    which is sometimes unnecessary. The results will be saved at ./results/.
    Use '--results_dir <directory_path_to_save_result>' to specify the results directory.

    Test a pix2pix model:
        python test.py --dataroot ./datasets/facades --name facades_pix2pix --model pix2pix --direction BtoA

See options/base_options.py and options/test_options.py for more test options.
See training and test tips at: https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/blob/master/docs/tips.md
See frequently asked questions at: https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/blob/master/docs/qa.md
"""
import os
import rasterio
from options.test_options import TestOptions
from data import create_dataset
from models import create_model
from util.visualizer import save_images
from util import html


def main():
    opt = TestOptions().parse()  # get test options
    # hard-code some parameters for test
    opt.num_threads = 0   # test code only supports num_threads = 0
    opt.batch_size = 1    # test code only supports batch_size = 1
    opt.serial_batches = True  # disable data shuffling; comment this line if results on randomly chosen images are needed.
```

```python
    opt.no_flip = True    # no flip; comment this line if results on flipped images are needed.
    opt.display_id = -1   # no visdom display; the test code saves the results to a HTML file.
    opt.eval = True
    dataset = create_dataset(opt)  # create a dataset given opt.dataset_mode and other options
    model = create_model(opt)      # create a model given opt.model and other options
    model.setup(opt)               # regular setup: load and print networks; create schedulers
    if opt.load_iter > 0:  # load_iter is 0 by default
        web_dir = '{:s}_iter{:d}'.format(web_dir, opt.load_iter)
    # test with eval mode. This only affects layers like batchnorm and dropout.
    # For [pix2pix]: we use batchnorm and dropout in the original pix2pix. You can experiment it with and without eval() mode.
    # For [CycleGAN]: It should not affect CycleGAN as CycleGAN uses instancenorm without dropout.
    if opt.eval:
        model.eval()
    for i, data in enumerate(dataset):
        model.set_input(data)  # unpack data from data loader
        model.test()           # run inference
        visuals = model.get_current_visuals()  # get image results
        img_path = model.get_image_paths()     # get image paths

        # save images
        for idx, image in enumerate(model.real_B):
            img = image.cpu().numpy()
            # print(img.shape)
            with rasterio.open(f'./output/set{i:03}_true{idx}.tif','w', driver='GTiff', height=img.shape[1],
                                   width=img.shape[2], count=img.shape[0], dtype=img.dtype) as dst:
                dst.write(img)
        for idx, image in enumerate(model.fake_B):
            img = image.cpu().numpy()
            with rasterio.open(f'./output/set{i:03}_pred{idx}.tif','w', driver='GTiff', height=img.shape[1],
                                   width=img.shape[2], count=img.shape[0], dtype=img.dtype) as dst:
                dst.write(img)
        for idx, image in enumerate(model.real_A):
            img = image.cpu().numpy()
            with rasterio.open(f'./output/set{i:03}_img{idx}.tif','w', driver='GTiff', height=img.shape[1],
                                  width=img.shape[2], count=img.shape[0], dtype=img.dtype) as dst:
                dst.write(img)


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

C.2 FLAVR Modifications

C.2.1 Dataset Loader

```python
import os
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image
import random
import rasterio
from sklearn.model_selection import train_test_split

im_size = 128

def get_loc_paths(loc_dir:str, ic=None)-> list:
  loc_paths = list()
  for date in sorted(os.listdir(loc_dir)):
    date_path = os.path.join(loc_dir, date)

    if not os.path.isdir(date_path):
      continue
    date_images = list()
    for image in os.listdir(date_path):
      # if specified a single ic & if the name of the image
    matches the ic desired
      if ic is not None:
          if ic in image:
              date_images.append(os.path.join(date_path, image))
      else:
          # if we want all ics
          date_images.append(os.path.join(date_path,image))
    loc_paths.append(date_images)
  # remove empty parts
  loc_paths = [paths for paths in loc_paths if len(paths)!=0]
  return loc_paths

def get_train_test(data_root, set_length, test_frac=0.2, ic='modis
    ', random_state=None, shuffle=True):
  paths = get_loc_paths(data_root, ic)
  test_frac= test_frac*0.01 if test_frac>=1 else test_frac
  test_size = int(test_frac*len(paths))
  tr_idx, test_idx = train_test_split(list(range(len(paths)-
    set_length)), # all idx but last set_length-1
                                      test_size=test_size,
                                      random_state=random_state,
                                      shuffle=shuffle)
  return paths, tr_idx, test_idx

class SatelliteLoader(Dataset):
    def __init__(self, paths, idx, is_training, inter_frames=3,
    n_inputs=4, channels=None):
        """
        Creates a Vimeo Septuplet object.
        Inputs.
            data_root: Root path for the Vimeo dataset containing
    the sep tuples.
            is_training: Train/Test.
        """
        super().__init__()
        self.paths = paths
        self.idx = idx
        self.training = is_training
        self.channels = channels

        self.inter_frames = inter_frames
        self.n_inputs = n_inputs
        self.set_length = (n_inputs-1)*(inter_frames+1)+1 ## We
    require these many frames in total for interpolating `
    interFrames` number of
                                                    ## intermediate
    frames with `n_input` input frames.
        self.transforms = None
        if self.training:
            self.transforms =  transforms.Compose([
                transforms.RandomHorizontalFlip(0.5),
                transforms.RandomVerticalFlip(0.5),
                transforms.CenterCrop(im_size)
                #transforms.ColorJitter(0.05, 0.05, 0.05, 0.05),
                #transforms.ToTensor()
            ])

        else:
            self.transforms = transforms.CenterCrop(im_size) #256

    def __getitem__(self, index):
        # get the paths corresponding to the images needed from
    the index
        img_paths = [self.paths[i+self.idx[index]] for i in range(
    self.set_length)]

        # Load images as tensors
        images = list()
        for pth in img_paths:
          with rasterio.open(pth[0]) as src:
            img = torch.from_numpy(src.read(out_dtype='float')[:
    self.channels]).type(torch.FloatTensor)
            img = img/img.max()
            images.append(img)

        # apply transformations if training
        seed = random.randint(0, 2**32)
        images_ = []
        if self.training:
            for img_ in images:
                # Apply the same transformation by using the same seed
                random.seed(seed)
                images_.append(self.transforms(img_))
            # Random Temporal Flip
            if random.random() >= 0.5:
                images_ = images_[::-1]
        else:
            # ensure sizes match with a crop
            for img_ in images:
                # Apply the same transformation by using the same seed
                random.seed(seed)
                images_.append(self.transforms(img_))
        images = images_
        # pick out every inter_frame+1 images as inputs
        inp_images = [images[idx] for idx in range(0, self.set_length, self.inter_frames+1)]
        rem = self.inter_frames%2
        gt_images = [images[idx] for idx in range(self.set_length//2-self.inter_frames//2 , self.set_length//2+self.inter_frames//2+rem)]
        return inp_images, gt_images

    def __len__(self):
        return len(self.idx)

def get_loader(paths, idx, batch_size, shuffle, num_workers, is_training=True, inter_frames=3, n_inputs=4, channels=3):
    dataset = SatelliteLoader(paths, idx , is_training, inter_frames=inter_frames, n_inputs=n_inputs, channels=channels)
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle, num_workers=num_workers, pin_memory=True, drop_last=True)
```

C.2.2 Architecture

C.2.2.1 FLAVR

```
import math
import numpy as np
import importlib

import torch
import torch.nn as nn
import torch.nn.functional as F
from .resnet_3D import SEGating

channels = 3
def joinTensors(X1 , X2 , type="concat"):

    if type == "concat":
```

```python
        return torch.cat([X1 , X2] , dim=1)
    elif type == "add":
        return X1 + X2
    else:
        return X1


class Conv_2d(nn.Module):

    def __init__(self, in_ch, out_ch, kernel_size, stride=1, padding=0, bias=False, batchnorm=False):

        super().__init__()
        self.conv = [nn.Conv2d(in_ch, out_ch, kernel_size=kernel_size, stride=stride, padding=padding, bias=bias)]

        if batchnorm:
            self.conv += [nn.BatchNorm2d(out_ch)]

        self.conv = nn.Sequential(*self.conv)

    def forward(self, x):

        return self.conv(x)

class upConv3D(nn.Module):

    def __init__(self, in_ch, out_ch, kernel_size, stride, padding, upmode="transpose" , batchnorm=False):

        super().__init__()

        self.upmode = upmode

        if self.upmode=="transpose":
            self.upconv = nn.ModuleList(
                [nn.ConvTranspose3d(in_ch, out_ch, kernel_size=kernel_size, stride=stride, padding=padding),
                SEGating(out_ch)
                ]
            )

        else:
            self.upconv = nn.ModuleList(
                [nn.Upsample(mode='trilinear', scale_factor=(1,2,2), align_corners=False),
                nn.Conv3d(in_ch, out_ch , kernel_size=1 , stride=1),
                SEGating(out_ch)
                ]
            )

        if batchnorm:
            self.upconv += [nn.BatchNorm3d(out_ch)]

        self.upconv = nn.Sequential(*self.upconv)

    def forward(self, x):

        return self.upconv(x)

class Conv_3d(nn.Module):

    def __init__(self, in_ch, out_ch, kernel_size, stride=1, padding=0, bias=True, batchnorm=False):

        super().__init__()
        self.conv = [nn.Conv3d(in_ch, out_ch, kernel_size=kernel_size, stride=stride, padding=padding, bias=bias),
                    SEGating(out_ch)
                    ]

        if batchnorm:
            self.conv += [nn.BatchNorm3d(out_ch)]

        self.conv = nn.Sequential(*self.conv)

    def forward(self, x):

        return self.conv(x)

class upConv2D(nn.Module):

    def __init__(self, in_ch, out_ch, kernel_size, stride, padding, upmode="transpose" , batchnorm=False):

        super().__init__()

        self.upmode = upmode

        if self.upmode=="transpose":
            self.upconv = [nn.ConvTranspose2d(in_ch, out_ch, kernel_size=kernel_size, stride=stride, padding=padding)]

        else:
            self.upconv = [
                nn.Upsample(mode='bilinear', scale_factor=2, align_corners=False),
                nn.Conv2d(in_ch, out_ch , kernel_size=1 , stride=1)
            ]

        if batchnorm:
            self.upconv += [nn.BatchNorm2d(out_ch)]

        self.upconv = nn.Sequential(*self.upconv)

    def forward(self, x):

        return self.upconv(x)


class UNet_3D_3D(nn.Module):
    def __init__(self, block , n_inputs, n_outputs, batchnorm=False , joinType="concat" , upmode="transpose", channels=3):
        super().__init__()
        nf = [512 , 256 , 128 , 64]
        out_channels = channels*n_outputs
        self.joinType = joinType
        self.n_outputs = n_outputs
        self.channels = channels

        growth = 2 if joinType == "concat" else 1
        self.lrelu = nn.LeakyReLU(0.2, True)

        unet_3D = importlib.import_module(".resnet_3D_2" , "model")
        unet_3D.channels = channels
        if n_outputs > 1:
            unet_3D.useBias = True
        self.encoder = getattr(unet_3D , block)(pretrained=False , bn=batchnorm)

        self.decoder = nn.Sequential(
            Conv_3d(nf[0], nf[1] , kernel_size=3, padding=1, bias=True, batchnorm=batchnorm),
            upConv3D(nf[1]*growth, nf[2], kernel_size=(3,4,4), stride=(1,2,2), padding=(1,1,1) , upmode=upmode, batchnorm=batchnorm),
            upConv3D(nf[2]*growth, nf[3], kernel_size=(3,4,4), stride=(1,2,2), padding=(1,1,1) , upmode=upmode, batchnorm=batchnorm),
            Conv_3d(nf[3]*growth, nf[3] , kernel_size=3, padding=1, bias=True, batchnorm=batchnorm),
            upConv3D(nf[3]*growth , nf[3], kernel_size=(3,4,4), stride=(1,2,2), padding=(1,1,1) , upmode=upmode, batchnorm=batchnorm)
        )

        self.feature_fuse = Conv_2d(nf[3]*n_inputs , nf[3] , kernel_size=1 , stride=1, batchnorm=batchnorm)

        self.outconv = nn.Sequential(
            nn.ReflectionPad2d(3),
            nn.Conv2d(nf[3], out_channels , kernel_size=7 , stride=1, padding=0)
        )

    def forward(self, images):

        images = torch.stack(images , dim=2)

        ## Batch mean normalization works slightly better than global mean normalization, thanks to https://github.com/myungsub/CAIN
        mean_ = images.mean(2, keepdim=True).mean(3, keepdim=True).mean(4,keepdim=True)
        images = images-mean_

        x_0 , x_1 , x_2 , x_3 , x_4 = self.encoder(images)

        dx_3 = self.lrelu(self.decoder[0](x_4))
        dx_3 = joinTensors(dx_3 , x_3 , type=self.joinType)

        dx_2 = self.lrelu(self.decoder[1](dx_3))
        dx_2 = joinTensors(dx_2 , x_2 , type=self.joinType)

        dx_1 = self.lrelu(self.decoder[2](dx_2))
        dx_1 = joinTensors(dx_1 , x_1 , type=self.joinType)

        dx_0 = self.lrelu(self.decoder[3](dx_1))
        dx_0 = joinTensors(dx_0 , x_0 , type=self.joinType)

        dx_out = self.lrelu(self.decoder[4](dx_0))
        dx_out = torch.cat(torch.unbind(dx_out , 2) , 1)

        out = self.lrelu(self.feature_fuse(dx_out))
        out = self.outconv(out)

        out = torch.split(out, dim=1, split_size_or_sections=self.channels)
        mean_ = mean_.squeeze(2)
        out = [o+mean_ for o in out]

        return out
```

C.2.2.2 Resnet

```
# Modified from https://github.com/pytorch/vision/tree/master/torchvision/models/video

import torch
import torch.nn as nn

__all__ = ['unet_18', 'unet_34']

useBias = False
channels = 3

class identity(nn.Module):

    def __init__(self , *args , **kwargs):
        super().__init__()

    def forward(self , x):
        return x
```

```python

class Conv3DSimple(nn.Conv3d):
    def __init__(self,
                 in_planes,
                 out_planes,
                 midplanes=None,
                 stride=1,
                 padding=1):

        super(Conv3DSimple, self).__init__(
            in_channels=in_planes,
            out_channels=out_planes,
            kernel_size=(3, 3, 3),
            stride=stride,
            padding=padding,
            bias=useBias)

    @staticmethod
    def get_downsample_stride(stride , temporal_stride):
        if temporal_stride:
            return (temporal_stride, stride, stride)
        else:
            return (stride , stride , stride)

class BasicStem(nn.Sequential):
    """The default conv-batchnorm-relu stem
    """
    def __init__(self):
        super().__init__(
            nn.Conv3d(channels, 64, kernel_size=(channels, 7, 7), stride=(1, 2, 2),
                padding=(channels//2, 3, 3), bias=useBias),
            batchnorm(64),
            nn.ReLU(inplace=False))


class Conv2Plus1D(nn.Sequential):

    def __init__(self,
                 in_planes,
                 out_planes,
                 midplanes,
                 stride=1,
                 padding=1):
        if not isinstance(stride , int):
            temporal_stride , stride , stride = stride
        else:
            temporal_stride = stride

        super(Conv2Plus1D, self).__init__(
            nn.Conv3d(in_planes, midplanes, kernel_size=(channels//2, 3, 3),
                      stride=(1, stride, stride), padding=(0, padding, padding),
                        bias=False),
            # batchnorm(midplanes),
            nn.ReLU(inplace=True),
            nn.Conv3d(midplanes, out_planes, kernel_size=(channels, 1, 1),
                        stride=(temporal_stride, 1, 1), padding=(padding, 0, 0),
                        bias=False))

    @staticmethod
    def get_downsample_stride(stride , temporal_stride):
        if temporal_stride:
            return (temporal_stride, stride, stride)
        else:
            return (stride , stride , stride)

class R2Plus1dStem(nn.Sequential):
    """R(2+1)D stem is different than the default one as it uses separated 3D convolution
    """
    def __init__(self):
        super().__init__(
            nn.Conv3d(channels, 45, kernel_size=(1, 7, 7),
                        stride=(1, 2, 2), padding=(0, 3, 3),
                        bias=False),
            batchnorm(45),
            nn.ReLU(inplace=True),
            nn.Conv3d(45, 64, kernel_size=(channels, 1, 1),
                        stride=(1, 1, 1), padding=(1, 0, 0),
                        bias=False),
            batchnorm(64),
            nn.ReLU(inplace=True))


class SEGating(nn.Module):

    def __init__(self , inplanes , reduction=16):

        super().__init__()

        self.pool = nn.AdaptiveAvgPool3d(1)
        self.attn_layer = nn.Sequential(
            nn.Conv3d(inplanes , inplanes , kernel_size=1 , stride=1 , bias=True),
            nn.Sigmoid()
        )

    def forward(self , x):

        out = self.pool(x)
        y = self.attn_layer(out)
        return x * y

class BasicBlock(nn.Module):

    expansion = 1

    def __init__(self, inplanes, planes, conv_builder, stride=1,
   downsample=None):
        midplanes = (inplanes * planes * 3 * 3 * 3) // (inplanes *
    3 * 3 + 3 * planes)

        super(BasicBlock, self).__init__()
        self.conv1 = nn.Sequential(
            conv_builder(inplanes, planes, midplanes, stride),
            batchnorm(planes),
            nn.ReLU(inplace=True)
        )
        self.conv2 = nn.Sequential(
            conv_builder(planes, planes, midplanes),
            batchnorm(planes)
        )
        self.fg = SEGating(planes) ## Feature Gating
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.conv2(out)
        out = self.fg(out)
        if self.downsample is not None:
            residual = self.downsample(x)

        out += residual
        out = self.relu(out)

        return out

class VideoResNet(nn.Module):

    def __init__(self, block, conv_makers, layers,
                 stem, zero_init_residual=False):
        """Generic resnet video generator.

        Args:
            block (nn.Module): resnet building block
            conv_makers (list(functions)): generator function for
   each layer
            layers (List[int]): number of blocks per layer
            stem (nn.Module, optional): Resnet stem, if None,
   defaults to conv-bn-relu. Defaults to None.
        """
        super(VideoResNet, self).__init__()
        self.inplanes = 64

        self.stem = stem()

        self.layer1 = self._make_layer(block, conv_makers[0], 64, layers[0], stride=1 )
        self.layer2 = self._make_layer(block, conv_makers[1], 128, layers[1], stride=2 , temporal_stride=1)
        self.layer3 = self._make_layer(block, conv_makers[2], 256, layers[2], stride=2 , temporal_stride=1)
        self.layer4 = self._make_layer(block, conv_makers[3], 512, layers[3], stride=1, temporal_stride=1)

        # init weights
        self._initialize_weights()

        if zero_init_residual:
            for m in self.modules():
                if isinstance(m, Bottleneck):
                    nn.init.constant_(m.bn3.weight, 0)

    def forward(self, x):
        x_0 = self.stem(x)
        x_1 = self.layer1(x_0)
        x_2 = self.layer2(x_1)
        x_3 = self.layer3(x_2)
        x_4 = self.layer4(x_3)
        return x_0 , x_1 , x_2 , x_3 , x_4

    def _make_layer(self, block, conv_builder, planes, blocks, stride=1, temporal_stride=None):
        downsample = None

        if stride != 1 or self.inplanes != planes * block.expansion:
            ds_stride = conv_builder.get_downsample_stride(stride, temporal_stride)
            downsample = nn.Sequential(
                nn.Conv3d(self.inplanes, planes * block.expansion,
                          kernel_size=1, stride=ds_stride, bias=False),
                batchnorm(planes * block.expansion)
            )
            stride = ds_stride

        layers = []
        layers.append(block(self.inplanes, planes, conv_builder, stride, downsample ))

        self.inplanes = planes * block.expansion
        for i in range(1, blocks):
            layers.append(block(self.inplanes, planes, conv_builder ))

        return nn.Sequential(*layers)

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv3d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',
                                        nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm3d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)


def _video_resnet(arch, pretrained=False, progress=True, **kwargs)
    :
    model = VideoResNet(**kwargs)
    ## TODO: Other 3D resnet models, like S3D, r(2+1)D.

    if pretrained:
        state_dict = load_state_dict_from_url(model_urls[arch],
                                              progress=progress)
        model.load_state_dict(state_dict)
    return model


def unet_18(pretrained=False, bn=False, progress=True, **kwargs):
    """
    Construct 18 layer Unet3D model as in
    https://arxiv.org/abs/1711.11248

    Args:
        pretrained (bool): If True, returns a model pre-trained on
     Kinetics-400
        progress (bool): If True, displays a progress bar of the
    download to stderr

    Returns:
        nn.Module: R3D-18 encoder
    """
    global batchnorm
    if bn:
        batchnorm = nn.BatchNorm3d
    else:
        batchnorm = identity

    return _video_resnet('r3d_18',
                         pretrained, progress,
                         block=BasicBlock,
                         conv_makers=[Conv3DSimple] * 4,
                         layers=[2, 2, 2, 2],
                         stem=BasicStem, **kwargs)

def unet_34(pretrained=False, bn=False, progress=True, **kwargs):
```

```python
    """
    Construct 34 layer Unet3D model as in
    https://arxiv.org/abs/1711.11248

    Args:
        pretrained (bool): If True, returns a model pre-trained on
     Kinetics-400
        progress (bool): If True, displays a progress bar of the
    download to stderr

    Returns:
        nn.Module: R3D-18 encoder
    """
    global batchnorm
    # bn = False
    if bn:
        batchnorm = nn.BatchNorm3d
    else:
        batchnorm = identity


    return _video_resnet('r3d_34',
                         pretrained, progress,
                         block=BasicBlock,
                         conv_makers=[Conv3DSimple] * 4,
                         layers=[3, 4, 6, 3],
                         stem=BasicStem, **kwargs)
```

C.2.3 Training

```python
import os
import sys
import time

import torch
import numpy as np
from tqdm import tqdm
from torch.utils.tensorboard import SummaryWriter

import config
import myutils
from loss import Loss
from torch.utils.data import DataLoader

def load_checkpoint(args, model, optimizer , path):
    print("loading checkpoint %s" % path)
    checkpoint = torch.load(path)
    args.start_epoch = checkpoint['epoch'] + 1
    model.load_state_dict(checkpoint['state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer'])
    lr = checkpoint.get("lr" , args.lr)
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr

```

```

##### Parse CmdLine Arguments #####
args, unparsed = config.get_args()
cwd = os.getcwd()
print(args)

save_loc = os.path.join(args.checkpoint_dir, "saved_models_final"
    , args.dataset , args.exp_name)
if not os.path.exists(save_loc):
    os.makedirs(save_loc)
opts_file = os.path.join(save_loc , "opts.txt")
with open(opts_file , "w") as fh:
    fh.write(str(args))


##### TensorBoard & Misc Setup #####
writer_loc = os.path.join(args.checkpoint_dir , 'tensorboard_logs_
    %s_final/%s' % (args.dataset , args.exp_name))
writer = SummaryWriter(writer_loc)

device = torch.device('cuda' if args.cuda else 'cpu')
torch.backends.cudnn.enabled = True
torch.backends.cudnn.benchmark = True

torch.manual_seed(args.random_seed)
if args.cuda:
    torch.cuda.manual_seed(args.random_seed)

if args.dataset == "vimeo90K_septuplet":
    from dataset.vimeo90k_septuplet import get_loader
    train_loader = get_loader('train', args.data_root, args.
    batch_size, shuffle=True, num_workers=args.num_workers)
    test_loader = get_loader('test', args.data_root, args.
    test_batch_size, shuffle=False, num_workers=args.num_workers)
elif args.dataset == "gopro":
    from dataset.GoPro import get_loader
    train_loader = get_loader(args.data_root, args.batch_size,
    shuffle=True, num_workers=args.num_workers, test_mode=False,
    interFrames=args.n_outputs, n_inputs=args.nbr_frame)
    test_loader = get_loader(args.data_root, args.batch_size,
    shuffle=False, num_workers=args.num_workers, test_mode=True,
    interFrames=args.n_outputs, n_inputs=args.nbr_frame)
elif args.dataset == "satellite":
    from dataset.Satellite_normalize import get_loader,
    get_train_test
    set_length = (args.nbr_frame-1)*(args.n_outputs+1)+1
    paths, tr_idx, test_idx = get_train_test(args.data_root,
    set_length, random_state=214, ic=args.ic)
    train_loader = get_loader(paths, tr_idx, args.batch_size,
    shuffle=True, num_workers=args.num_workers, is_training=True,
    inter_frames=args.n_outputs, n_inputs=args.nbr_frame, channels
    =args.channels)
    test_loader = get_loader(paths, test_idx, args.batch_size,
    shuffle=False, num_workers=args.num_workers, is_training=False
    , inter_frames=args.n_outputs, n_inputs=args.nbr_frame,
    channels=args.channels)
else:
    raise NotImplementedError


from model.FLAVR_arch_2 import UNet_3D_3D
print("Building model: %s"%args.model.lower())
model = UNet_3D_3D(args.model.lower() , n_inputs=args.nbr_frame,
    n_outputs=args.n_outputs, joinType=args.joinType, upmode=args.
    upmode, channels=args.channels)
model = torch.nn.DataParallel(model).to(device)

##### Define Loss & Optimizer #####
criterion = Loss(args)

## ToDo: Different learning rate schemes for different parameters
from torch.optim import Adam
optimizer = Adam(model.parameters(), lr=args.lr, betas=(args.beta1
    , args.beta2))
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
    mode='min', factor=0.5, patience=5, verbose=True)

def train(args, epoch):
    losses, psnrs, ssims = myutils.init_meters(args.loss)
    model.train()
    criterion.train()

    t = time.time()
    for i, (images, gt_image) in enumerate(train_loader):

        # Build input batch
        images = [img_.cuda() for img_ in images]
        gt = [gt_.cuda() for gt_ in gt_image]

        # Forward
        optimizer.zero_grad()
        out = model(images)

        out = torch.cat(out)
        gt = torch.cat(gt)

        loss, loss_specific = criterion(out, gt)

        # Save loss values
        for k, v in losses.items():
            if k != 'total':
                v.update(loss_specific[k].item())
        losses['total'].update(loss.item())

        loss.backward()
        optimizer.step()

        # Calc metrics & print logs
```

```python
        if i % args.log_iter == 0:
            myutils.eval_metrics(out, gt, psnrs, ssims)

            print('Train Epoch: {} [{}/{}]\tLoss: {:.6f}\tPSNR: {:.4f}'.format(
                epoch, i, len(train_loader), losses['total'].avg, psnrs.avg , flush=True))

            # Log to TensorBoard
            timestep = epoch * len(train_loader) + i
            writer.add_scalar('Loss/train', loss.data.item(), timestep)
            writer.add_scalar('PSNR/train', psnrs.avg, timestep)
            writer.add_scalar('SSIM/train', ssims.avg, timestep)
            writer.add_scalar('lr', optimizer.param_groups[-1]['lr'], timestep)

            # Reset metrics
            losses, psnrs, ssims = myutils.init_meters(args.loss)
            t = time.time()


def test(args, epoch):
    print('Evaluating for epoch = %d' % epoch)
    losses, psnrs, ssims = myutils.init_meters(args.loss)
    model.eval()
    criterion.eval()

    t = time.time()
    with torch.no_grad():
        for i, (images, gt_image) in enumerate(test_loader):

            images = [img_.cuda() for img_ in images]
            gt = [gt_.cuda() for gt_ in gt_image]

            out = model(images) ## images is a list of neighboring frames
            out = torch.cat(out)
            gt = torch.cat(gt)

            # Save loss values
            loss, loss_specific = criterion(out, gt)
            for k, v in losses.items():
                if k != 'total':
                    v.update(loss_specific[k].item())
            losses['total'].update(loss.item())

            # Evaluate metrics
            myutils.eval_metrics(out, gt, psnrs, ssims)

    # Print progress
    print("Loss: %f, PSNR: %f, SSIM: %f\n" %
          (losses['total'].avg, psnrs.avg, ssims.avg))

    # Save psnr & ssim
    save_fn = os.path.join(save_loc, 'results.txt')
    with open(save_fn, 'a') as f:
        f.write('For epoch=%d\t' % epoch)
        f.write("PSNR: %f, SSIM: %f\n" %
                (psnrs.avg, ssims.avg))

    # Log to TensorBoard
    timestep = epoch +1
    writer.add_scalar('Loss/test', loss.data.item(), timestep)
    writer.add_scalar('PSNR/test', psnrs.avg, timestep)
    writer.add_scalar('SSIM/test', ssims.avg, timestep)

    return losses['total'].avg, psnrs.avg, ssims.avg


""" Entry Point """
def main(args):

    if args.pretrained:
        ## For low data, it is better to load from a supervised
   pretrained model
        loadStateDict = torch.load(args.pretrained)['state_dict']
        modelStateDict = model.state_dict()

        for k,v in loadStateDict.items():
            if v.shape == modelStateDict[k].shape:
                print("Loading " , k)
                modelStateDict[k] = v
            else:
                print("Not loading" , k)

        model.load_state_dict(modelStateDict)
    print('Beginning to train:')
    best_psnr = 0
    for epoch in range(args.start_epoch, args.max_epoch):
        train(args, epoch)

        test_loss, psnr, _ = test(args, epoch)

        # save checkpoint
        is_best = psnr > best_psnr
        best_psnr = max(psnr, best_psnr)
        myutils.save_checkpoint({
            'epoch': epoch,
            'state_dict': model.state_dict(),
            'optimizer': optimizer.state_dict(),
            'best_psnr': best_psnr,
            'lr' : optimizer.param_groups[-1]['lr']
        }, save_loc, is_best, args.exp_name)

        # update optimizer policy
        scheduler.step(test_loss)

```

```python
if __name__ == "__main__":
    main(args)
```

C.2.4 Testing

```python
import os
import sys
import time
import copy
import shutil
import random
import pdb

import torch
import numpy as np
from tqdm import tqdm
import rasterio
import config
import myutils

from torch.utils.data import DataLoader

##### Parse CmdLine Arguments #####
#os.environ["CUDA_VISIBLE_DEVICES"]='7'
args, unparsed = config.get_args()
cwd = os.getcwd()

print(f'Using cuda: {args.cuda}')
device = torch.device('cuda' if args.cuda else 'cpu')

torch.manual_seed(args.random_seed)
if args.cuda:
    torch.cuda.manual_seed(args.random_seed)

if args.dataset == "vimeo90K_septuplet":
    from dataset.vimeo90k_septuplet import get_loader
    test_loader = get_loader('test', args.data_root, args.
   test_batch_size, shuffle=False, num_workers=args.num_workers)
elif args.dataset == "ucf101":
    from dataset.ucf101_test import get_loader
    test_loader = get_loader(args.data_root, args.test_batch_size,
     shuffle=False, num_workers=args.num_workers)
elif args.dataset == "gopro":
    from dataset.GoPro import get_loader
    test_loader = get_loader(args.data_root, args.test_batch_size,
     shuffle=False, num_workers=args.num_workers, test_mode=True,
   interFrames=args.n_outputs)
elif args.dataset == "satellite":
    from dataset.Satellite_normalize import get_loader,
   get_train_test
    set_length = (args.nbr_frame-1)*(args.n_outputs+1)+1
    print(f'Loading {args.ic} image collection')
    paths, tr_idx, test_idx = get_train_test(args.data_root,
   set_length, random_state=214, ic=args.ic)
```

```python
    print(f'Number of validation samples: {len(test_idx)}')
    train_loader = get_loader(paths, tr_idx, args.batch_size, shuffle=True, num_workers=args.num_workers, is_training=True, inter_frames=args.n_outputs, n_inputs=args.nbr_frame, channels=args.channels)
    test_loader = get_loader(paths, test_idx, args.batch_size, shuffle=False, num_workers=args.num_workers, is_training=False, inter_frames=args.n_outputs, n_inputs=args.nbr_frame, channels=args.channels)
else:
    raise NotImplementedError


from model.FLAVR_arch_2 import UNet_3D_3D
print("Building model: %s"%args.model.lower())
model = UNet_3D_3D(args.model.lower() , n_inputs=args.nbr_frame, n_outputs=args.n_outputs, joinType=args.joinType, upmode=args.upmode, channels=args.channels)

model = torch.nn.DataParallel(model).to(device)
print("#params" , sum([p.numel() for p in model.parameters()]))


def test(args):
    print(f'Saving image every {args.test_im_freq} images')
    time_taken = []
    losses, psnrs, ssims = myutils.init_meters(args.loss)
    model.eval()
    psnr_list = []
    with torch.no_grad():
        for i, (images, gt_image ) in enumerate(tqdm(test_loader)):
            images = [img_.cuda() for img_ in images]
            gt = [g_.cuda() for g_ in gt_image]

            torch.cuda.synchronize()
            start_time = time.time()
            out = model(images)
            # save images

            if i%int(args.test_im_freq) == 0:
                for idx, image in enumerate(images):
                    img = image.cpu().numpy()[0]
                    with rasterio.open(f'./output/set{i:03}_img{idx}.tif','w', driver='GTiff', height=img.shape[1],
                                       width=img.shape[2], count=img.shape[0], dtype=img.dtype) as dst:
                        dst.write(img)
                for idx, image in enumerate(gt):
                    img = image.cpu().numpy()[0]
                    with rasterio.open(f'./output/set{i:03}_true{idx}.tif','w', driver='GTiff', height=img.shape[1],
                                       width=img.shape[2], count=img.shape[0], dtype=img.dtype) as dst:
```

```
                        dst.write(img)
                for idx, image in enumerate(out):
                    img = image.cpu().numpy()[0]
                    with rasterio.open(f'./output/set{i:03}_pred{idx}.tif','w', driver='GTiff', height=img.shape[1],
                                      width=img.shape[2], count=img.shape[0], dtype=img.dtype) as dst:
                        dst.write(img)

            out = torch.cat(out)
            gt = torch.cat(gt)

            torch.cuda.synchronize()
            time_taken.append(time.time() - start_time)

            myutils.eval_metrics(out, gt, psnrs, ssims)

    print("PSNR: %f, SSIM: %fn" %(psnrs.avg, ssims.avg))
    print("Average Time, " , sum(time_taken)/len(time_taken))

    return psnrs.avg


""" Entry Point """
def main(args):

    assert args.load_from is not None

    model_dict = model.state_dict()
    model.load_state_dict(torch.load(args.load_from)["state_dict"], strict=True)
    test(args)


if __name__ == "__main__":
    main(args)
```

C.2.5 Evaluation

```
import os
import sys
import time
import copy
import shutil
import random
import pdb

import torch
import numpy as np
from tqdm import tqdm
import rasterio
import config
import myutils

```

```python
from torch.utils.data import DataLoader

##### Parse CmdLine Arguments #####
#os.environ["CUDA_VISIBLE_DEVICES"]='7'
args, unparsed = config.get_args()
cwd = os.getcwd()

print(f'Using cuda: {args.cuda}')
device = torch.device('cuda' if args.cuda else 'cpu')

torch.manual_seed(args.random_seed)
if args.cuda:
    torch.cuda.manual_seed(args.random_seed)

if args.dataset == "vimeo90K_septuplet":
    from dataset.vimeo90k_septuplet import get_loader
    test_loader = get_loader('test', args.data_root, args.test_batch_size, shuffle=False, num_workers=args.num_workers)
elif args.dataset == "ucf101":
    from dataset.ucf101_test import get_loader
    test_loader = get_loader(args.data_root, args.test_batch_size, shuffle=False, num_workers=args.num_workers)
elif args.dataset == "gopro":
    from dataset.GoPro import get_loader
    test_loader = get_loader(args.data_root, args.test_batch_size, shuffle=False, num_workers=args.num_workers, test_mode=True, interFrames=args.n_outputs)
elif args.dataset == "satellite":
    from dataset.Satellite_normalize import get_loader, get_train_test
    set_length = (args.nbr_frame-1)*(args.n_outputs+1)+1
    print(f'Loading {args.ic} image collection')
    paths, tr_idx, test_idx = get_train_test(args.data_root, set_length, random_state=214, ic=args.ic, shuffle=False)
    all_idx = tr_idx + test_idx
    print(f'Number of validation samples: {len(all_idx)}')
    all_loader = get_loader(paths, all_idx, args.batch_size, shuffle=False, num_workers=args.num_workers, is_training=False, inter_frames=args.n_outputs, n_inputs=args.nbr_frame, channels=args.channels)
    #test_loader = get_loader(paths, test_idx, args.batch_size, shuffle=False, num_workers=args.num_workers, is_training=False, inter_frames=args.n_outputs, n_inputs=args.nbr_frame, channels=args.channels)
else:
    raise NotImplementedError


from model.FLAVR_arch_2 import UNet_3D_3D
print("Building model: %s"%args.model.lower())
model = UNet_3D_3D(args.model.lower() , n_inputs=args.nbr_frame, n_outputs=args.n_outputs, joinType=args.joinType, upmode=args.upmode, channels=args.channels)

model = torch.nn.DataParallel(model).to(device)
print("#params" , sum([p.numel() for p in model.parameters()]))


def evaluate(args):
    print(f'Saving image every {args.test_im_freq} images')
    #time_taken = []
    losses, psnrs, ssims = myutils.init_meters(args.loss)
    model.eval()
    psnr_list = []
    with torch.no_grad():
        for i, (images, gt_image ) in enumerate(tqdm(all_loader)):
            images = [img_.cuda() for img_ in images]
            gt = [g_.cuda() for g_ in gt_image]

            torch.cuda.synchronize()
            #start_time = time.time()
            out = model(images)
            # save images

            for idx, image in enumerate(images):
                img = image.cpu().numpy()[0]
                with rasterio.open(f'./output/set{i:03}_img{idx}.tif','w', driver='GTiff', height=img.shape[1],
                               width=img.shape[2], count=img.shape[0], dtype=img.dtype) as dst:
                    dst.write(img)
            for idx, image in enumerate(gt):
                img = image.cpu().numpy()[0]
                with rasterio.open(f'./output/set{i:03}_true{idx}.tif','w', driver='GTiff', height=img.shape[1],
                               width=img.shape[2], count=img.shape[0], dtype=img.dtype) as dst:
                    dst.write(img)
            for idx, image in enumerate(out):
                img = image.cpu().numpy()[0]
                with rasterio.open(f'./output/set{i:03}_pred{idx}.tif','w', driver='GTiff', height=img.shape[1],
                              width=img.shape[2], count=img.shape[0], dtype=img.dtype) as dst:
                    dst.write(img)

            out = torch.cat(out)
            gt = torch.cat(gt)

            #torch.cuda.synchronize()
            #time_taken.append(time.time() - start_time)

            myutils.eval_metrics(out, gt, psnrs, ssims)

    print("PSNR: %f, SSIM: %fn" %(psnrs.avg, ssims.avg))
    #print("Average Time, " , sum(time_taken)/len(time_taken))
    print('Images saved')
    return
```

```python


""" Entry Point """
def main(args):

    assert args.load_from is not None

    model_dict = model.state_dict()
    model.load_state_dict(torch.load(args.load_from)["state_dict"]
    , strict=True)
    evaluate(args)


if __name__ == "__main__":
    main(args)
```

C.3 Naive

```python
import rasterio
import numpy as np
import os
import concurrent.futures as cp
import re
import argparse

import myutils
parser = argparse.ArgumentParser()
parser.add_argument('--dir1', type=str, help='Input dir 1')
parser.add_argument('--dir2', type=str, help='Input dir 2')
parser.add_argument('--out_dir', type=str, default='./average',
    help='Name of output dir')
parser.add_argument('--weight', type=float, default=0.5, help='
    dir1 image:dir2 image ratio in final average')
parser.add_argument('--key_word1', type=str, default='', help='Key
     word to select only some images from dir1')
parser.add_argument('--key_word2', type=str, default='', help='Key
     word to select only some images from dir2')
parser.add_argument('--normalize', action='store_true', help='
    Match normalized images')
parser.add_argument('--set_delta', type=int, help='Offset in set
    indices between the two directories', default=1)

args = parser.parse_args()

losses, psnrs, ssims = myutils.init_meters('1*L1')
## Match the two datasets
with cp.ProcessPoolExecutor() as ex:
  for image in sorted(os.listdir(args.dir1)):
    if args.key_word1 not in image:
      continue
    image_path = os.path.join(args.dir1, image)
    with rasterio.open(image_path, 'r') as src:
      img = src.read()
    if args.normalize:
      img=img/img.max()
    image_set = re.findall('\d+', image.split('_')[0])

    # get true image
    true_key = image_set[0]
    print('true key', true_key)
    current_set = [image for image in os.listdir(args.dir1) if true_key in image]
    true_path = [image for image in current_set if 'true' in image][0]
    with rasterio.open(os.path.join(args.dir1, true_path), 'r') as src:
      true = src.read()
    if args.normalize:
      true=true/true.max()

    # recover name for other directory
    paired_set = int(image_set[0])+args.set_delta
    pair_key = f'set{paired_set:03}'
    print('paired key', pair_key)

    matching_set = [image for image in os.listdir(args.dir2) if pair_key in image]
    matching_image = [image for image in matching_set if args.key_word2 in image][0]
    with rasterio.open(os.path.join(args.dir2, matching_image), 'r') as src:
      img2 = src.read()
    if args.normalize:
      img2=img2/img2.max()

    out_img = args.weight*img + (1-args.weight)*img2
    out_img = out_img/out_img.max()

    # myutils.eval_metrics(true, out_img, psnrs, ssims)

    print(os.path.join(args.out_dir, image))
    if not  os.path.exists(args.out_dir):
      os.makedirs(args.out_dir)
    with rasterio.open(os.path.join(args.out_dir, true_key+'pred'),'w', driver='GTiff',
                       height=out_img.shape[1],width=out_img.shape[2], count=out_img.shape[0], dtype=out_img.dtype) as dst:
                        dst.write(out_img)
    with rasterio.open(os.path.join(args.out_dir, true_key+'true'),'w', driver='GTiff',
                       height=true.shape[1],width=true.shape[2], count=true.shape[0], dtype=true.dtype) as dst:
                        dst.write(true)

print(f'psnr: {psnrs.avg}, ssim: {ssims.avg}')
```

C.4 HR Boosting and Secondary Fusion

C.4.1 Pairing Images

```python
import os
import numpy as np
import argparse
import rasterio
from concurrent.futures import ThreadPoolExecutor

def parse_args():
  parser = argparse.ArgumentParser('create image pairs')
  parser.add_argument('--inputA_dir', help='input directory for image A', type=str, default='../dataset/inputA')
  parser.add_argument('--inputB_dir', help='input directory for image B', type=str, default='../dataset/inputB')
  parser.add_argument('--inputA_keyword', help='input keyword selector for dir A', type=str, default='pred')
  parser.add_argument('--inputB_keyword', help='input keyword selector for dir B', type=str, default='pred')
  parser.add_argument('--inputB_dropN', help='number of inputs to drop from B to match lengths', type=int, default=1)
  parser.add_argument('--out_dir', help='output directory', type=str, default='../dataset/test_AB')
  parser.add_argument('--bands', help='number of bands to save', type=int, default=1)
  parser.add_argument('--im_size', help='size of each image', type=int, default=128)
  parser.add_argument('--split_val', action='store_true', help='create a validation dir')
  parser.add_argument('--normalize', action='store_true', help='normalize images first')
  args = parser.parse_args()
  return args

def center_crop(im, im_size):
  """
  Center crop image to im_size
  """
  im_size = im_size//2
  return im[:,(im.shape[1]//2-im_size):(im.shape[1]//2+im_size),(im.shape[2]//2-im_size):(im.shape[2]//2+im_size)]

def load_image(path, bands, size):
    """
    Use rastrio to open up tif file, extract the desired number of bands, and center crop to right size
    """
    with rasterio.open(path) as src:
      im = src.read()[:bands]
    im = center_crop(im, size)
    return im

```

```python
def image_write(path_A, path_B, path_AB, bands=1, im_size=128, normalize=True):
    """
    Get the images, concatenate them together, and save in the correct format
    """
    # get images
    im_A = load_image(path_A, bands, im_size)
    im_B = load_image(path_B, bands, im_size)

    if normalize:
      im_A = im_A/im_A.max()
      im_B = im_B/im_B.max()

    # concat and save
    im_AB = np.concatenate([im_A, im_B])
    save_image(im_AB, path_AB)

def save_image(array, path):
  """
  Save a numpy array as a tif file at a specified path
  """
  with rasterio.open(path, mode='w', driver='GTiff',
                       height=array.shape[1], width=array.shape[2],
                       count=array.shape[0], dtype=array.dtype) as src:
    src.write(array)

def read_write_image(inp_path, out_path, bands, im_size, normalize):
  im = load_image(inp_path, bands, im_size)
  if normalize:
    im = im/im.max()
  save_image(im, out_path)

def get_train_test(paths, test_frac=0.2, random_state=12345, shuffle=True):
    """
    Returns the training and validation paths
    :param data_root: root directory for images
    :param test_frac: fraction to make into a validation set
    :param random_state: random state
    :param shuffle: whether to shuffle the input paths
    :return: training paths, test paths (lists)
    """
    from sklearn.model_selection import train_test_split

    test_frac = test_frac * 0.01 if test_frac >= 1 else test_frac
    test_size = int(test_frac * len(paths))
    tr_paths, test_paths = train_test_split(paths,
                                            test_size=test_size,
                                            random_state=random_state,
                                              shuffle=shuffle)
    return tr_paths, test_paths




def main():
  args = parse_args()
  for arg in vars(args):
    print('[%s] = ' % arg, getattr(args, arg))

  # Get full paths to images in the directories
  trA = sorted([os.path.join(args.inputA_dir, path) for path in os
   .listdir(args.inputA_dir) if args.inputA_keyword in path])
  trB = sorted([os.path.join(args.inputB_dir, path) for path in os
   .listdir(args.inputB_dir) if args.inputB_keyword in path])
  trB = trB[args.inputB_dropN:]

  true = sorted([os.path.join(args.inputA_dir, path) for path in
   os.listdir(args.inputA_dir) if 'true' in path])

  if args.split_val:
    trA, valA = get_train_test(trA)
    trB, valB = get_train_test(trB)
    true, true_val = get_train_test(true)

  if not os.path.exists(args.out_dir+'/trainA'):
    print('Making train dir')
    os.makedirs(os.path.join(args.out_dir, 'trainA'))
    os.makedirs(os.path.join(args.out_dir, 'trainB'))

  path_pairs = [os.path.join(os.path.join(args.out_dir,'trainA'),
   f'sample{i:03}.tif') for i in range(len(trA))]
  path_true =  [os.path.join(os.path.join(args.out_dir,'trainB'),
   f'sample{i:03}.tif') for i in range(len(trA))]
  with ThreadPoolExecutor() as e:
    print('making trainA')
    for i, pth in enumerate(zip(trA, trB)):
      e.submit(image_write, pth[0], pth[1], path_pairs[i], bands=
   args.bands, im_size=args.im_size, normalize=args.normalize)
    print('making trainB')
    for pth, pth_true in zip(true, path_true):
      e.submit(read_write_image, pth, pth_true, bands=args.bands,
   im_size=args.im_size, normalize=args.normalize)

  if args.split_val:
    if not os.path.exists(args.out_dir+'/valA'):
      print('Making validation dir')
      os.mkdir(os.path.join(args.out_dir, 'valA'))
      os.mkdir(os.path.join(args.out_dir, 'valB'))

    path_pairs = [os.path.join(os.path.join(args.out_dir,'valA'),
   f'sample{i:03}.tif') for i in range(len(valA))]
```

```python
    path_true = [os.path.join(os.path.join(args.out_dir,'valB'), f'sample{i:03}.tif') for i in range(len(valA))]
    with ThreadPoolExecutor() as e:
      for i, pth in enumerate(zip(valA, valB)):
        e.submit(image_write, pth[0], pth[1], path_pairs[i], bands=args.bands, im_size=args.im_size, normalize=args.normalize)
      for pth, pth_true in zip(true_val, path_true):
        e.submit(read_write_image, pth, pth_true, bands=args.bands, im_size=args.im_size, normalize=args.normalize)

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

C.4.2 Dataset Loader

```python
import os
from data.base_dataset import BaseDataset, get_transform
from data.image_folder import make_dataset
#from PIL import Image
import numpy as np
import random
import rasterio

class UnalignedDataset(BaseDataset):
    """
    This dataset class can load unaligned/unpaired datasets.

    It requires two directories to host training images from domain A '/path/to/data/trainA'
    and from domain B '/path/to/data/trainB' respectively.
    You can train the model with the dataset flag '--dataroot /path/to/data'.
    Similarly, you need to prepare two directories:
    '/path/to/data/testA' and '/path/to/data/testB' during test time.
    """

    def __init__(self, opt):
        """Initialize this dataset class.

        Parameters:
            opt (Option class) -- stores all the experiment flags; needs to be a subclass of BaseOptions
        """
        BaseDataset.__init__(self, opt)
        self.dir_A = os.path.join(opt.dataroot, opt.phase + 'A')  # create a path '/path/to/data/trainA'
        self.dir_B = os.path.join(opt.dataroot, opt.phase + 'B')  # create a path '/path/to/data/trainB'

        self.A_paths = sorted(make_dataset(self.dir_A, opt.max_dataset_size))   # load images from '/path/to/data/trainA'
        self.B_paths = sorted(make_dataset(self.dir_B, opt.max_dataset_size))    # load images from '/path/to/data/trainB'
        #self.A_paths = sorted([path for path in self.A_paths if opt.keywordA in path])
        #self.B_paths = sorted([path for path in self.A_paths if opt.keywordB in path])
        self.A_size = len(self.A_paths)  # get the size of dataset A
        self.B_size = len(self.B_paths)  # get the size of dataset B
        btoA = self.opt.direction == 'BtoA'
        input_nc = self.opt.output_nc if btoA else self.opt.input_nc       # get the number of channels of input image
        output_nc = self.opt.input_nc if btoA else self.opt.output_nc      # get the number of channels of output image
        self.transform_A = get_transform(self.opt, grayscale=(input_nc == 1))
        self.transform_B = get_transform(self.opt, grayscale=(output_nc == 1))
        self.eval = opt.eval

    def __getitem__(self, index):
        """Return a data point and its metadata information.

        Parameters:
            index (int)      -- a random integer for data indexing

        Returns a dictionary that contains A, B, A_paths and B_paths
            A (tensor)       -- an image in the input domain
            B (tensor)       -- its corresponding image in the target domain
            A_paths (str)    -- image paths
            B_paths (str)    -- image paths
        """
        A_path = self.A_paths[index % self.A_size]  # make sure index is within then range
        if self.opt.serial_batches:   # make sure index is within then range
            index_B = index % self.B_size
        else:   # randomize the index for domain B to avoid fixed pairs.
            index_B = random.randint(0, self.B_size - 1)
        B_path = self.B_paths[index_B]
        with rasterio.open(A_path) as src:
          A = src.read()
        with rasterio.open(B_path) as src:
          B = src.read()

        # apply transformations if training
        if not self.eval:
          if np.random.uniform() < 0.5:
            A = np.fliplr(A)
            B = np.fliplr(B)
          if np.random.uniform() < 0.5:
            A = np.flip(A, axis=2)
            B = np.flip(B, axis=2)
          A = A.copy()
          B = B.copy()

        return {'A': A, 'B': B, 'A_paths': A_path, 'B_paths': B_path}

    def __len__(self):
        """Return the total number of images in the dataset.

        As we have two datasets with potentially different number of images,
        we take a maximum of
        """
        return max(self.A_size, self.B_size)
```

C.4.3 Tuning

```
"""General-purpose training script for image-to-image translation.

This script works for various models (with option '--model': e.g., pix2pix, cyclegan, colorization) and
different datasets (with option '--dataset_mode': e.g., aligned, unaligned, single, colorization).
You need to specify the dataset ('--dataroot'), experiment name ('--name'), and model ('--model').

It first creates model, dataset, and visualizer given the option.
It then does standard network training. During the training, it also visualize/save the images, print/save the loss plot, and save models.
The script supports continue/resume training. Use '--continue_train' to resume your previous training.

Example:
    Train a CycleGAN model:
        python train.py --dataroot ./datasets/maps --name maps_cyclegan --model cycle_gan
    Train a pix2pix model:
        python train.py --dataroot ./datasets/facades --name facades_pix2pix --model pix2pix --direction BtoA

See options/base_options.py and options/train_options.py for more training options.
See training and test tips at: https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/blob/master/docs/tips.md
See frequently asked questions at: https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/blob/master/docs/qa.md
"""
import time
from options.train_options import TrainOptions
```

```python
from data import create_dataset
from models import create_model
from util.visualizer import Visualizer
import myutils

import optuna

def test(model, test_ds):
    # test with eval mode. This only affects layers like batchnorm
     and dropout.
    model.eval()
    #initialize metrics
    losses, psnrs, ssims = myutils.init_meters('1*L1')
    for i, data in enumerate(test_ds):
        model.set_input(data)  # unpack data from data loader
        model.test()           # run inference
        visuals = model.get_current_visuals()  # get image results
        img_path = model.get_image_paths()     # get image paths
        myutils.eval_metrics(model.real_B, model.fake_B, psnrs,
    ssims)
    model.train()
    return [val.avg for val in losses.values()][0], psnrs.avg,
    ssims.avg

def objective(trial):
    opt = TrainOptions().parse()   # get training options

    ### Define Optuna parameters
    opt.n_layers_D = trial.suggest_int('n_layers_D', 3, 5)
    opt.lr = trial.suggest_float('lr', 1e-6, 1e-3)
    opt.gan_mode = trial.suggest_categorical('gan_mode', ['vanilla
    ', 'lsgan'])
    opt.lr_policy = trial.suggest_categorical('lr_policy', ['
    linear', 'step', 'cosine', 'plateau'])
    opt.lr_decay_iters = trial.suggest_int('lr_decay_iters', 40,
    70)
    opt.beta1 = trial.suggest_float('beta1', 1-1e-3, 1-1e-5)
    opt.batch_size = trial.suggest_categorical('batch_size', [1,
    2, 4, 8])
    print('trial paramaters: ', trial.params)

    # create datasets
    dataset = create_dataset(opt)  # create a dataset given opt.
    dataset_mode and other options
    dataset_size = len(dataset)    # get the number of images in
    the dataset.
    #print('The number of training images = %d' % dataset_size)

    opt.isTraining = False
    opt.num_threads = 0   # test code only supports num_threads =
    0
    opt.batch_size = 1    # test code only supports batch_size = 1
    opt.serial_batches = True  # disable data shuffling; comment
    this line if results on randomly chosen images are needed.
    opt.no_flip = True    # no flip; comment this line if results on flipped images are needed.
    opt.display_id = -1   # no visdom display; the test code saves the results to a HTML file.
    opt.phase = 'val'
    test_ds = create_dataset(opt)  # create a dataset given opt.dataset_mode and other options

    model = create_model(opt)      # create a model given opt.model and other options
    model.setup(opt)               # regular setup: load and print networks; create schedulers
    total_iters = 0                # the total number of training iterations

    for epoch in range(opt.epoch_count, opt.n_epochs + opt.n_epochs_decay + 1):    # outer loop for different epochs; we save the model by <epoch_count>, <epoch_count>+<save_latest_freq>
        epoch_start_time = time.time()  # timer for entire epoch
        iter_data_time = time.time()    # timer for data loading per iteration
        epoch_iter = 0                  # the number of training iterations in current epoch, reset to 0 every epoch
        model.update_learning_rate()    # update learning rates in the beginning of every epoch.
        model.epoch_loss_G = 0
        model.epoch_loss_D = 0
        for i, data in enumerate(dataset):  # inner loop within one epoch

            total_iters += opt.batch_size
            epoch_iter += opt.batch_size
            model.set_input(data)         # unpack data from dataset and apply preprocessing
            model.optimize_parameters()   # calculate loss functions, get gradients, update network weights

            l1, psnr, ssim = test(model, test_ds)
            #print(f'l1: {l1}, psnr {psnr}, ssim {ssim}')

        loss_G = model.epoch_loss_G/len(dataset)
        loss_D = model.epoch_loss_D/len(dataset)
        #print(f'Loss G: {loss_G}')
        #print(f'Loss D: {loss_D}')
        #print('End of epoch %d / %d \t Time Taken: %d sec' % (epoch, opt.n_epochs + opt.n_epochs_decay, time.time() - epoch_start_time))
    return psnr, ssim

def main():
  study = optuna.create_study(study_name='Pix2pixSecondary', directions=['maximize', 'minimize'])
  study.optimize(objective, n_trials=30)
  print('Best Parameters: ', study.best_trial.value)

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

C.5 HT Injection

C.5.1 injFLAVR

```
import math
import numpy as np
import importlib

import torch
import torch.nn as nn
import torch.nn.functional as F
from .resnet_3D import SEGating

channels = 3
def joinTensors(X1 , X2 , type="concat"):

    if type == "concat":
        return torch.cat([X1 , X2] , dim=1)
    elif type == "add":
        return X1 + X2
    else:
        return X1


class Conv_2d(nn.Module):

    def __init__(self, in_ch, out_ch, kernel_size, stride=1,
   padding=0, bias=False, batchnorm=False):

        super().__init__()
        self.conv = [nn.Conv2d(in_ch, out_ch, kernel_size=
   kernel_size, stride=stride, padding=padding, bias=bias)]

        if batchnorm:
            self.conv += [nn.BatchNorm2d(out_ch)]

        self.conv = nn.Sequential(*self.conv)

    def forward(self, x):

        return self.conv(x)

class upConv3D(nn.Module):

    def __init__(self, in_ch, out_ch, kernel_size, stride, padding
   , upmode="transpose" , batchnorm=False):

        super().__init__()
```

```python

        self.upmode = upmode

        if self.upmode=="transpose":
            self.upconv = nn.ModuleList(
                [nn.ConvTranspose3d(in_ch, out_ch, kernel_size=kernel_size, stride=stride, padding=padding),
                SEGating(out_ch)
                ]
            )

        else:
            self.upconv = nn.ModuleList(
                [nn.Upsample(mode='trilinear', scale_factor=(1,2,2), align_corners=False),
                nn.Conv3d(in_ch, out_ch , kernel_size=1 , stride=1),
                SEGating(out_ch)
                ]
            )

        if batchnorm:
            self.upconv += [nn.BatchNorm3d(out_ch)]

        self.upconv = nn.Sequential(*self.upconv)

    def forward(self, x):

        return self.upconv(x)

class Conv_3d(nn.Module):

    def __init__(self, in_ch, out_ch, kernel_size, stride=1, padding=0, bias=True, batchnorm=False):

        super().__init__()
        self.conv = [nn.Conv3d(in_ch, out_ch, kernel_size=kernel_size, stride=stride, padding=padding, bias=bias),
                    SEGating(out_ch)
                    ]

        if batchnorm:
            self.conv += [nn.BatchNorm3d(out_ch)]

        self.conv = nn.Sequential(*self.conv)

    def forward(self, x):

        return self.conv(x)

class upConv2D(nn.Module):

    def __init__(self, in_ch, out_ch, kernel_size, stride, padding, upmode="transpose" , batchnorm=False):
        super().__init__()

        self.upmode = upmode

        if self.upmode=="transpose":
            self.upconv = [nn.ConvTranspose2d(in_ch, out_ch,
    kernel_size=kernel_size, stride=stride, padding=padding)]

        else:
            self.upconv = [
                nn.Upsample(mode='bilinear', scale_factor=2,
    align_corners=False),
                nn.Conv2d(in_ch, out_ch , kernel_size=1 , stride
    =1)
            ]

        if batchnorm:
            self.upconv += [nn.BatchNorm2d(out_ch)]

        self.upconv = nn.Sequential(*self.upconv)

    def forward(self, x):

        return self.upconv(x)


class UNet_3D_3D(nn.Module):
    def __init__(self, block , n_inputs, n_outputs, batchnorm=
    False , joinType="concat" , upmode="transpose", channels=3):
        super().__init__()
        nf = [512 , 256 , 128 , 64]
        out_channels = channels*n_outputs
        self.joinType = joinType
        self.n_outputs = n_outputs
        self.channels = channels

        growth = 2 if joinType == "concat" else 1
        self.lrelu = nn.LeakyReLU(0.2, True)

        unet_3D = importlib.import_module(".resnet_3D_2" , "model"
    )
        unet_3D.channels = channels
        if n_outputs > 1:
            unet_3D.useBias = True
        self.encoder = getattr(unet_3D , block)(pretrained=False ,
     bn=batchnorm)

        self.decoder = nn.Sequential(
            Conv_3d(nf[0]+1, nf[1] , kernel_size=3, padding=1,
    bias=True, batchnorm=batchnorm),
            upConv3D(nf[1]*growth, nf[2], kernel_size=(3,4,4),
    stride=(1,2,2), padding=(1,1,1) , upmode=upmode, batchnorm=
    batchnorm),
```

```python
            upConv3D(nf[2]*growth, nf[3], kernel_size=(3,4,4), stride=(1,2,2), padding=(1,1,1) , upmode=upmode, batchnorm=batchnorm),
            Conv_3d(nf[3]*growth, nf[3] , kernel_size=3, padding=1, bias=True, batchnorm=batchnorm),
            upConv3D(nf[3]*growth , nf[3], kernel_size=(3,4,4), stride=(1,2,2), padding=(1,1,1) , upmode=upmode, batchnorm=batchnorm)
        )

        self.feature_fuse = Conv_2d(nf[3]*n_inputs , nf[3] , kernel_size=1 , stride=1, batchnorm=batchnorm)

        self.outconv = nn.Sequential(
            nn.ReflectionPad2d(3),
            nn.Conv2d(nf[3], out_channels , kernel_size=7 , stride=1, padding=0)
        )

        self.lr_injection = nn.MaxPool2d((8, 8))

    def forward(self, images, injection):

        images = torch.stack(images , dim=2)
        injection = torch.stack(injection)

        ## Batch mean normalization works slightly better than global mean normalization, thanks to https://github.com/myungsub/CAIN
        mean_ = images.mean(2, keepdim=True).mean(3, keepdim=True).mean(4,keepdim=True)
        images = images-mean_

        x_0 , x_1 , x_2 , x_3 , x_4 = self.encoder(images)

        # downscale injection and append as channel to x_4 -> lowest dim result of the encoder
        inj = torch.unsqueeze(self.lr_injection(injection), dim=1)
        x_4 = torch.cat((x_4, inj.repeat(1,1,2,1,1)), dim=1)

        dx_3 = self.lrelu(self.decoder[0](x_4))
        dx_3 = joinTensors(dx_3 , x_3 , type=self.joinType)

        dx_2 = self.lrelu(self.decoder[1](dx_3))
        dx_2 = joinTensors(dx_2 , x_2 , type=self.joinType)

        dx_1 = self.lrelu(self.decoder[2](dx_2))
        dx_1 = joinTensors(dx_1 , x_1 , type=self.joinType)

        dx_0 = self.lrelu(self.decoder[3](dx_1))
        dx_0 = joinTensors(dx_0 , x_0 , type=self.joinType)

        dx_out = self.lrelu(self.decoder[4](dx_0))
        dx_out = torch.cat(torch.unbind(dx_out , 2) , 1)

        out = self.lrelu(self.feature_fuse(dx_out))
        out = self.outconv(out)

        out = torch.split(out, dim=1, split_size_or_sections=self.
    channels)
        mean_ = mean_.squeeze(2)
        out = [o+mean_ for o in out]

        return out
```

C.5.2 Tuning

```
import os
import sys
import time

import torch
import numpy as np
from tqdm import tqdm
from torch.utils.tensorboard import SummaryWriter
from torch.optim import Adam


import config
import myutils
from loss import Loss
from torch.utils.data import DataLoader

import optuna

from dataset.set_n import get_loader, get_train_test
from model.FLAVR_arch_2 import UNet_3D_3D


def load_checkpoint(args, model, optimizer , path):
    print("loading checkpoint %s" % path)
    checkpoint = torch.load(path)
    args.start_epoch = checkpoint['epoch'] + 1
    model.load_state_dict(checkpoint['state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer'])
    lr = checkpoint.get("lr" , args.lr)
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr


def train(args, epoch, model, criterion, optimizer, train_loader):
    losses, psnrs, ssims = myutils.init_meters(args.loss)
    model.train()
    criterion.train()

    t = time.time()
    for i, (images, gt_image, inj) in enumerate(train_loader):

        # Build input batch
        inj = [inj_.cuda() for inj_ in inj]
        images = [img_.cuda() for img_ in images]
        gt = [gt_.cuda() for gt_ in gt_image]

        # Forward
        optimizer.zero_grad()
        out = model(images, inj)

        out = torch.cat(out)
        gt = torch.cat(gt)

        loss, loss_specific = criterion(out, gt)

        # Save loss values
        for k, v in losses.items():
            if k != 'total':
                v.update(loss_specific[k].item())
        losses['total'].update(loss.item())

        loss.backward()
        optimizer.step()


def test(args, epoch, model, criterion, test_loader):
    #print('Evaluating for epoch = %d' % epoch)
    losses, psnrs, ssims = myutils.init_meters(args.loss)
    model.eval()
    criterion.eval()

    t = time.time()
    with torch.no_grad():
        for i, (images, gt_image, inj) in enumerate(test_loader):

            images = [img_.cuda() for img_ in images]
            gt = [gt_.cuda() for gt_ in gt_image]
            inj = [inj_.cuda() for inj_ in inj]

            out = model(images, inj) ## images is a list of
   neighboring frames
            out = torch.cat(out)
            gt = torch.cat(gt)

            # Save loss values
            loss, loss_specific = criterion(out, gt)
            for k, v in losses.items():
                if k != 'total':
                    v.update(loss_specific[k].item())
            losses['total'].update(loss.item())

            # Evaluate metrics
            myutils.eval_metrics(out, gt, psnrs, ssims)

    # Print progress
    # print("Loss: %f, PSNR: %f, SSIM: %f\n" %
    #       (losses['total'].avg, psnrs.avg, ssims.avg))
    return losses['total'].avg, psnrs.avg, ssims.avg


""" Entry Point """
def objective(trial):
    ##### Parse CmdLine Arguments #####
    args, unparsed = config.get_args()

    # set optuna trials
    args.lr = trial.suggest_float('lr', 1e-5, 1e-3, log=True)
    args.upmode = trial.suggest_categorical('upmode', ['transpose'
    , 'upsample'])
    args.batch_size = trial.suggest_categorical('batch_size', [1,
    2, 4, 8])

    cwd = os.getcwd()
    print(args)

    ##### TensorBoard & Misc Setup #####
    device = torch.device('cuda' if args.cuda else 'cpu')
    torch.backends.cudnn.enabled = True
    torch.backends.cudnn.benchmark = True

    torch.manual_seed(args.random_seed)
    if args.cuda:
        torch.cuda.manual_seed(args.random_seed)

    paths, tr_idx, test_idx = get_train_test(args.data_root,
    random_state=214, ic=args.ic)
    train_loader = get_loader(paths, tr_idx, args.batch_size,
    shuffle=True, num_workers=args.num_workers, is_training=True,
    inter_frames=args.n_outputs, n_inputs=args.nbr_frame, channels
    =args.channels)
    test_loader = get_loader(paths, test_idx, args.batch_size,
    shuffle=False, num_workers=args.num_workers, is_training=False
    , inter_frames=args.n_outputs, n_inputs=args.nbr_frame,
    channels=args.channels)

    print("Building model: %s"%args.model.lower())
    model = UNet_3D_3D(args.model.lower() , n_inputs=args.
    nbr_frame, n_outputs=args.n_outputs, joinType=args.joinType,
    upmode=args.upmode, channels=args.channels)
    model = torch.nn.DataParallel(model).to(device)

    ##### Define Loss & Optimizer #####
    criterion = Loss(args)

    ## ToDo: Different learning rate schemes for different
    parameters
    optimizer = Adam(model.parameters(), lr=args.lr, betas=(args.
    beta1, args.beta2))
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=5, verbose=True)

    print('Beginning to train:')

    for epoch in range(args.start_epoch, args.max_epoch):
        train(args, epoch, model, criterion, optimizer,
    train_loader)

        test_loss, psnr, ssim = test(args, epoch, model, criterion
    , test_loader)
        #print(f'Test loss: {test_loss} PSNR: {psnr}, SSIM {ssim
    }')

        # update optimizer policy
        scheduler.step(test_loss)

    return psnr, ssim

def main():
  study = optuna.create_study(study_name='injFLAVRTune',
    directions=['maximize', 'minimize'])
  study.optimize(objective, n_trials=30)
  print(study.best_trial.value)

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

C.6 Overall Testing

The following was used as a jupyter notebook and converted to a .py file for formatting purposes.

```
# -*- coding: utf-8 -*-
"""testing.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1mnwx5jEHfbPZgf9-
    LLzdS7trKsT61Pxc
"""

#!pip install rasterio
#!pip install sewar
#!pip install scikit-image

import rasterio
import matplotlib.pyplot as plt
import numpy as np
import zipfile
import os
import sewar

def show_imgs(dir, name):
  for image in sorted(os.listdir(dir)):
    if name in image:
      path = os.path.join(dir, image)
      if '.ipynb' in path:
        continue
      with rasterio.open(path, 'r') as src:
        img = src.read()



      print(path)
      print(f'Number of bands: {img.shape[0]}')
      fig, ax = plt.subplots(1, img.shape[0], figsize= (5, 5))
      img = img/img.max()
      for i, band in enumerate(img):
        if img.shape[0] == 1:
          ax.imshow(band, cmap='gray')
          ax.set_xticks([])
          ax.set_yticks([])
        else:
          ax[i].imshow(band, cmap='gray')
          ax[i].set_xticks([])
          ax[i].set_yticks([])
      im = image.split('.')[0]
      plt.savefig(f'/content/{im}.png')

      #plt.show()

ZIP_PATH = '/content/predictions.zip'
with zipfile.ZipFile(ZIP_PATH, 'r') as zip_ref:
    zip_ref.extractall()
    print('Zip extracted')

show_imgs('/content/predictions/2nd_fusion_pred', '')

# clean all of the images
dir = '/content/'
for img in os.listdir(dir):
  if '0' in img:
    try:
      os.remove(os.path.join(dir, img))
    except:
      None

"""Evaluating metrics"""

from sewar import mse, rmse, psnr, uqi, ergas, scc, rase, sam,
    vifp, psnrb
from skimage.metrics import structural_similarity as ssim

import os
```

```python
import rasterio
import argparse
from collections import defaultdict
import concurrent.futures as cp
import numpy as np



# max obtained from google earth engine
def psnr_(x,y):
  return 10*np.log10(1/np.mean((x-y)**2))

def ssim_(x,y):
  return ssim(x[0],y[0],data_range=1)


tests= [mse, rmse, psnr_, uqi, ssim_, ergas, scc, rase, sam, vifp,
    psnrb]
def evaluate(test_dir, verbose=False):
  global tests

  # get image paths
  pred_pths = [os.path.join(test_dir,im) for im in sorted(os.
   listdir(test_dir)) if 'pred' in im]
  true_pths = [os.path.join(test_dir,im) for im in sorted(os.
   listdir(test_dir)) if 'true' in im]
  assert len(pred_pths) == len(true_pths)

  # run through the images in pairs
  results= defaultdict(lambda: 0)
  for i, (pred, true) in enumerate(zip(pred_pths, true_pths)):
    # read in the images
    with rasterio.open(pred, 'r') as src:
      pred_im = src.read()
    with rasterio.open(true, 'r') as src:
      true_im = src.read()

    # for test the images using each of the tests
    for test in tests:
      try:
        result = test(true_im, pred_im)
        if verbose:
          print(f'Iteration {i} - test: {test.__name__} = {result}
   ')
        if not np.isnan(result):
          results[test.__name__] += result
      except:
        if verbose:
          print(f'Iteration {i} - test: {test.__name__} Failed')

  # average results across all images
  results = {key: val/len(true_pths) for key, val in results.items
   ()}
  return results
```

```python

test_dirs = ['p2p_pred', 'flavr_pred', 'naive_pred', '2
    nd_fusion_pred', 'hr_pred', 'injflavr_pred', '
    injflavr_e2e_pred']

#for dir in test_dirs:
  #path = os.path.join('/content/predictions', dir)
path = '/content/o10'
print(dir, evaluate(path))
```