GET Lab

Universität Paderborn
Fakultät EIM
Institut für Elektrotechnik und Informationstechnik

MultiFlow – Optical Flow Estimation Using Deep Neural Networks

Master's Thesis
for the Master's Program in Computer Science
submitted by

Anshul Suresh Bansal

supervised by

Dr.-Ing. Mahmoud Mohamed
Dipl.-Ing. Markus Hennig

submitted to

Prof. Dr. Eyke Hüllermeier
Prof. Dr.-Ing. Bärbel Mertsching

Paderborn, December 19, 2020

Abstract

Motion detection and activity recognition are some of the most prominent tasks in the domain of computer vision. With large scale developments in autonomous driving and robotics, solving these tasks efficiently with accurate results becomes even more critical. Taking inspiration from these challenges, a new architecture called MultiFlow is proposed in this thesis to obtain robust optical flow in terms of accuracy and smoothness. This model is based on the works of FlowNetC [DFI+15], and it estimates optical flow by taking consecutive multiple image frames (image triplets) as inputs and computing the correlation between these image frames. The MultiFlow model is evaluated against the state-of-the-art neural network models that estimate the optical flow using multiple image frames and the original FlowNetC model. However, the MultiFlow model is a scene-specific model rather than a generalized model. Nonetheless, results show that the MultiFlow model outperforms several models and hugely improves on the best performing FlowNetC model by a margin of over 36 % on the MPI Sintel dataset [BWSB12].

Acknowledgements

I want to thank Clement Pinard, the author of FlowNetPytorch [Pin17], for answering the queries related to the implementation of the loss function and the architectural details of the model.

I want to thank Junhwa et al. the author of UnFlow [MHR18] for giving their time and answering the questions related to weight distribution provided to the loss function and also providing suggestions for training the neural network.

I want to thank Sam Pepose, author of FlowNet2 [Pep17], for providing his repository on GitHub from where code for flow visualization is taken as a reference.

Contents

1 Introduction

1.1 Motivation

Optical flow estimation is one of the core tasks in the computer vision domain. Optical flow estimation using multiple image frames is currently a research area that is explored extensively. Two factors motivate the task of optical flow estimation:

- The first motivating factor in estimating optical flow comes from its applications in diversified fields such as visual odometry, autonomous driving, and semantic segmentation of images. With the help of optical flow, autonomous driving cars can predict and analyze the position of objects in their surroundings (e.g., pedestrians), thereby helping the cars to understand motion of several objects. This would help to make the autonomous driving car more safe for its passengers and for the people outside of the car.

  In semantic segmentation of images, optical flow helps in identifying the moving and stationary regions in the image. Figure 1.1 shows a semantic segmentation of an image frame using optical flow. In the right side of the figure the blue region indicates the background water and pink region indicates the moving object (bird). Such segmentation of image can help to identify and analyze movement patterns of the non-rigid objects. With revolutionary technologies coming up in robotics, optical flow finds its application in several tasks in robotics. Notable tasks include motion detection and action detection, which a robot senses in its surroundings with a camera and light sensors. Optical flow can also be used in Simultaneous Localization And Mapping (SLAM), where accurate optical flow helps the robot to obtain accurate SLAM.

- The second motivating factor for computing optical flow is the development and advancement done in the field of Convolutional Neural Networks (CNNs) and their applicability in several computer vision tasks such as image classification, image segmentation and optical flow flow estimation. CNN does not require hand-engineered features

Figure 1.1: Semantic Segmentation of Image Frame using Optical Flow from [SSJB16]

since CNN consists of convolution layers which automatically extract the features from the images. Results provided by the work of authors in [DFI+15] and [SYLK18] show that estimating optical flow using CNNs has proven to be successful and accurate. Hence, CNNs are a powerful tool for optical flow estimation.

1.2 Overview of Optical Flow

The perceivable motion between two consecutive image frames in 2D is termed as optical flow. An optical flow field is described using a 2D vector field. The process of estimating this 2D vector field is known as optical flow estimation. To perform the task of optical flow estimation, various approaches are developed for pixel-level motion analysis over the years. These approaches are divided into two categories: variational and machine learning-based techniques.

- The first approach used for optical flow estimation falls under the category of variational techniques. The Horn-Schunck method described in [HS81] laid the foundation of optical flow estimation by treating it as an energy minimization problem. The method given by Horn-Schunck is a variational based global method that formulates the change in pixel intensity in temporal images and regularizes neighboring pixels to have a spatially smooth flow for the entire image frame at once. The Horn-Schunck formulation is based on two assumptions: brightness constancy and smoothness constancy. These assumptions are essential in solving the underlying problem of drastic change in pixel intensity and aperture problem in optical flow estimation. On the contrary, Lucas-Kanade in [LK81] addressed the problem of optical flow estimation differently. Instead of a global method, Lucas-Kanade uses a patch based technique for computing the optical flow. Instead of dealing with the entire image frame at

once, a small patch in the local neighborhood of the pixel is considered. The displacement vector is calculated for a pixel and its local neighborhood in an iterative scheme. To improve on the above methods for better estimation of optical flow, descriptor matching techniques such as Histogram of Gradients (HOG) introduced in [DT05] and Scale-Invariant Feature Transform (SIFT) introduced in [Low04] can be used for feature extractions. Weinzaepfel et al. proposed a model named DeepFlow in [WRHS13], which makes use of the so-called DeepMatching algorithm for the estimation of optical flow. The DeepMatching algorithm helps the DeepFlow model compute dense correspondence matching between the pixels of consecutive image frames. While the advantage of variational techniques is that they do not require any dataset for training the optical flow estimation algorithm, the disadvantage is that the estimated flow obtained by these methods may not be robust and completely accurate, since these methods perform poorly in case of large motion and are sensitive to noise.

- The second approach to estimate optical flow uses machine learning, particularly deep learning. Deep learning involves creating neural networks with deep structure (hidden layers ≥ 2). Hidden layers are layers between input and output of the neural network. Since these neural networks are required to be trained, the training methodologies are categorized into supervised and unsupervised learning techniques. Supervised learning techniques requires labeled data (dataset with ground-truth optical flow), and unsupervised learning techniques do not require labeled data. Deep learning requires a large dataset for model training, but the advantage is that the resultant optical flow field is robust and very accurate. Furthermore in deep learning, CNNs are used to solve the task of optical flow estimation. In [PC15], the authors show that CNNs are good at pixel level operations. CNNs learn from the features that are extracted from the image frames. Since optical flow estimation also operates on pixel-level granularity, CNNs are an excellent match in serving as a tool for this task.

Typically, to compute an optical flow field, consecutive image frame pairs are needed so that the apparent motion between them can be computed. These image frame pairs can be a sequence of image frames captured from a video sequence like the MPI Sintel [BWSB12] dataset or random image pairs like the Flying Chairs [DFI+15] dataset where the image frames are not ordered in a sequence and have planar motion of objects in the image frames. The pair of consecutive image frames are passed to an optical flow estimation model, and the model predicts the optical flow field for the corresponding image frame pair. Figure 1.2 shows consecutive image frames from the MPI Sintel dataset in the bottom section, along with the ground-truth optical flow field in the top section. The arrows inside the ground-truth optical

flow indicate the direction of movement of pixels between the two image frames.

Figure 1.2: Example of Optical Flow [DFI+15]. Refer Section 2.2 for Color Coding Scheme

Till now, the authors in [DFI+15] and [SYLK18] have estimated the optical flow field only using image pairs, i.e., two consecutive image frames per optical flow field. However, only a few works estimate the optical flow field using multiple consecutive image frames like image triplets. For example, the work of authors described in [RGS+19] is based on image frame triplets.

The intuition behind using multiple image frames is that an additional image frame will provide more motion information that can be leveraged to obtain better flow estimates. It is expected that having multiple image frames can help in better learning of displacement of the vector field of pixels as described in [RGS+19] and [MB18]. The inclusion of an additional image frame encourages better performance in correspondence learning and correspondence matching between the pixels of the image frames. Hence, in this thesis, the original FlowNetC architecture presented in [DFI+15] is adapted and modified to form the MultiFlow model which uses consecutive image frame triplets, i.e., three consecutive image frames per optical flow field, to estimate accurate, robust, and smooth optical flow fields. Additionally, the MultiFlow model estimates the optical flow field by executing the neural network only once.

1.3 Objectives of this Thesis

The main objective of this thesis is to estimate optical flow accurately and robustly using multiple image frames rather than the standard consecutive image pairs approach. While

estimating the optical flow field, the goal is to obtain smooth flow field vectors. For developing the CNN model, various machine learning frameworks are investigated. Since the CNN model needs to be trained on optical flow dataset(s) to learn to estimate optical flow, another objective is to investigate various datasets available for optical flow computation and select the appropriate dataset that satisfies the constraint of consecutive multiple image frames. After estimating the optical flow field, the objective is to compare the performance of the proposed neural network model against the state-of-the-art optical flow estimation models using an evaluation metric. Finally, the final objective of this thesis is to analyze the proposed model and see in which areas the neural network model can potentially be improved.

1.4 Structure of this Thesis

This thesis is further organized as follows: Chapter 2 introduces the fundamental concepts used in this thesis. A mathematical formulation of optical flow and various other concepts related to optical flow is presented. The chapter also discusses the core concepts used in CNNs. Additionally, a detailed description of the dataset used in this thesis is also given. Chapter 3 discusses related works that are relevant for this thesis and are used for performance evaluation. In Chapter 4, the original FlowNetC model is explained along with modifications made to the model to implement the MultiFlow model. The chapter also explains the training procedures used to train the MultiFlow model. Furthermore, in Chapter 5, results obtained by the MultiFlow model are presented, and Chapter 6 discusses the challenges faced during the training of the MultiFlow model. Finally, Chapter 7 presents the conclusion and future areas of research.

2 Fundamentals

This chapter explains the fundamental concepts used in this thesis. It starts with formulating optical flow with the brightness constancy assumption. Then a color wheel approach is described, which is used to visualize the optical flow field. Afterwards, the performance metric known as average end-point-error is defined. The color wheel and performance metric are used while evaluating the results in Chapter 5. This chapter also introduces core concepts in deep learning such as Convolutional Neural Networks (CNNs), the layers used to build a CNN, and a particular type of CNN based architecture named autoencoders.

2.1 Optical Flow

The task of optical flow estimation involves pixel-level motion prediction. Motion of several objects between two consecutive image frames that arise due to relative motion between the camera and the objects, is termed as optical flow. Figure 2.1 illustrates the idea of optical flow:

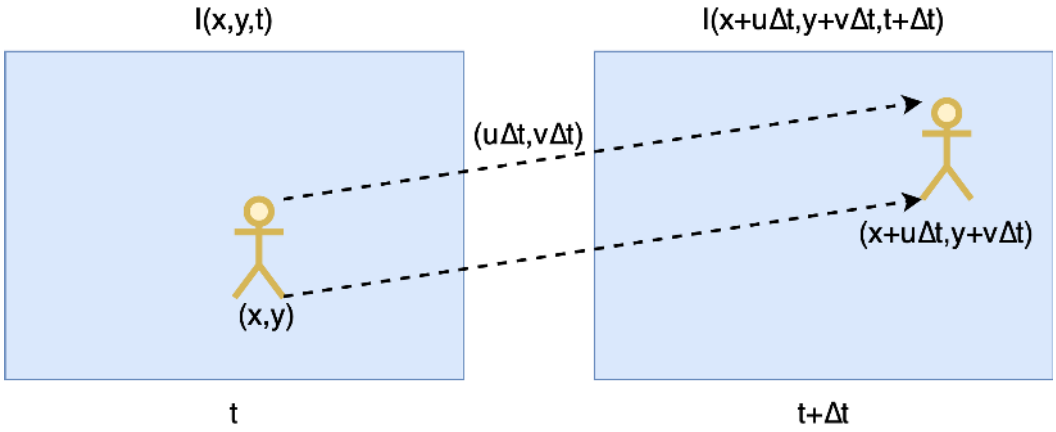


Figure 2.1: Optical Flow Estimation

In Figure 2.1, the two image frames can be expressed as a function of x, y, and t, where x and y denotes the location of a pixel at (x,y) and t denotes time. In image frame F_1 at time

t, the pixel is at location (x,y). After a certain amount of time Δt has passed, the pixel in image frame F_2 is at location ($x + u\Delta t$,$y + v\Delta t$) at time $t + \Delta t$ with velocity vectors u and v. While estimating the displacement vectors (u,v), the brightness constancy assumption is maintained meaning the intensity value of the pixel itself does not change.

The brightness constancy assumption in mathematical terms is stated as:

$$I(x, y, t) = I(x + u\Delta t, y + v\Delta t, t + \Delta t) \tag{2.1}$$

In simple words, if a pixel with RGB values (0,0,0) is at location (5,10) at time t=10 seconds in image frame F_1, when the pixel will be at location (10,5) at time t=11 seconds in image frame F_2, it will have the same RGB values (0,0,0). The resultant displacement of the pixel is (5,-5).

2.2 Optical Flow Visualization

The values of the optical flow field are vectors that denote the directionality of the movement of pixels. In simple words, these values will indicate where to find the pixel in the next image frame. To understand the displacement of the pixels between the image frames, the technique of visualization is used. A color-coding scheme is used to understand the flow of motion between the consecutive image frames and to visualize the optical flow field. The color-coding scheme used in this thesis is the color wheel. The location of color in the wheel depicts the direction of the displacement of pixels between the consecutive image frames. Figure 2.2 illustrates the color wheel.

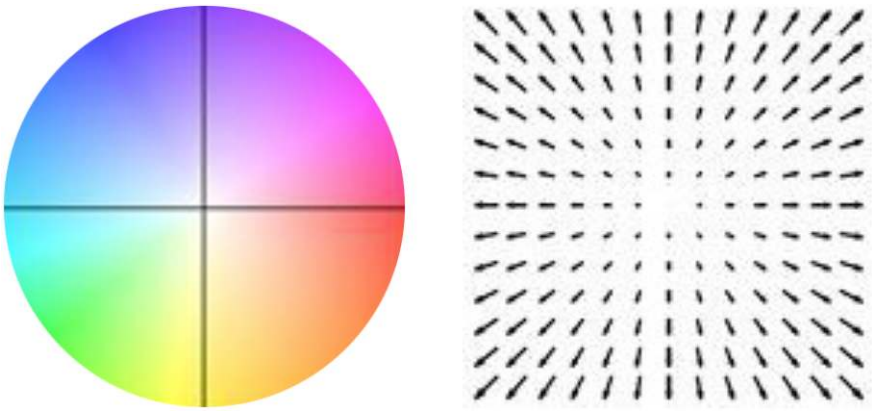


Figure 2.2: Color Wheel for Optical Flow Visualization [Git19]

In Figure 2.2, red color denotes that pixels have a displacement in the forward direction and yellow color denotes the displacement of pixels in the downward direction. A similar interpretation can be made for all other directions and their respective colors. The darker the color is (saturation), more is the displacement of that particular pixel.

2.3 Average End-Point-Error (AEPE)

The performance metric used to evaluate the MultiFlow model is called as the Average End-Point-Error (AEPE). This is a standard metric used to evaluate the performance of all optical flow models. AEPE metric calculates the euclidean distance between the predicted and the ground-truth optical flow. Large euclidean distance values denote that the predicted displacement of the pixel values are inaccurate, thereby yielding inaccurate optical flow fields. The loss function used to train the model also incorporates the AEPE function to compute the loss. Larger values of AEPE contributes to higher losses while training the model. The aim is to minimize the AEPE as much as possible, thereby estimating a robust and accurate optical flow field. Figure 2.3 gives a diagrammatic representation of EPE.

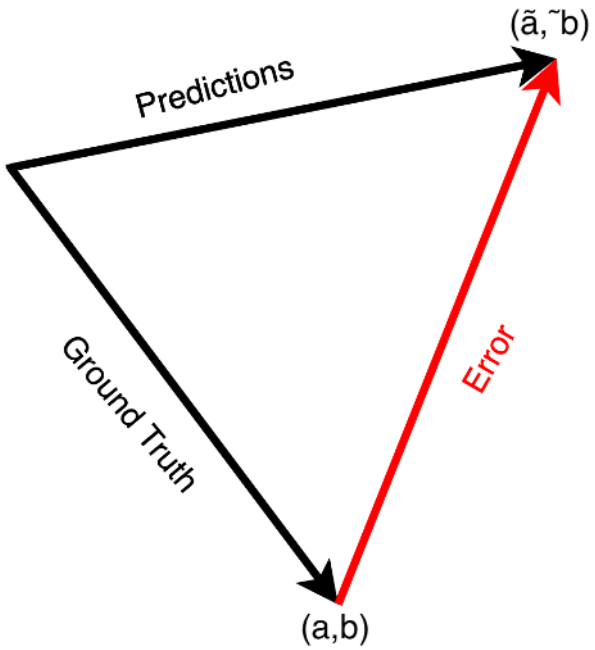


Figure 2.3: Diagrammatic Representation of EPE. Based on the figure given by the author in [Git19].

In mathematical terms, EPE is described as follows:

$$EPE = \left(\frac{1}{|AB|} \sum_{a,b \in A,B} \sqrt{(a - \tilde{a})^2 + (b - \tilde{b})^2} \right) \tag{2.2}$$

In Equation 2.2, a and b denotes the (x,y) location of the pixels in ground-truth optical flow and $\tilde{a}$, $\tilde{b}$ denotes the (x,y) location of the pixels in predicted optical flow. A and B denotes the total number of pixels over which EPE is calculated.

When computing EPE over N number of optical flow fields, AEPE is described as:

$$AEPE = \frac{1}{N} \left(\frac{1}{|AB|} \sum_{a,b \in A,B} \sqrt{(a - \tilde{a})^2 + (b - \tilde{b})^2} \right) \tag{2.3}$$

```
def a_epe(labels, predictions):
    squared_difference = tf.square(tf.subtract(predictions, labels))
    loss = tf.reduce_mean(squared_difference)
    loss = tf.sqrt(loss)
    loss = loss/len(df) #Number of image frames
    return loss
```

The code snippet above illustrates the implementation of average end-point-error in this thesis. This average end-point-error function is used in the multi-scale loss function of the MultiFlow model discussed in Section 4.3.2.

2.4 Convolutional Neural Networks

Convolutional Neural Networks are algorithms in the deep learning paradigm. CNNs have applications in various computer vision tasks such as image/object classification and image segmentation. The basic building blocks of convolutional neural networks are convolutional layers (refer to Section 2.5). The development of these networks is inspired by the working of neurons in a human brain. Since a CNN is made up of several convolution layers, each layer is responsible for identifying a particular visual receptive field pattern. For example, if a three-layer CNN is used for image classification, the first convolution layer can find the structure patterns. The second layer can find edge patterns. The third layer can find color patterns. Due to its applicability in various computer vision tasks, CNN has become a potent tool for solving complex computer vision tasks.

The general working principle of a convolution neural network is as follows:

A convolutional neural network is built up of an input layer, multiple hidden layers, and an output layer. A single image or multiple images are fed as inputs to the input layers. The input layer parameters are the dimensions of the input image (height, width, channels). The hidden layers contain a series of many convolution layers. These convolution layers are responsible for learning image characteristics and provide a compact representation in the form of feature maps with the help of inherent hierarchical patterns found in the image data. While passing the input image through many convolution layers, the extracted simple patterns are gradually combined to form an intricate pattern and, at last, forming the input image itself.

For the convolution process, the input image is convolved with a weighted convolution filter. The weights of the filter are automatically adjusted during the learning process by using backpropagation. Every neuron in the convolution filter has a limited receptive field in the region of the previous layer and operates within it. To introduce non-linearity and reduce the training time, activation functions are used in CNN architectures. These activation functions ensure that non-linear transformations are performed, making the CNN model learn and solve a complex task. Activation functions such as Rectified Linear Unit (ReLU), LeakyReLU, Sigmoid, and tanh are used for this purpose. Furthermore, CNN architectures with multiple convolutional layers (typically more than 10) also employ batch regularization. Batch regularization helps solve the internal covariate shift problem, which normalizes every input to the next layer to have zero-mean and variance of 1.

To carry out spatial down-sampling, pooling layers are used. Pooling layers reduces the dimension of the image, thereby reducing the number of parameters required to train the network. Finally, in many cases like for an image classification task, the output layer of the CNN architecture is a fully connected layer. After the feature extraction and dimensionality reduction operations, all the neurons in the fully connected layer are connected to the activation function of the previous layer. The activation values are summed for each neuron, and the neuron with the highest value is given as the output.

To develop a CNN, the most prominent machine learning frameworks present are TensorFlow and PyTorch. Both frameworks support various layers, such as convolution, pooling, fully connected, and transposed convolutions. Popular CNN architectures for image classification include GoogleNet given by the authors in [SLJ+15], ResNet introduced in [HZRS16], and VGGNet proposed by the authors of [SZ15].

Figure 2.4 shows a typical CNN created by stacking multiple convolution and pooling layers.

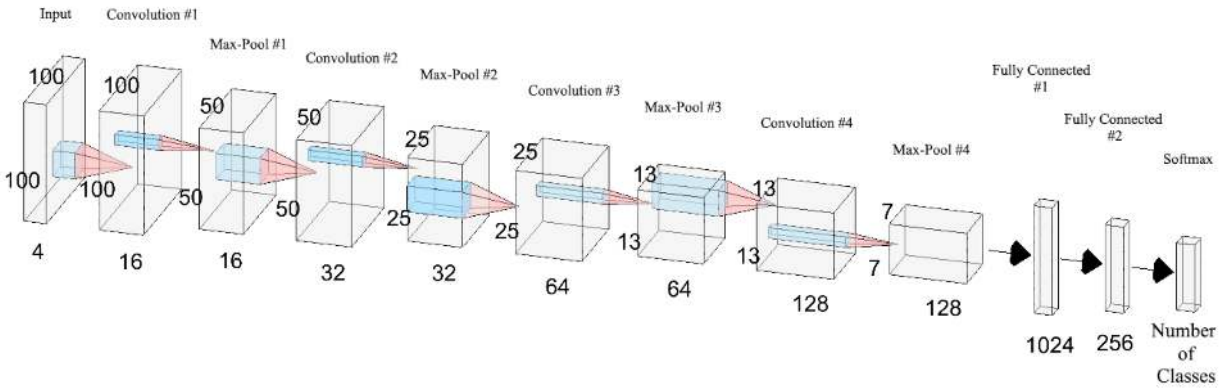


Figure 2.4: Typical CNN Architecture [MO18]

2.5 Convolution Layers

Convolution layers are the basic building blocks of CNNs. Convolution layers are referred to as feature extraction layers. To extract features from images, convolution layers make use of convolution filters. The process of extracting features from the images is as follows:

An input image (size x×y px) and a convolution filter is taken. Common filter sizes include 1×1, 3×3, 5×5, and 7×7. The image contains pixel information, and the convolution filter consist of weights. These weights denote how important the pixel is at that location to learn and extract features. The filter then slides across the entire x, y dimension of the image. During the sliding operation, the dot product between the filter's location and the image's pixels at that location is computed. The resulting dot product generates a 2D feature map. When there are several of these filters, each of these filter produces a feature map respectively. All of the generated feature maps are stacked on top of each other to produce the final convolution output. One pass of the convolution operation is illustrated in Figure 2.5:

In Figure 2.5, red, blue, and green are three different filters applied on the input channel (image frame). These filters then slide over the whole input channel, where their final results are summed up, and the final output volume is produced. The dimension of the output volume is controlled with the help of the following hyperparameters:

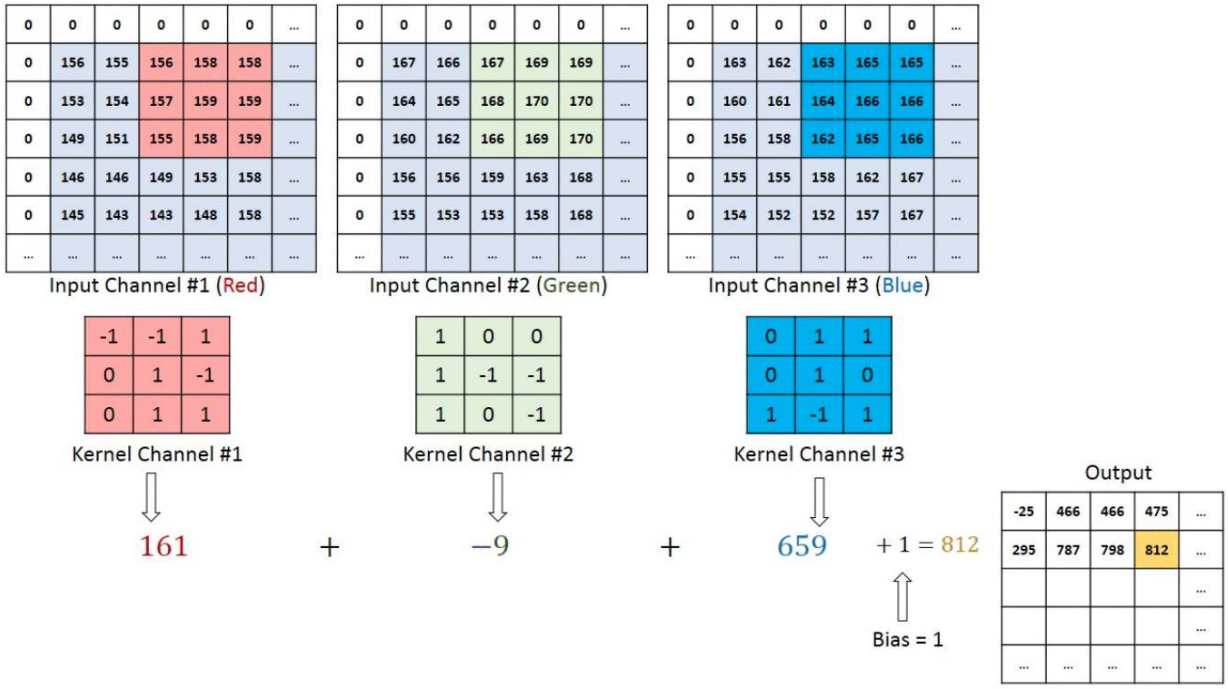


Figure 2.5: Example of Convolution Operation [Sah17]

- Zero-Padding (p): With the help of padding, the input can be padded around the border with zero. This helps in manipulating the spatial dimensions of the output volume.

- Stride (s): Striding helps to control the sliding operation of the filter. A stride of 1 indicates that the filter is moved by 1 pixel at a time in both x and y direction. Higher stride values lead to dimensionality reduction.

- Number of Filters (f_c): This parameter allows to control the depth of output volume. For example, if 32 filters are used, the output volume will be of size x × y × 32.

In mathematical terms, the dimension of the obtained output volume is given as:

$$[n, n, n_c], [f, f, n_c] = \left[\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor, \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor, n_f \right] \tag{2.4}$$

In Equation 2.4, n denotes image size, n_c denotes number of channels in the image (generally 3 denoting RGB), f denotes number of filters, p denotes padding size, s denotes the size of stride and n_f indicates the number of channels in the output convolution volume.

Convolution layer is implemented in TensorFlow using the following syntax:

```
tf.keras.layers.Conv2D
(
    filters, kernel_size, strides=(1, 1), padding='valid', data_format=None,
    dilation_rate=(1, 1), groups=1, activation=None, use_bias=True,
    kernel_initializer='glorot_uniform', bias_initializer='zeros',
    kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,
    kernel_constraint=None, bias_constraint=None, **kwargs
)
```

```
  #layer Declaration
  self.i1= InputLayer(input_shape=(436,1024,3))
  self.conva_1= Conv2D(64,7,strides=2,padding='same')

  #Layer Definition
  i_1=self.i1(input1)
  cona1=self.conva_1(i_1)
```

The code snippet above illustrates an example of a convolution layer implemented in the MultiFlow model.

2.6 Transposed Convolution Layers

The transposed convolution layer is the opposite of the convolution layer. The transposed convolution layer is also known by the name of Upconvolution. The whole idea of transposed convolution is to perform the convolution operation in the opposite direction. In convolution, a pixel value in the feature map contributes to a region in the input image. In contrast, transposed convolution layers tries to reproduce the same image region from the pixel value of the feature map. Transposed convolution layers are popularly used in semantic segmentation networks such as U-Net proposed in [RFB15], and optical flow networks. The transposed convolution operation is illustrated in Figure 2.6:

In Figure 2.6, a filter of size 3×3 along with a stride value of 1 is used, which upsamples the image size from 2×2 to 3×3. The filter is multiplied with each element of the image. The resulting feature maps are stacked and overlapped with each other to upsample the image size.

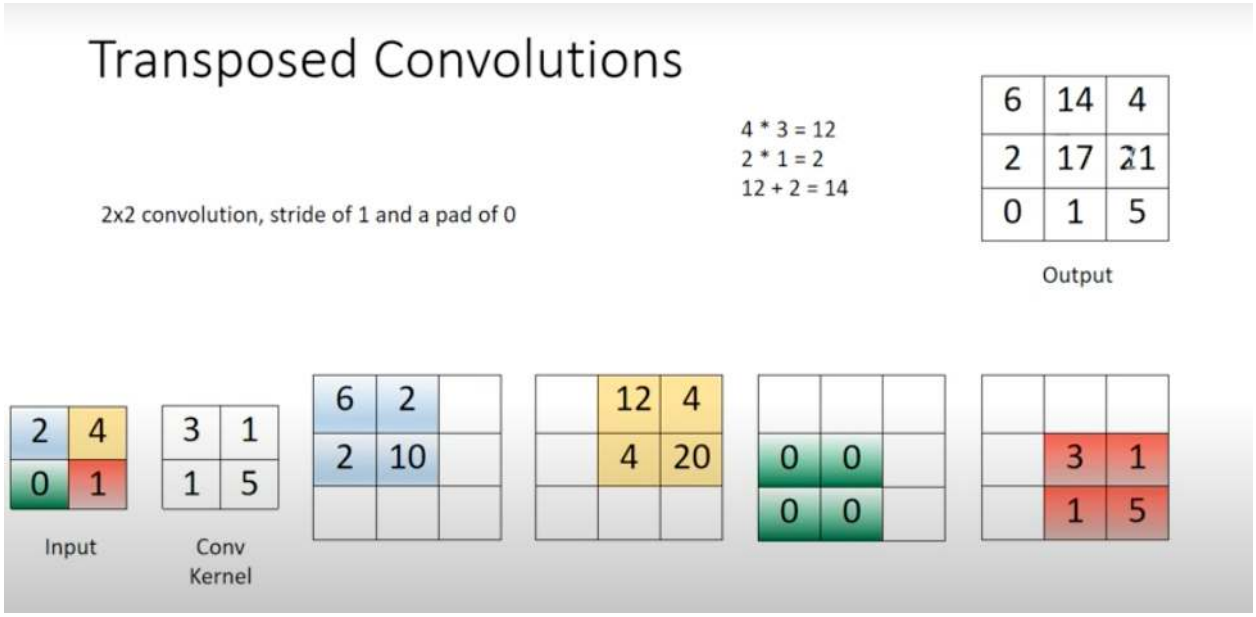


Figure 2.6: Example of Transposed Convolution Operation [Lie]

Transposed convolution layer in TensorFlow is implemented using the following syntax:

```
tf.keras.layers.Conv2DTranspose
(
    filters, kernel_size, strides=(1, 1), padding='valid', output_padding=None,
    data_format=None, dilation_rate=(1, 1), activation=None, use_bias=True,
    kernel_initializer='glorot_uniform', bias_initializer='zeros',
    kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,
    kernel_constraint=None, bias_constraint=None, **kwargs
)
```

```
  #layer Declaration
  self.dc5= Conv2DTranspose(512,4,strides=2,padding='same')
  self.up_6to5= Conv2DTranspose(2,4,strides=2,padding='same')

  #Layer Definition
  dc_5=self.dc5(con6_1_act)
  ups_6to5=self.up_6to5(pf_6)
```

The code snippet above illustrates an example of a transposed convolution layer implemented in the MultiFlow model.

2.7 Autoencoder - An Encoder Decoder Architecture

Autoencoders are a special type of CNN. They are referred to as regenerative architectures. Autoencoders are made up of two types of networks: an encoder network and a decoder network. The autoencoder architecture consists of convolution and transposed convolution layers for the task of image regeneration. Long Short Term Memory (LSTM) networks are used in autoencoders for text regeneration. The architecture of an autoencoder is as follows:

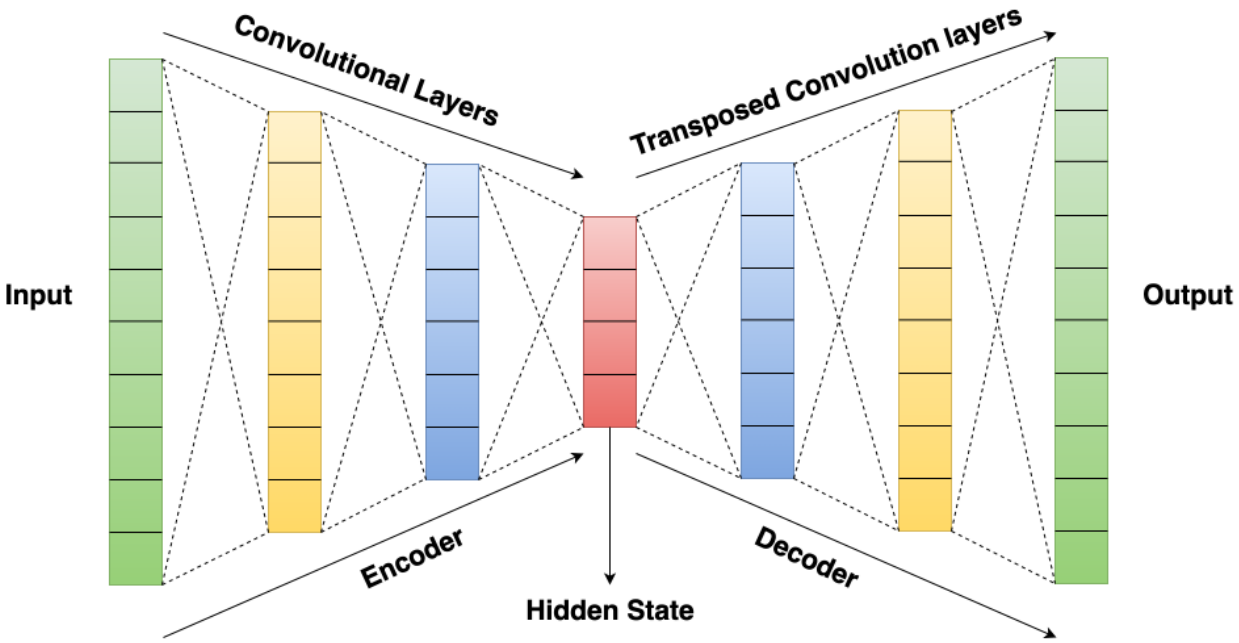


Figure 2.7: Autoencoder Architecture

- **Encoder Network:** The encoder network takes the original image/text as an input. Afterwards, the image/text information is encoded in the form of features using the convolution layers and passed further to the hidden layers of the neural network. The features are then passed to the decoder network. Thus, the encoder network can be seen as a compressor since these features are compressed using convolution layers.
- **Decoder Network:** The decoder network takes the compressed features as the inputs. From the compressed features, the decoder network tries to reproduce the original image/text using the transposed convolution layers to extract information from the features.

The autoencoders have become one of the favorite research topics in the field of machine learning where their applicability is seen in computer vision tasks such as Semantic Seg-

mentation, Genetive Adversarial Networks (GANS), Optical Flow computation, and also in tasks related to Natural Language Processing (NLP) such as Text Generation.

The architecture of the proposed MultiFlow model in this thesis is similar to an autoencoder network. The contractive network (initial convolution layers and the correlation layers) are part of an encoder network, and the refinement network is part of the decoder network.

2.8 Dataset

Neural networks require a large amount of training data because they learn from the training dataset and make predictions on the test (unseen) dataset. Since datasets can contain diversified information, it often leads to them being balanced or unbalanced. A balanced dataset has an equal number of images for all the categories present in the dataset. An unbalanced dataset does not contain an equal number of images for all the categories. From empirical studies it has been found that the CNN model that learns through a balanced dataset performs better. On the other hand, a model that learns through an unbalanced dataset tends to have a bias while making a prediction towards the majority class. Work done by the authors of [JK19] also affirms that the neural networks have better prediction accuracy when working with a balanced dataset.

Image datasets are preferred for CNNs, while time-series datasets are preferred for LSTM networks. Dataset generation are of various types:

- Real-life datasets - These datasets, as the name suggests, contain information and data about real-life entities that are captured through camera and motion sensors. For example, the KITTI 2012 dataset described in [GLU12], and KITTI 2015 dataset introduced in [MG15] contains motion information about a car moving on the road and capturing motion information in its surrounding.

- Synthetic datasets - These datasets are artificially generated datasets through computers. These datasets cannot mirror real-life scenarios, but they try to imitate them as much as possible. For example, MPI Sintel dataset proposed in [BWSB12] or the Flying Chairs dataset given by the authors of [DTSB15].

For training a neural network to estimate optical flow, several datasets are present. Table 2.1 gives a detailed description of widely used optical flow datasets.

Dataset	Image Frame Pairs	Image Frames with Ground-Truth
KITTI 2012	194	194
KITTI 2015	200	200
Flying Chairs	22872	22872
MPI Sintel	1041	1041

Table 2.1: Size of Various Optical Flow Datasets

2.8.1 MPI Sintel Dataset

MPI Sintel is a dataset inspired by a short 3D film named Sintel. Although all of the datasets mentioned above are very popular for estimating optical flow, in this thesis the primary choice for training the MultiFlow model is the MPI Sintel dataset. The reasons are as follows:

- The main idea behind the MultiFlow model developed in this thesis is to use image frame triplets to estimate optical flow. Such kind of data is only present in the MPI Sintel dataset. Although KITTI 2012/2015 dataset has consecutive image frames, the dataset consists of fewer training image frames with ground-truth optical flow. Also, the Flying Chairs dataset only consists of discrete consecutive image pairs. Hence, the Flying Chairs and KITTI 2012/2015 datasets cannot be used to train the MultiFlow model.

- The MPI Sintel dataset contains different scenes with different motion characteristics, thereby helping the MultiFlow model learn varied motions and generalize its performance better than the existing models. MPI Sintel consists of 1041 image pairs. However, due to the requirement of the MultiFlow model, these image pairs are rearranged to form 1018 image frame triplets.

- For performance evaluation several models presented in Chapter 5 use this dataset. Therefore, the MPI Sintel dataset serves as the best choice for the MultiFlow model.

Table 2.2 provides a detailed description of the MPI Sintel dataset.

Dataset	**MPI Sintel**
Image Resolution	**436 * 1024** px
Total Scenes	**23**
Scenes with 50 Image Frames	**19**
Scenes with <50 Image Frames	**4**

Table 2.2: MPI Sintel Dataset Details

$.flo$ are ground-truth optical flow files that contain the 2D vector field of the optical flow. Furthermore, the dataset is rendered into three categories based on various complexity levels: albedo, clean, and final. The characteristics of each render category are as follows:

1. **Albedo:** Image frames are rendered as 2D, satisfying the constraint of constant brightness assumption everywhere except in the regions of occlusion. Figure 2.8 and 2.9 shows the consecutive image frames of the albedo category and Figure 2.10 shows the optical flow between them.

2. **Clean:** Enhances the image frames with natural shading, lighting effects, and cast shadows, thereby enhancing the details in the image frames. Figure 2.11 and 2.12 shows the consecutive image frames of the clean category and Figure 2.13 shows the optical flow between them.

3. **Final:** Image frames are further enhanced by using motion blur, focus blur making the image frames look as real and cinematic as possible. Figure 2.14 and 2.15 shows the consecutive image frames of the final category and Figure 2.16 shows the optical flow between them.

Although the image frames from all the categories have different texture characteristics, they exert the same motion, hence the optical flow between the image frame F_1 and image frame F_2 of all the categories is the same.

Figure 2.8: Albedo Frame F_1 [BWSB12]

Figure 2.9: Albedo Frame F_2 [BWSB12]

Figure 2.10: Optical Flow [BWSB12]

Figure 2.11: Clean Frame F_1 [BWSB12]

Figure 2.12: Clean Frame F_2 [BWSB12]

Figure 2.13: Optical Flow [BWSB12]

Figure 2.14: Final Frame F_1 [BWSB12]

Figure 2.15: Final Frame F_2 [BWSB12]

Figure 2.16: Optical Flow [BWSB12]

3 Related Work

This chapter describes the prominent works done for optical flow estimation using consecutive image frame pairs and consecutive image frame triplets with the help of Convolution Neural Networks (CNNs). A comparative study between the image frames pairs and image frame triplets approaches is performed. Decisions taken for developing the MultiFlow model are underlined.

3.1 Image Frame Pairs Approaches

This section describes the foundational works done for optical flow estimation using CNNs with the help of two consecutive image frames.

- Fischer et al. in [DFI+15] proposed two CNN based models which are fully end-to-end trainable. FlowNetSimple (FlowNetS) is the first model, which uses consecutive image pairs to estimate the optical flow. It is a sequential neural network model where the input image frames are stacked together. These image frames pass through a series of convolution layers, where features are extracted from the image frames, and the model then estimates the optical flow. FlowNetCorrelation (FlowNetC) is another variant of FlowNet models where two input streams merge into a correlation layer. In FlowNetC, learning image features of the two image frames is not sufficient for optical flow estimation, since the features of the two image frames do not have any one-to-one correspondence between them. An additional layer is required to find the correspondences between the two feature maps and match the pixels between them. A correlation layer is used to perform the correspondence matching, which correlates pixels between two feature maps and then computes optical flow. FlowNetC can predict complete flow fields and gives promising results on optical flow benchmark datasets. FlowNetS and FlowNetC obtained good results on the Flying Chairs and MPI Sintel dataset which paved the way for CNNs to be used in the task of optical flow estimation.

Advantages of FlowNetC:

- Good performance on the Flying Chairs and the MPI Sintel Dataset.
- Use of non-weighting correlation layer for dense correspondence matching between the pixels of the feature maps.

Disadvantage of FlowNetC:

- Designed only for two consecutive image frames.

- PWC-Net introduced by the authors in [SYLK18] is another prominent architecture which computes optical flow using pyramidal structure, image warping techniques, and cost volume. Using convolution layers a l-level feature pyramid is constructed. After constructing the feature pyramid, the features of the second image frame are warped with the upsampled flow to obtain the first image frame in the warping layer. Since interpolation techniques are used, the warped image is not a perfect reconstruction. After this, the first image frame and the warped image frame are passed to the cost volume layer, where a pixel matching function is executed. PWC-Net performed significantly better than the FlowNet models on the MPI Sintel dataset.

- **Limitation:** All the architectures mentioned above have the limitation that they can estimate optical flow while using only two consecutive image frames.

3.2 Multiple Image Frames Approaches

This section describes the foundational works performed for optical flow estimation using CNNs with the help of consecutive image frames in the form of image triplets. All the models mentioned below are used for evaluation purposes against the MultiFlow model in this thesis in Chapter 5.

- PWC-Fusion introduced in [RGS+19] is a CNN architecture which estimates optical flow using multiple image frames. PWC-Fusion architecture is similar to PWC-Net and uses the same pyramid structure, warping, and cost volume technique for flow estimation. Multiple execution of the PWC-Net model is performed and the resulting flow fields are fused together in PWC-Fusion model.

Advantage of PWC-Fusion:

- Has best performance on the final render category of MPI Sintel dataset.

Disadvantage of PWC-Fusion:

- Cannot predict the optical flow by executing the model only once. The model is executed 3 times to estimate a single optical flow field.

- Another architecture that works on a multi-frame approach for optical flow estimation is introduced in [YCVDWM20] called TIMCflow. The TIMCflow model uses a pre-trained CNN extractor for feature extraction. It uses an image matching cost function (also called a likelihood function) for optical flow estimation. The authors use the previous and future image frames in the likelihood function and utilize the local constancy assumption of optical flow to estimate the flow field in the occluded regions of image frames.

- One more architecture to use multiple image frames for optical flow estimation introduced by the authors of [WSLB17] is MR-Flow. The architecture of MR-Flow uses a mixture of an optical flow network and a semantic segmentation network to estimate the optical flow. The semantic segmentation network is used to separate the static (rigid regions) and moving regions of the image frames. The flow in these regions is estimated separately and later combined to obtain the final optical flow.

  Drawback of MR-Flow:

  - High computation time to estimate optical flow since it requires the execution of several neural networks. The optical flow is estimated in 2 minutes per image frame triplet by the model.

- The architecture of ProFlow proposed in [MB18] also uses multiple image frames, and it is trained using unsupervised learning techniques. ProFlow computes backward and forward flow between the image triplets. ProFlow combines and refines the forward and backward flow to estimate the complete optical flow field. To combine the forward and backward flow, ProFlow uses the technique of interpolation.

  Advantage of ProFlow:

  - Even though ProFlow model is trained using unsupervised learning technique, it outperforms the supervised models such as TIMCflow and MR-Flow, since the results obtained by the work of authors in [Lov02] show that unsupervised trained models do not perform as good as models trained using supervised training techniques.

Disadvantage of ProFlow:

- High computation time to predict the optical flow field. After passing an image frame triplet as the input, the optical flow is predicted in 112 seconds.

Figure 3.1 illustrates the chronological order of various CNN based optical flow estimation models.

Figure 3.1: Timeline of CNN based Optical Flow Estimation Models

3.3 Why MultiFlow?

- The inspiration behind developing the MultiFlow model was taken from the results obtained by the models that used multiple image frames for optical flow estimation. These models regularly outperformed the models which used image frame pairs for optical flow estimation. Hence, developing a model which uses multiple image frames for

optical flow estimation was a compelling reason due to the performance gains obtained using this approach.

- FlowNetC model uses an additional correlation layer for correspondence matching between the pixels. Hence, the FlowNetC model performed better than the FlowNetS model for all datasets with the exception of the final render category of the MPI Sintel dataset. FlowNetC model has also been used in several works such as [IMS+17], where FlowNetC is a sub-part of the FlowNet2 model or [MHR18] where FlowNetC is stacked with other networks and trained in the unsupervised learning method. Also, the image pair based PWC-Net model has an advanced variant named PWC-Fusion, which used multiple image frames to estimate optical flow. However, no such exploration of the original FlowNetC model was performed where the standalone FlowNetC model used multiple image frames. Hence, the reasons mentioned above served as the inception of MultiFlow model development.

4 Development & Implementation

This chapter explains in detail the implementation of the MultiFlow model. First, the original FlowNetC model is explained on which the MultiFlow model is based. The working of correlation operation is also explained in detail. Furthermore, the chapter explains the implementation of the MultiFlow model to estimate optical flow using image frame triplets. Afterwards, the chapter specifies the hardware and software resources used to develop and train the MultiFlow model. Finally, the chapter investigates various training procedures to train the MultiFlow model.

4.1 Original FlowNetC Architecture

For the task of supervised optical flow estimation, many CNN models have been developed. However, most of the CNN models for optical flow estimation are inspired from the FlowNetS and FlowNetC model based on the results they achieved on the Flying Chairs and MPI Sintel dataset.

The idea for developing FlowNetC given by Fischer et al. in [DFI+15] was based on the applicability of Convolutional Neural Networks (CNNs) for solving computer vision-related task. The FlowNetC model has three key components. The first component in the contractive network which is used for generating the features from the image frames. The next component is the correlation layer that generates a correlation cost volume using the features maps. The final component is a multi-stage refinement network that outputs the predicted flow fields in various dimensions. The working of each component is described below.

4.1.1 Contractive Network of FlowNetC

The FlowNetC model has two separate input streams for each image frame. These input layer is followed by a series of three convolution layers for feature extraction. Both the

input streams have identical layer structure. The idea behind such an architecture is to obtain meaningful representations of the image frames in the form of feature maps before estimating the optical flow.

The next step is to take a patch of features from the first feature map and compare it with the patch of features at the same location in the second feature map to compute pixel correspondences between the two feature maps.

4.1.2 Correlation Layer

The *correlation* layer is used to combine the two feature maps and find dense pixel correspondences between these feature maps. The correlation layer performs multiplicative patch comparison between the two feature maps. The working of the correlation layer is as follows:

- **Input:** Two feature maps f_1 and f_2 of the input image frames with width $= w$, height $= h$, and number of channels $= c$, where $f_1,\ f_2 : \mathbb{R}^2 \rightarrow \mathbb{R}^C$.
- **Output:** Correlation volume containing the outputs of the multiplicative patch comparisons from the feature maps f_1 and f_2.
- **Process:** The working of one pass of the multiplicative patch comparison is as follows:

  The correlation between the patches of two feature maps f_1 and f_2 centered at locations x_1 and x_2 in respective maps is computed using the following formula:

$$c(x_1, x_2) = \sum_{o \in [-k,k]*[-k,k]} \langle f_1(x_1 + o), f_2(x_2 + o) \rangle \tag{4.1}$$

  In Equation 4.1 correlation is computed for a patch of size $K = 2k + 1$. The process of computing the correlation between the feature maps is similar to the process of convolution. In convolution operation, a convolution filter is convolved with the pixel data from the feature map. In contrast, in correlation operation, data from the first feature map is convolved with data from the second feature map, resulting in a correlation volume. It is due to this reason the correlation volume does not consist of any weights. Hence, this layer is not trained during the training process of the neural network.

  One patch of correlation computation involves $c \times K^2$ multiplication operations. Computing the correlation volume over the whole width and height of the image frames

multiplies the multiplicative operations by the number of $w^2 \times h^2$, thereby making the process computationally expensive.

In order to reduce the computations, a displacement window d is defined for each location in x_1 over which correlation cost $c(x_1, x_2)$ is computed in the local region of $D = 2d + 1$, thereby reducing the range of x_2. Also striding parameters s_1 & s_2 are introduced to quantize the locations x_1 & x_2. Figure 4.1 illustrates the correlation operation between two feature maps.

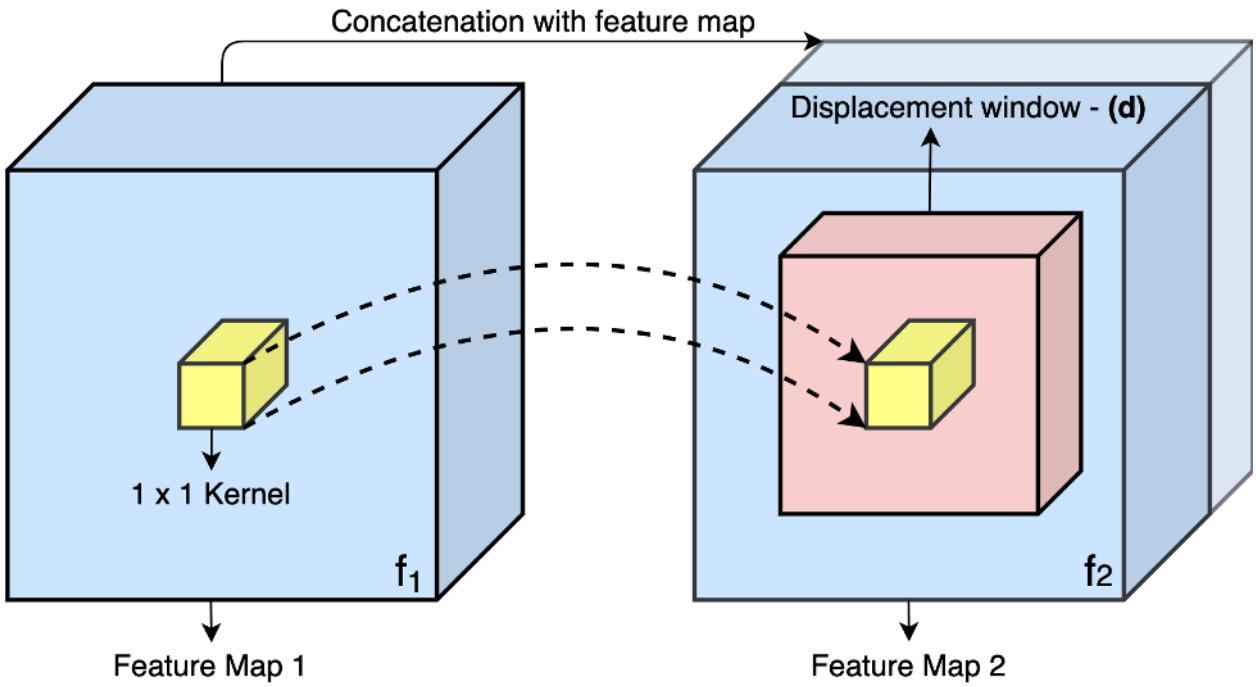


Figure 4.1: Correlation Operation between Two Features Maps

In Figure 4.1, the blue region denotes the features maps of the image frames. The red region indicates the displacement window over which the correlation is computed using the 1×1 kernel denoted by yellow color.

Correlation layer syntax in TensorFlow is as follows:

```
tfa.layers.CorrelationCost(kernel_size: int, max_displacement: int, stride_1:
    int, stride_2: int, pad: int, data_format: str, **kwargs)}}
```

For optical flow estimation, Figure 4.2 illustrates the contractive network of FlowNetC. In the figure, two input streams are present for two consecutive image frames. The two image frames go through a series of three convolution layers that extract features from the image frames. A correlation layer is then used, which computes the correlation volume between

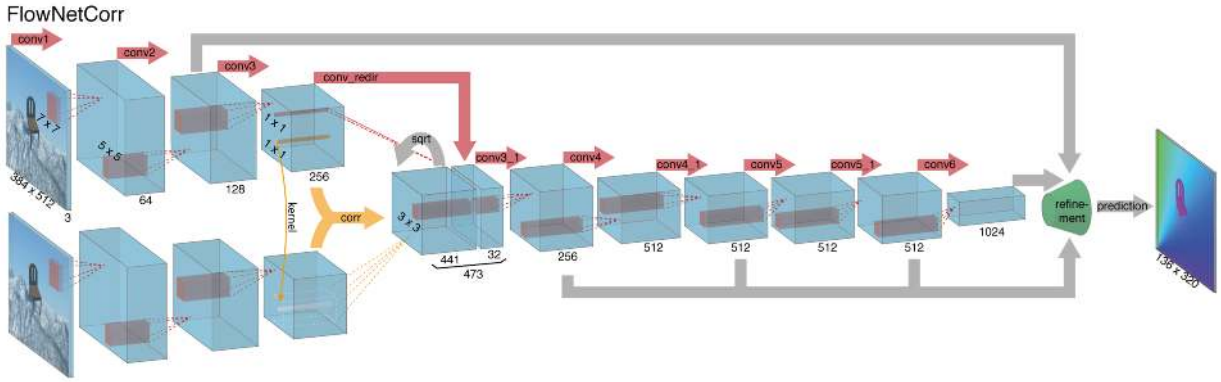


Figure 4.2: Standard FlowNetC Architecture [DFI+15]

the feature maps. As mentioned earlier, since the correlation volume does not consist of any weights, a feature map of the first input stream is concatenated with the correlation volume, thereby adding meaningful information to the correlation volume. The correlation volume then passes through a series of seven convolution layers where more features are extracted.

```
#MultiFlow Model

class MultiFlow(tf.keras.Model):
  # Layer declaration
  def __init__(self):
    super(MultiFlow, self).__init__()
    self.i1= InputLayer(input_shape=(436,1024,3)) #(Height,Width,Channels)
    self.i2= InputLayer(input_shape=(436,1024,3))

    #Input path-1
    self.conva_1= Conv2D(64,7,strides=2,padding='same') #Convolution layer
    self.conva_1_act = LeakyReLU(alpha=0.1) # LeakyRelu function
    self.conva_2= Conv2D(128,5,strides=2,padding='same')
    self.conva_2_act = LeakyReLU(alpha=0.1)
    self.conva_3= Conv2D(256,5,strides=2,padding='same')
    self.conva_3_act = LeakyReLU(alpha=0.1)

    #Input path-2
    self.convb_1= Conv2D(64,7,strides=2,padding='same')
    self.convb_1_act =LeakyReLU(alpha=0.1)
    self.convb_2= Conv2D(128,5,strides=2,padding='same')
    self.convb_2_act =LeakyReLU(alpha=0.1)
    self.convb_3= Conv2D(256,5,strides=2,padding='same')
```

```
24      self.convb_3_act =LeakyReLU(alpha=0.1)
25
26      #Correlation layer
27      self.cc = CorrelationCost(1,20,1,2,20,data_format='channels_last')
28      #LeakyRelu for correaltion volume
29      self.cr_1_act= LeakyReLU(alpha=0.1)
30      #Convolution redir for features for correlation volume
31      self.conva_redir= Conv2D(32,1,strides=1)
32      self.conva_redir_act =LeakyReLU(alpha=0.1)
33      self.vol_1= Concatenate(axis=3)
34
35    #Layer Definition
36    def call(self, input1,input2,training=False):
37      #Layer Definition for Input path-1
38      i_1=self.i1(input1)
39      cona1=self.conva_1(i_1)
40      cona1_act=self.conva_1_act(cona1)
41      cona2=self.conva_2(cona1_act)
42      cona2_act=self.conva_2_act(cona2)
43      cona3=self.conva_3(cona2_act)
44      cona3_act=self.conva_3_act(cona3)
45
46      #Layer Definition for Input path-2 same as Input path-1
47
48      #Layer Definition for Correlation Volume
49      cc1=self.cc([cona3_act,conb3_act])
50      cc1_act=self.cr_1_act(cc1)
51      cona_r=self.conva_redir(cona3_act)
52      cona_r_act=self.conva_redir_act(cona_r)
53      v1=self.vol_1([cc1_act,cona_r_act])
```

The code snippet above shows the implementation of the correlation layer in the MultiFlow model. The In the class API of TensorFlow, all the layers are declared with the required arguments in the *def__init__* function and called using the *def call* function. In the *def__init__* function on line 7-8, input layers are defined where the image frames are taken as input. Line 11-16 indicates the convolution layers used for the first input stream. Line 21 declares the correlation layer with the values of the respective arguments. Line 25 declares the convolution layer used for concatenating with the correlation volume, and line 27 declares

the concatenation layer. In the *def call* function, all the layers declared above are called, line 31-38 performs the convolution operations for the first input stream. Line 43 performs the correlation operation, and line 47 performs the concatenation of the correlation with the convolution layer of the first input stream.

4.1.3 Refinement Network of FlowNetC

The refinement network of FlowNetC has modifications to the approaches used by Long et al. in [LSD15] and Dosovitskiy et al. in [DTSB15]. These modifications are as follows:

1. In the neural network model presented by Long et al. a single patch from the coarse prediction is *upsampled* using a transposed convolution layer (refer 2.6) and this process is repeated for all the patches of the coarse prediction. In contrast, Fischer et al. *upsamples* the whole coarse prediction at once in the FlowNetC model.

2. Dosovitskiy et al. also used the transposed convolution layers to *upsample* coarse predictions. However, Fischer et al. modified this approach by *upsampling* the coarse predictions and concatenating the *upsampled* coarse predictions with feature maps from the contractive network of the FlowNetC model. This allowed for more information to be passed to the refinement network of the FlowNetC model to obtain fine-grained optical flow predictions.

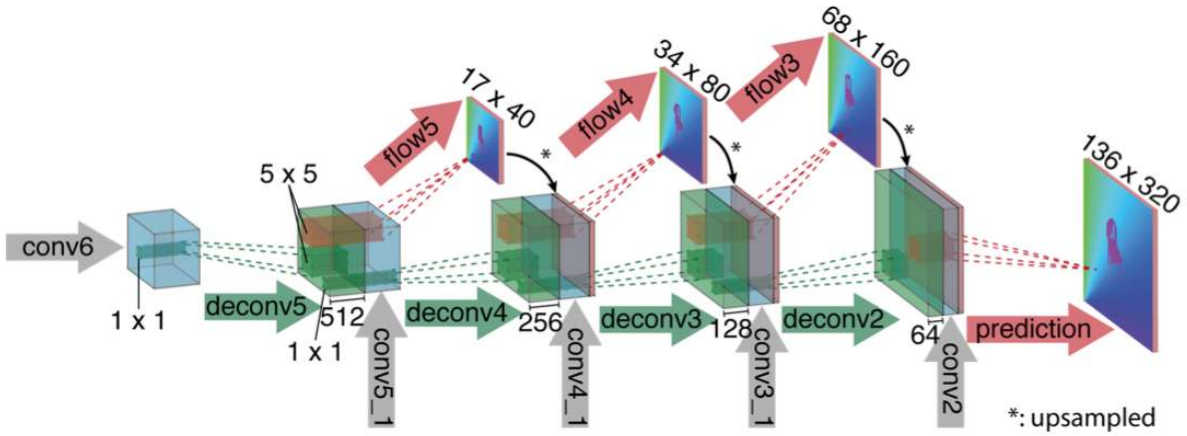


Figure 4.3: Refinement Network of FlowNetC [DFI+15]

Figure 4.3 illustrates the refinement network of the FlowNetC model. The refinement network uses transposed convolution layers at each level to upsample the coarse predictions. By performing this operation at each level of the refinement network, the resolution of the

coarse prediction increases by twice (2X). As mentioned above at each stage, the upsampled coarser flow prediction and the feature maps from the contractive network are concatenated using skip connections. In skip connections the output of a convolution layer is passed to the next convolution layer along with a copy passed to some other layer of the neural network. The idea behind performing the concatenation is to help the FlowNetC model learn displacement patterns efficiently. This is done by providing the refinement network, local pixel information of the feature maps from the contractive network and, at the same time, providing information obtained from the coarser predictions. The upsampling process is performed four times, which increases the resolution of the flow predictions. However, the resolution of the predicted flow field is still four times (4X) smaller than the original input resolution. The reason for this is because more convolution layers are applied on the image frame in the contractive network of the FlowNetC model as opposed to the transposed convolution layers in the refinement network.

Fischer et al. observed that adding more refinement layers did not help to increase accuracy performance of the FlowNetC model. Adding more refinement layers was proving to be computationally expensive because it increased the number of trainable parameters in the FlowNetC model. Hence, 4X bilinear upsampling (a technique used to resize image frames using bilinear interpolation) is used for obtaining the flow in the original resolution. Bilinear upsampling is computationally less expensive since it had no trainable weights and provided satisfactory results. The output of the FlowNetC model was the optical flow field between the consecutive image frames which did not require any post-processing of the predicted flow.

4.2 Implementation of MultiFlow

As mentioned earlier, the FlowNetC model is one of the first deep learning neural networks to estimate optical flow. However, it has a limitation. The FlowNetC model's limitation is that it estimates the optical flow by only using two consecutive image frames. Due to this constraint, the optical flow that is obtained from the FlowNetC model is not consistently accurate. To solve the limitation mentioned above, the MultiFlow model is developed. The MultiFlow model is based on FlowNetC architecture, which inherits the basic idea of the FlowNetC architecture, but modifies it for estimating optical flow by using more than two consecutive image frames. In this case, the MultiFlow model uses consecutive image frames in the form of image frame triplets.

4.2.1 Contractive Network of MultiFlow

The difference between the original FlowNetC model and the MultiFlow model is the inclusion of an additional input stream for the third image frame and more learning layers in the contractive network to extract features and learn the flow characteristics. To estimate optical flow using the MultiFlow model, three input image frames *Image* I_1, *Image* I_2 and *Image* I_3 are used. Figure 4.6 illustrates the contractive network of the MultiFlow network with the modifications highlighted in the red box. The figure is present at the end of the chapter. Similar to the original FlowNetC architecture, first I_1 and I_2 are passed through a series of convolution layers and the correlation volume $CR1$ between I_1 and I_2 is computed. Then the feature map $Conv3$ of I_1 is concatenated with the correlation volume $CR1$.

- **Modifications:**
  Simultaneously, I_2 and I_3 are passed through a series of convolution layers, and the correlation volume $CR2$ between I_2 and I_3 is computed. Similarly, the feature map $Conv3$ of I_2 is concatenated with the correlation volume $CR2$. All the three input streams have the same layer architecture.

  After calculating the two correlation volumes, $CR1$ and $CR2$ are passed through convolution layers $Conv\_cr1$ and $Conv\_cr2$ for extracting more features from the correlation volumes $CR1$ and $CR2$. The final correlation volume $CR3$ between the convolution layers is then computed. Similarly, the feature map $Conv\_cr1$ of the correlation volume $CR1$ is concatenated with the correlation volume $CR3$. The correlation volume $CR3$ of the MultiFlow model passes through seven convolution layers, where more features are extracted from $CR3$.

  The reason for considering this type of architecture is because I_3 is used for providing more motion information for predicting the flow between I_1 and I_2. Additional convolution layers are used to extract features from I_3 and correlation volumes. Also, the correlation layer can only take features map as inputs to perform the correlation operation. Hence, after computing $CR1$ and $CR2$, they are passed through convolution layers to obtain $Conv\_cr1$ and $Conv\_cr2$. Additional correlation layers are used to perform simultaneous correlation between between I_1, I_2 and I_2, I_3. Computing third correlation $CR3$ performs correlation among correlation volumes, thereby helping the neural network to interrelate the three image frames with other, which makes the neural network easier to understand the displacement vectors between the pixels of all the three image frames. Since, the image frames are consecutive, it helps to obtain better optical flow in terms of accuracy and smoothness.

```
#MultiFlow Model

class MultiFlow(tf.keras.Model):
  # Layer declaration
  def __init__(self):
    super(MultiFlow, self).__init__()
    self.i1= InputLayer(input_shape=(436,1024,3)) #(Height,Width,Channels)
    self.i2= InputLayer(input_shape=(436,1024,3))
    self.i3= InputLayer(input_shape=(436,1024,3))

    #Input path-1
    self.conva_1= Conv2D(64,7,strides=2,padding='same') #Convolution layer
    self.conva_1_act = LeakyReLU(alpha=0.1) # LeakyRelu function
    self.conva_2= Conv2D(128,5,strides=2,padding='same')
    self.conva_2_act = LeakyReLU(alpha=0.1)
    self.conva_3= Conv2D(256,5,strides=2,padding='same')
    self.conva_3_act = LeakyReLU(alpha=0.1)

    # Layer Declaration for Input path-2 same as Input path-1

    # Layer Declaration for Input path-3 same as Input path-1

    #Correlation layer for CR1
    self.cc = CorrelationCost(1,20,1,2,20,data_format='channels_last')
    #LeakyRelu for correaltion volume
    self.cr_1_act= LeakyReLU(alpha=0.1)
    #Convolution redir for features for correlation volume
    self.conva_redir= Conv2D(32,1,strides=1)
    self.conva_redir_act =LeakyReLU(alpha=0.1)
    self.vol_1= Concatenate(axis=3)

    #Correlation layer declaration for CR2 same as CR1

    #Passing CR1 and CR2 through convolution layers
    self.conv_v1= Conv2D(256,5,strides=1,padding='same')
    self.conv_v1_act =LeakyReLU(alpha=0.1)
    self.conv_v2= Conv2D(256,5,strides=1,padding='same')
    self.conv_v2_act =LeakyReLU(alpha=0.1)
```

```
39
40    #Correlation between volumes CR1 and CR2
41    self.cr_3_act= LeakyReLU(alpha=0.1)
42    self.conv_v1_redir= Conv2D(32,1,strides=1)
43    self.conv_v1_redir_act =LeakyReLU(alpha=0.1)
44    self.vol_3= Concatenate(axis=3)
45
46    #Single seven convolution stream
47    self.conv3_1= Conv2D(256,3,strides=1,padding='same')
48    self.conv3_1_act= LeakyReLU(alpha=0.1)
49    self.conv4= Conv2D(512,3,strides=2,padding='same')
50    self.conv4_act= LeakyReLU(alpha=0.1)
51    self.conv4_1= Conv2D(512,3,strides=1,padding='same')
52    self.conv4_1_act= LeakyReLU(alpha=0.1)
53    self.conv5= Conv2D(512,3,strides=2,padding='same')
54    self.conv5_act= LeakyReLU(alpha=0.1)
55    self.conv5_1= Conv2D(512,3,strides=1,padding='same')
56    self.conv5_1_act= LeakyReLU(alpha=0.1)
57    self.conv6= Conv2D(1024,3,strides=2,padding='same')
58    self.conv6_act= LeakyReLU(alpha=0.1)
59    self.conv6_1= Conv2D(1024,3,strides=1,padding='same')
60    self.conv6_1_act= LeakyReLU(alpha=0.1)
61
62
63  #Layer Definition
64  def call(self, input1,input2,input3,training=False):
65    #Layer Definition for Input path-1
66    i_1=self.i1(input1)
67    cona1=self.conva_1(i_1)
68    cona1_act=self.conva_1_act(cona1)
69    cona2=self.conva_2(cona1_act)
70    cona2_act=self.conva_2_act(cona2)
71    cona3=self.conva_3(cona2_act)
72    cona3_act=self.conva_3_act(cona3)
73
74    #Layer Definition for Input path-2 same as Input path-1
75
76    #Layer Definition for Input path-3 same as Input path-1
77
```

```
78      #Layer Definition for Correlation volume CR1
79      cc1=self.cc([cona3_act,conb3_act])
80      cc1_act=self.cr_1_act(cc1)
81      cona_r=self.conva_redir(cona3_act)
82      cona_r_act=self.conva_redir_act(cona_r)
83      v1=self.vol_1([cc1_act,cona_r_act])
84
85      #Layer Definition for Correlation volume CR2 same as CR1
86
87      #Layer Definition for passing CR1 and CR2 through convolutions
88
89      #Layer Definition for Correlation volume CR3
90      cc3=self.cc([con_v1_act,con_v2_act])
91      cc3_act=self.cr_3_act(cc3)
92      con_v1_r=self.conv_v1_redir(con_v1_act)
93      con_v1_r_act=self.conv_v1_redir_act(con_v1_r)
94      v3=self.vol_3([cc3_act,con_v1_r_act])
```

The code snippet above explains the implementation of the contractive network of the MultiFlow model. In the *def_init_* function on line 7-9, input layers are defined where the image frame triplets are taken as input. Line 12-17 indicates the convolution layers used for the first input stream. Line 24 declares the correlation layer with the arguments. Line 28 declares the convolution layer used for concatenating with the correlation volume, and line 30 declares the concatenation layer. Line 35-38 declares the convolution layers through which correlation volumes are passed. Line 41-44 declares the correlation layer for performing correlation among correlation volumes. Line 47-60 declares the convolution layer for the convolution stream. In the *def call* function, all the layers declared above are called, line 66-72 performs the convolution operations for the first input stream. Line 79 performs the correlation operation, and line 83 performs the concatenation of the correlation with the convolution layer first input stream. Line 90 performs the correlation among the correlation volumes.

Table 4.1 gives detailed information for all the layers used to develop the contractive network of the MultiFlow model. *redir* indicates the feature maps which are used to concatenate with the correlation volumes. Apart from the input layers, all the other layers use LeakyReLU (Leaky Rectified Linear Units) as their activation function since the original FlowNetC model given in [DFI+15] uses LeakyReLU as the activation function. All the arguments for the convolution layers and correlation layers are taken from the original FlowNetC model.

Layer Name	Layer Type	Kernel Size	Filters	Activation	Stride
Image I_1	Input Layer	-	-	-	
Image I_2	Input Layer	-	-	-	
Image I_3	Input Layer	-	-	-	
Conv1	Convolution Layer	7 x 7	64	Leaky ReLU	2
Conv2	Convolution Layer	5 x 5	128	Leaky ReLU	2
Conv3	Convolution Layer	5 x 5	256	Leaky ReLU	2
Conv3_redir	Convolution Layer	1 x 1	32	Leaky ReLU	1
CR1	Correlation Layer	-	-	Leaky ReLU	
CR2	Correlation Layer	-	-	Leaky ReLU	
Conv_cr1	Convolution Layer	5 x 5	256	Leaky ReLU	1
Conv_cr2	Convolution Layer	5 x 5	256	Leaky ReLU	1
CR3	Correlation Layer	-	-	Leaky ReLU	
Conv_cr1_redir	Convolution Layer	1 x 1	32	Leaky ReLU	1
Conv3_1	Convolution Layer	3 x 3	256	Leaky ReLU	1
Conv4	Convolution Layer	3 x 3	512	Leaky ReLU	2
Conv4_1	Convolution Layer	3 x 3	512	Leaky ReLU	1
Conv5	Convolution Layer	3 x 3	512	Leaky ReLU	2
Conv5_1	Convolution Layer	3 x 3	512	Leaky ReLU	1
Conv6	Convolution Layer	3 x 3	1024	Leaky ReLU	2
Conv6_1	Convolution Layer	3 x 3	1024	Leaky ReLU	1

Table 4.1: Layer Details of Contractive Network of MultiFlow

4.2.2 Refinement Network of MultiFlow

The refinement network of the MultiFlow model consists of transposed convolution layers. Due to the addition of the correlation and convolution layers in the contractive network of the MultiFlow model, the refinement network also gets an additional upsampled flow.

- **Modifications:**
  As opposed to the original refinement network architecture of FlowNetC, where four upsampled flows are present, in MultiFlow, five upsampled flows are generated. This can be seen in Figure 4.7. An additional upsampled flow arises due to the presence of convolution $Conv\_cr1$ generated from the correlation volume $CR1$. Additional skip connection from the contractive network is also used to refine the upsampled flow prediction. Also, weights values are multiplied with upsampled flows while training the neural network. This leads to a better prediction of the flow vectors which, are then upsampled to obtain better dense per pixel predictions in original resolution. Figure 4.7 illustrates the refinement network of MultiFlow. The modifications are highlighted in the red box. The figure is present at the end of the chapter.

```
class MultiFlow(tf.keras.Model):
  # Layer declaration
  def __init__(self):
    super(MultiFlow, self).__init__()

    #Refinement Network
    #Layer declaration for Level 1 of Refinement Network
    self.pf6= Conv2D(2,3,strides=1,padding='same') #Predicted flow
    self.dc5= Conv2DTranspose(512,4,strides=2,padding='same') #Transpose
        convolution
    self.dc5_act= LeakyReLU(alpha=0.1)
    self.up_6to5= Conv2DTranspose(2,4,strides=2,padding='same') #Upsampled flow
    #Concating 3 streams convolution-features, transposed convolution,
        upsampled_flow
    self.con_5= Concatenate(axis=3)

    #Layer declaration for Level 2, 3, 4, 5 of Refinement Network same as Level 1

    #Final Predicted Flow
    self.pf1= Conv2D(2,3,strides=1,padding='same')

  #Layer Definition
  def call(self, input1,input2,input3,training=False):
    pf_6=self.pf6(con6_1_act)
    dc_5=self.dc5(con6_1_act)
    dc_5_act=self.dc5_act(dc_5)
    ups_6to5=self.up_6to5(pf_6)
    concat5=self.con_5([con5_1_act,dc_5_act,ups_6to5])

    pf_1=self.pf1(concat1)

    #Bilinear upsampling
    flow=tf.image.resize(pf_1,tf.stack([436,1024]),method='bilinear')

    return {'flow': flow ,'predict_flow6': pf_6, 'predict_flow5': pf_5,
        'predict_flow4': pf_4, 'predict_flow3': pf_3, 'predict_flow2': pf_2,
        'predict_flow1': pf_1}
```

The code snippet above explains the implementation of the refinement network of the MultiFlow model. In the *def_init_* function on line 8-13, the layers of the first refinement level are declared. Line 9 indicates the transposed convolution layer for upsampling the refinement level later used in the *def call* function. Line 11 declares the transposed convolution layer used for upsampling the predicted optical flow at each level. Line 13 declares the concatenation layer. In the *def call* function, all the layers declared above are called, line 22-26 performs the first upsampling of the predicted optical flow. Line 28 indicates the final predicted optical flow, which is then upsampled using bilinear interpolation on line 30. Line 32 returns the predicted flow at each refinement level for loss computation.

Table 4.2 gives detailed information for all the layers of the refinement network of the MultiFlow neural model. The *Transposed Conv* in the Layer column indicates transposed convolution layers. The transposed convolution layers use LeakyReLU (Leaky Rectified Linear Units) as their activation function since the original FlowNetC model given in [DFI+15] uses LeakyReLU as the activation function. All the arguments for the convolution and transposed convolution layers are taken from the original FlowNetC model.

Layer Name	**Layer Type**	**Kernel**	**Filters**	**Activation**	**Stride**
Predicted _Flow6	Convolution Layer	3 x 3	2	-	1
Deconv5	Transposed Conv Layer	4 x 4	512	Leaky ReLU	2
Upsampled Flow_6to5	Transposed Conv Layer	4 x 4	2	-	2
Concat5	Concatenation Layer	-	-	-	-
Predicted_Flow5	Convolution Layer	3 x 3	2	-	1
Deconv4	Transposed Conv Layer	4 x 4	256	Leaky ReLU	2
Upsampled Flow_5to4	Transposed Conv Layer	4 x 4	2	-	2
Concat4	Concatenation Layer	-	-	-	-
Predicted_Flow4	Convolution Layer	3 x 3	2	-	1
Deconv3	Transposed Conv Layer	4 x 4	128	Leaky ReLU	2
Upsampled Flow_4to3	Transposed Conv Layer	4 x 4	2	-	2
Concat3	Concatenation Layer	-	-	-	-
Predicted_Flow3	Convolution Layer	3 x 3	2	-	1
Deconv2	Transposed Conv Layer	4 x 4	128	Leaky ReLU	2
Upsampled Flow_3to2	Transposed Conv Layer	4 x 4	2	-	2
Concat2	Concatenation Layer	-	-	-	-
Predicted_Flow2	Convolution Layer	3 x 3	2	-	1
Deconv1	Transposed Conv Layer	4 x 4	64	Leaky ReLU	2
Upsampled Flow_2to1	Transposed Conv Layer	4 x 4	2	-	2
Concat1	Concatenation Layer	-	-	-	-
Predicted_Flow1	Convolution Layer	3 x 3	2		1
Flow	Bilinear Interpolation (4X)	-	-	-	-

Table 4.2: Layer Details of Refinement Network of MultiFlow

Table 4.3 gives detailed information of all the inputs to each of the concatenation layers in the refinement network of the MultiFlow model.

Concatenation Layer	Inputs
Concat5	Conv5_1, Deconv5, Upsampled Flow_6to5
Concat4	Conv4_1, Deconv4, Upsampled Flow_5to4
Concat3	Conv3_1, Deconv3, Upsampled Flow_4to3
Concat2	Conv_cr1, Deconv2, Upsampled Flow_3to2
Concat1	Conv2, Deconv1, Upsampled Flow_2to1

Table 4.3: Inputs to the Concatenation Layer of Refinement Network of the MultiFlow Model

4.3 Training Procedures

4.3.1 Hardware and Software Resources

- **Programming Language:** Python.
- **Machine Learning Framework:** TensorFlow GPU 2.1.0, Keras API.
- **Data Handling:** Numpy and Pandas library from Python.
- **Integrated Development Environement (IDE):** Jupyter Notebook.
- **Graphics Processing Unit (GPU):** Google Colab GPUs namely Nvidia K80s, T4s, P4s and P100s (dynamic allocation), GET Lab GPUs (Nvidia GTX 1080Ti).

4.3.2 Multi-Scale Loss Function

The loss function used to train the MultiFlow model is a multiscale weighted loss function. The multi-scale loss function is also used to train the original FlowNetC, FlowNetS model given in [DFI+15], PWC-Net model given in [SYLK18] and PWC-Fusion model given in [RGS+19]. However, the loss function is adapted to accommodate the additional predicted upsampled flow generated in the refinement network of the MultiFlow model. The loss function multiplies the predicted flow obtained at each level of the refinement network with a weight-factor denoting the importance of flow at that level. The weight values are obtained after performing several experiments and incorporates the implementation of the average

Predicted Flows	Weights
predicted_flow6	0.32
predicted_flow5	0.32
predicted_flow4	0.32
predicted_flow3	0.32
predicted_flow2	0.32
predicted_flow1	0.64

Table 4.4: Weights for Predicted Flows

end-point-error function for the loss computation of the MultiFlow model. The weights used with the predicted flows are as follows:

The final predicted_flow1 has a weight value double than that of all the previous predicted flow. The last predicted flow contributes to the MultiFlow model's most critical flow because this flow is upscaled using bilinear interpolation. The idea behind associating weight factors with the predicted flows is to penalize the predictions at early levels to obtain more refined flows in subsequent levels of the refinement network. Figure 4.8 illustrates the implementation of upsampled flows along with their respective weights. The figure is present at the end of the chapter.

```
def loss_function(real, pred):

  pred6 = pred['predict_flow6']
  size = [pred6.shape[1], pred6.shape[2]]
  df6 = tf.image.resize(real, tf.stack(size))
  epe6 = 0.32 * (a_epe(df6, pred6))

  #Similar loss computation for epe5, epe4, epe3, epe2

  pred1=pred['predict_flow1']
  size = [pred1.shape[1], pred1.shape[2]]
  df1 = tf.image.resize(real, tf.stack(size))
  epe1 = 0.64 * (a_epe(df1, pred1))

  #Loss addition
  loss = tf.math.add_n([epe6,epe5,epe4,epe3,epe2,epe1])

  return loss
```

The code snippet above indicates the multi-scale loss function used in this thesis. Line 3-6 indicates the block of code for computing the loss of predicted flow obtained at each refinement level. Line 4 extracts the dimension from the predicted flow and reshapes the ground-truth optical flow using these dimensions. The loss is computed in line 6 with the help of the average end-point-error discussed in Section 2.3. Line 11 adds all the losses and sends the value for the training process.

4.3.3 Hyperparameter Optimization - Variable Learning Rate

The process of training a neural network is dependent on the gradient descent optimization algorithm. This algorithm is used to find the minimum value of the loss function. The loss function can also be referred to as the cost function. The cost function can be a mean squared error or mean absolute error used for image classification task and binary cross-entropy in case of binary segmentation of images. In this thesis, the loss function is the multi-scale AEPE loss function. A neural network consists of various hyperparameters. Some of the hyperparameters are optimizer, batch size, and learning rate. Perhaps the most crucial hyperparameter is the learning rate of the neural network. After each epoch, the neural network weights are updated, and the target is to minimize the value of the loss function. The learning rate hyperparameter controls the amount by which these weights are updated. Hence, the learning rate helps the neural network to reach the minimum value for its loss function.

A large learning rate value results in the training process of a neural network to reach its state of convergence faster. Convergence is the state when loss function value reaches its minimum value. However, at the same time, a very large learning rate will make the model update its weights at a faster rate, possibly making the loss function skip its minimum value by overshooting over the minimum. On the other hand, a small learning rate for training will require more time to reach convergence. Also, a very small learning rate will increase the training time to an extent where the model will never converge. It is due to the reasons mentioned above, choosing an appropriate learning rate is a crucial task.

While training a small neural network, the learning rate is kept constant, since the neural network trains faster. However, while training deep neural networks such as MultiFlow, a constant learning rate can lead to the model not yielding desirable results. The reason for this is the number of parameters of a neural network. The MultiFlow consists of millions of parameters, and optimizing each parameter becomes a challenging task.

The MultiFlow model is trained using a variable learning rate to deal with the challenges mentioned above. A variable learning rate is one in which the learning rate increases or decreases (in our case decreases) as the training progresses in order to minimize the loss function. Figure 4.4 shows the nature of learning curves with constant and variable learning rate. From the figure, it can be noticed that when the learning rate is constant, the loss function skips the minimum value and overshoots, but with a variable learning rate, the loss function reaches the minimum value.

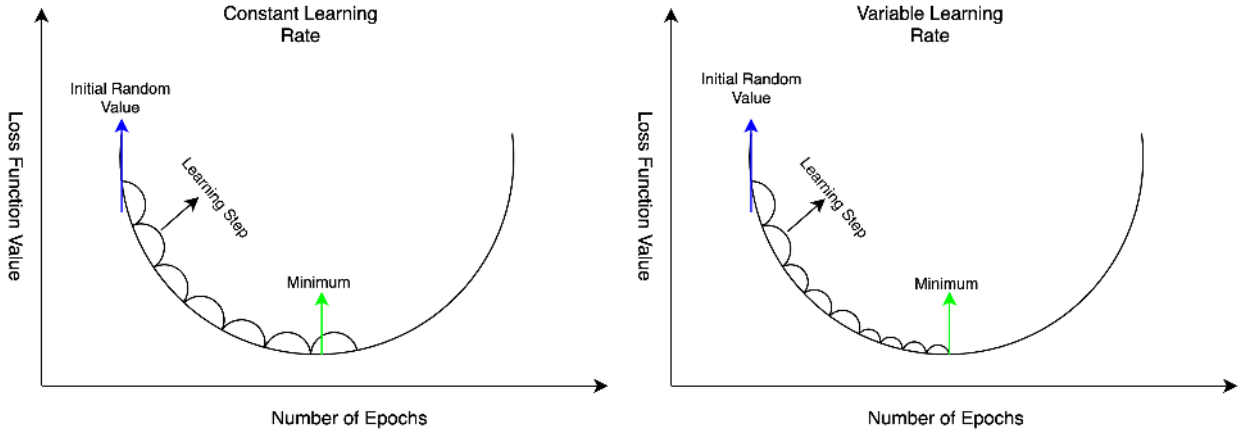


Figure 4.4: Constant Learning Rate versus Variable Learning Rate

The learning rate of the MultiFlow model is decreased after every certain number of epochs. The learning rate is decreased in 4 stages to train the MultiFlow model. Table 4.5 illustrates the learning rate at each stage. From the table, it can be seen that the learning rate is halved after every stage.

Learning Stage	**Learning Rate Value**
1	1×10^{-4}
2	5×10^{-5}
3	2.5×10^{-5}
4	1.25×10^{-5}

Table 4.5: Learning Rate Values

4.3.4 Training Challenges and Roadblocks

Training a neural network is a challenging task. Neural networks contain various hyper-parameters such as learning rate, optimizer, and batch size. These hyperparameters are

responsible for how well the neural network trains. If these parameters have the optimal value, then the neural network training performs faster, and the training converges.

While training a neural network, several procedures exist. These are as follows:

- The first procedure to train a neural network can be referred to as a complete training process. Depending on the task to solve, a neural network model is developed. After developing the neural network, the model is trained on the whole dataset from scratch to learn from the dataset and be able to solve the task. While training the network, the model's weights are initialized to a random value. As the training progresses, the weights are updated after each epoch. When the hyperparameters are set to an optimum value, the model's loss value starts to decrease, indicating that the model is starting to learn and improve itself, leading the model weights to reach accurate value. An example of a model using this type of training process could be to develop a model that performs image classification to identify whether the given image is of a cat or dog.
- The second procedure to train the neural network is referred to as transfer learning. In this procedure, the knowledge gained from solving one task is used to solve another task, which is similar to the task the model initially solved. Here, rather than training the whole neural model from scratch, the model is modified, and only the modified part of the model is trained on the dataset. The reason for doing this is because the model has already learned the general characteristics of the dataset, and now it should learn the specific characteristics. To achieve this, the part of the model that is not modified is frozen (weights of the neural network layers are not updated as training progresses), and the part of the model modified, in that the weights are updated after each epoch. An example of a model using this training process could be hierarchical image classification model in which the model not only classifies whether the given image is of a cat or dog but also classifies which breed the cat or dog belongs.

For training the MultiFlow model developed in this thesis, both of the above training procedures were tried. However, some roadblocks were encountered. These are described as follows:

1. **Complete Training:** This was the first training procedure used to train the model. In this training procedure, the model was trained on the whole dataset at once. The model consists of several million parameters. Also, the images used to train the MultiFlow model are of high resolution. Due to the image size and model size, the model required high training time. The model was trained for approximately 2600 epochs.

Every hundred epochs were completed in around 26 hours. Also, during the training procedure, the GET Lab GPU unexpectedly reinitialized the weights of the MultiFlow model. Hence, the model started retraining again from scratch. Figure 4.5 shows the results obtained from this technique. The images below are results obtained after training for 2600 epochs. The first image indicates the ground truth optical flow, and the rest of the images depict the predictions made at several epochs.

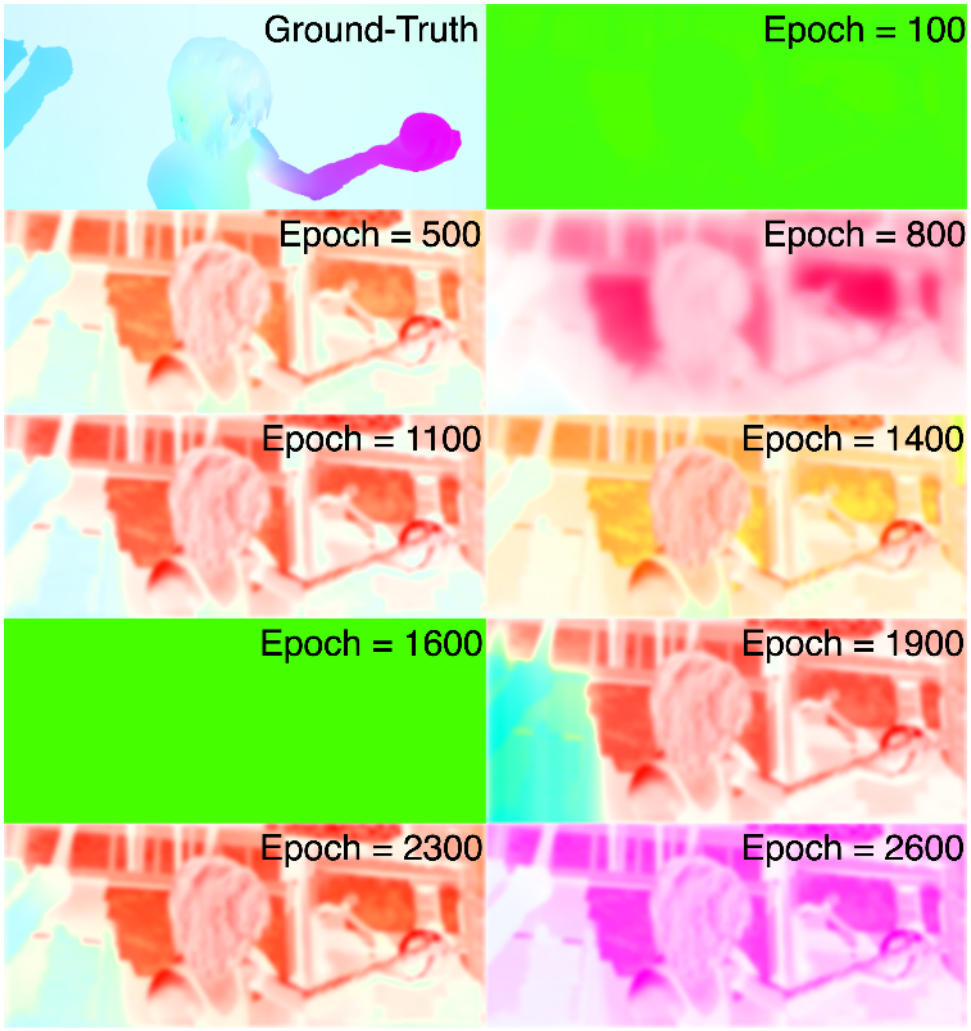


Figure 4.5: Results of Complete Training Process at Several Epochs

Solution to overcome the above roadblock: Google Colab GPUs is used to train the MultiFlow model and instead of training the MultiFlow model on the complete dataset, small subsets of the complete dataset are created for training the model.

2. **Transfer learning without freezing layers of the model:** This was the second training procedure used to train the model. In this training procedure, subsets of the whole dataset were created, and then the model was trained on each subset of the

dataset one after the other. However, the model did not perform well since the model, after being trained on the next subset of the dataset, forgets the previously learned data.

3. **Transfer learning with freezing layers of the model:** This was the third training procedure used to train the model. In this training procedure, the model was trained on the first subset of the data, and then the feature extraction layers of the model were frozen, not to forget the previously learned data. However, the MultiFlow model was not able to learn new vector displacements between the pixels of the image frames in the next subset due to freezing the feature extraction layers. Hence, it always gave the same result as before. Therefore, this procedure also did not work for training the MultiFlow model.

The second and third techniques did not solve the task of MultiFlow model learning. The reasons are as follows:

- Since the architectural computations of the MultiFlow model are different than other models, for feature extraction a pretrained optical flow model cannot be used.
- When creating subsets of the MPI Sintel dataset, a subset may not contain all possible types of pixel displacements. Hence selecting a model trained on only one subset of the dataset may give poor performance.

Since the training processes mentioned above proved to be either not useful or time-consuming, an alternate method is used to train the MultiFlow model.

Solution to overcome the above roadblocks: Instead of training the MultiFlow model on subsets of the dataset, train a MultiFlow model for each scene of the dataset using Google Colab GPU.

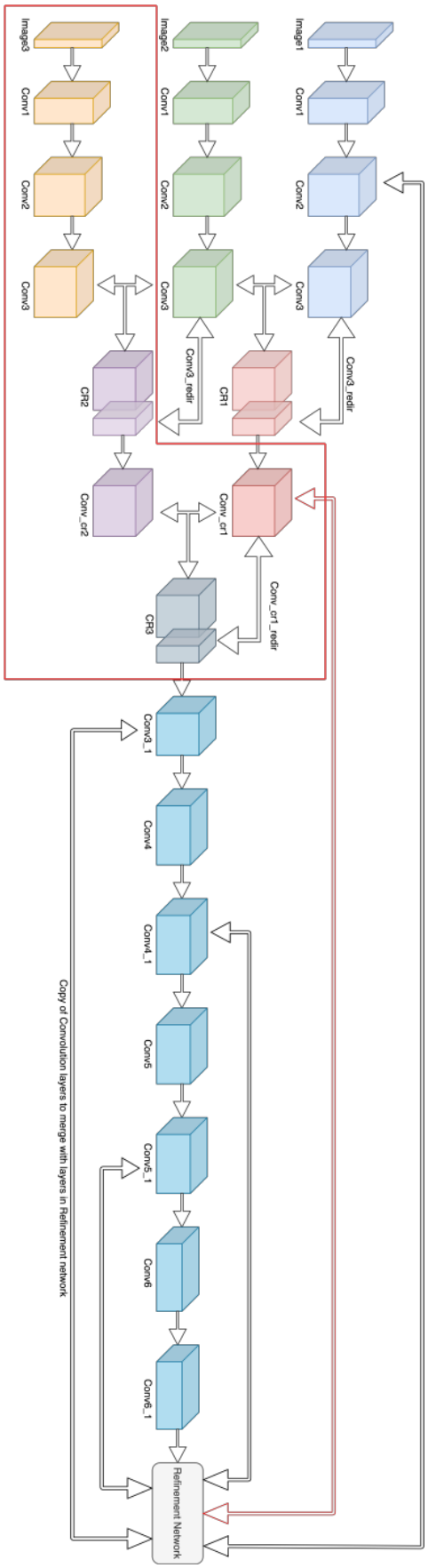


Figure 4.6: MultiFlow Architecture - Contractive Network

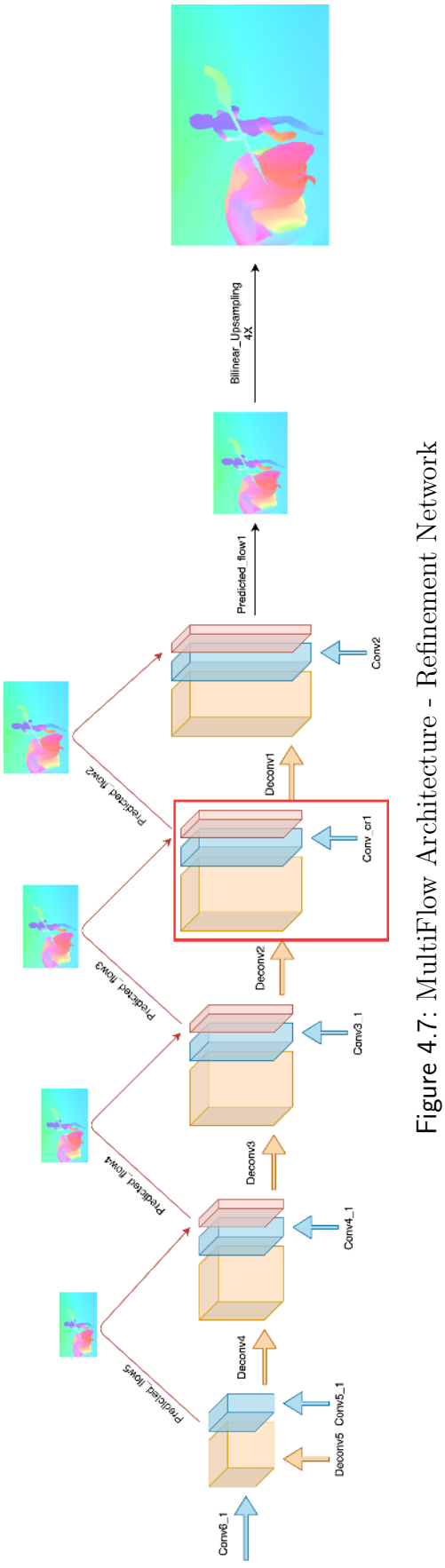


Figure 4.7: MultiFlow Architecture - Refinement Network

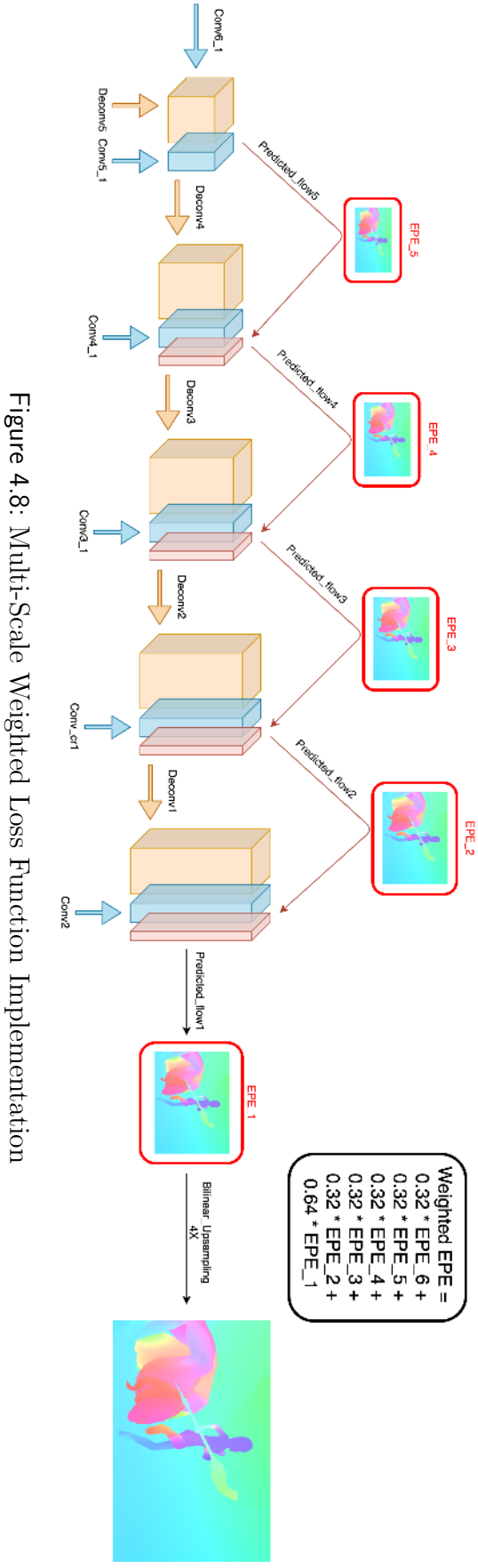


Figure 4.8: Multi-Scale Weighted Loss Function Implementation

5 MultiFlow Test Results

5.1 Final Training Procedure

The MPI Sintel dataset contains three categories of the image frames, depending on how they are rendered in the dataset. Each of the three categories contains 23 specific scenes in the dataset. To train the MultiFlow model on the whole dataset, the approach is to train a MultiFlow model for each specific scene. This leads to 23 different MultiFlow models of the dataset but, at the same time, ensures that the MultiFlow models can capture all possible vector displacements of the optical flow field. Each scene's accuracy is calculated using Average End-Point-Error (AEPE) in the dataset. The overall accuracy of the MultiFlow model is the average of the AEPE obtained from all 23 scenes. The MultiFlow models in this thesis are trained on the final render category of the dataset because that category tries to depict as much realism as possible. This also ensures that if the MultiFlow models works well on the final render category, it will also work on the other render categories.

For each scene S_i, the overall Average End-Point-Error of the MultiFlow model is calculated as follows:

$$\frac{1}{23}\sum_{i=1}^{23} EPE(S_i) \tag{5.1}$$

The training parameters are as follows:

- **Train-Test Spilt:** 90-10 for each 23 scenes.
- **Number of Images:** 912 train image frames and 106 test image frames from all scenes.
- **Optimizer:** Adaptive moment estimation (Adam) with weight decay. The weight decay value is set to 0.004 with momentum parameters β_1=0.9 and β_2=0.999. The

parameters for the optimizer are taken from the original FlowNetC model given in [DFI+15].

- **Batch Size:** 4
- **Learning Rate:** Variable learning rate recursively reduced to half after every certain number of epochs. Initial learning rate=1×10^{-4}.
- **Correlation Layer Parameters:** k=0, s_1=1, s_2=2, $d = 20$. These values are taken in reference from [DFI+15].

Since each scene is trained for a different number of epochs, the decrease in the learning rate after a certain number of epochs is not the same for each scene. The MultiFlow models are trained and tested on the GPUs available by Google Colab.

Table 5.1 gives the training time required for each training procedure:

Training Procedure	Training Time
Complete Training	∼1 Month
Transfer learning without freezing layers of the model	∼2 Weeks
Transfer learning with freezing layers of the model	∼2 Weeks
Final Training Procedure	∼3 Weeks

Table 5.1: Training Time Requirement for Various Training Procedures

Results obtained using the final training procedure show that the MultiFlow model developed indeed obtains excellent results and therefore ensures its validity and correctness.

5.2 Results

The MultiFlow model's performance is evaluated on the final render category of the MPI Sintel dataset. Table 5.2 depicts the Average End-Point-Error (AEPE) obtained by the different neural network models.

The claim that the scene-specific MultiFlow models outperforms the existing generalized models is based on the following line of reasoning:

- Consider a CNN model trained on the whole final render category of the MPI Sintel dataset. If the performance of the CNN model will be evaluated for each scene separately similar to the evaluation procedure of the MultiFlow model, then in that case, the overall AEPE value of the CNN model will still remain the same if the evaluation

was to be performed on whole test data at once. The only difference is that the CNN model is a generalized model, whereas the MultiFlow is a scene-specific model obtained in this thesis is due to the reasons mentioned in Chapter 6.

- Looking at the promising results of the scene-specific MultiFlow models, it can be expected that the generalized MultiFlow model will have the same performance as scene-specific models.

For the performance evaluation, all the models presented in this chapter use a separate test set from the MPI Sintel dataset. Due to the scene-specific MultiFlow models, the separate test set cannot be used for evaluating the MultiFlow models. However, as mentioned in Section 5.1, 10 % of the train set is used as the test set for the MultiFlow model. Since the images in the separate test set are similar to the scenes in the train set in terms of motion and texture characteristics, the performance comparison with other multi-frame CNN models is possible in this case.

Models	**Sintel Final (AEPE)**		**Runtime (ms)**	**GPU/CPU**
	Train	Test		
PWC-Fusion	N/A	4.566	N/A	N/A
MultiFlow	**4.020**	**4.970**	**28** **18**	**Google Colab** **Nvidia 1080Ti**
ProFlow	N/A	5.017	N/A	N/A
TIMCflow	N/A	5.049	N/A	N/A
MR-Flow	3.590	5.380	∼120000	i7 - CPU
FlowNetC+ft+v	4.830	7.880	1120	GTX Titan GPU
FlowNetC+ft	5.280	8.510	150	GTX Titan GPU

Table 5.2: Average End-Point-Error (in pixels) of Different Models

From the table, it can be inferred that the MultiFlow models outperforms all the existing models used for optical flow estimation by a margin of 1 to 8 %, albeit the PWC-Fusion model. The PWC-Fusion model performs slightly better than the MultiFlow models. A possible reason for the better performance is because the PWC-Fusion model is executed three times to compute forward and backward flow which are then combined to estimate the optical flow field. Even though the ProFlow model computes both the forward and the backward flow, the MultiFlow models outperforms the ProFlow model by only computing the forward flow. On the other hand, the MR-Flow model executes two neural networks for optical flow estimation, whereas the MultiFlow models only executes a single neural network and performs better than the MR-Flow model.

The MultiFlow models have a large performance gain over the best performing FlowNetC+ft+v

model by 36.92 %. FlowNetC+ft+v is trained on the Flying Chairs dataset and fine-tuned on the clean and final render category of the MPI Sintel dataset. Rather than performing 4X bilinear upsampling, FlowNetC+ft+v uses variational techniques in the post-processing stage to obtain the optical flow in original dimensions. Similarly, FlowNetC+ft is trained on the Flying Chairs dataset and fine-tuned on the clean and final render category of the MPI Sintel dataset but uses 4X bilinear upsampling for obtaining the optical flow in original dimensions. The MultiFlow models outperforms the FlowNetC+ft model by a more significant margin of 41.59 %. The large performance gain over the FlowNetC+ft+v and FlowNetC+ft models is because the MultiFlow models use image frame triplets to estimate the optical flow field. However, since the FlowNetC+ft+v and FlowNetC+ft models use the MPI Sintel dataset for performance evaluation, a comparison is possible in this case.

Table 5.2 also gives the runtime in milliseconds (ms) required for predicting a single optical flow field per image frame triplets. From the table, it can be observed that the MultiFlow model predicts and generates the flow fields faster than both the FlowNetC models. Although the runtime for PWC-Fusion, ProFlow, and TIMCflow model are available for the KITTI dataset, the runtime values are not given in the table since the dataset is different than the MPI Sintel dataset. Since the MR-Flow model is executed on the Central Processing Unit (CPU), a fair comparison of runtime with the MultiFlow models is not possible. For the MultiFlow model, two separate runtimes are present beacuse the model was executed on the online Google Colab environment and on the local machine (GET Lab GPU). The runtime of the Google Colab environment is slightly higher than the local machine due to the latency associated with data processing and transfer on the online environment.

Figures 5.1, 5.2, and 5.3 illustrates the optical flow fields predicted by the MultiFlow models from the image frame triplets of each of the 23 scenes of the final render category. The left column contains the image frame triplets of each scene overlayed on top of each other. The middle column shows the ground-truth optical flow field. The scene name is written inside each ground-truth optical flow. The right column shows the optical flow field predicted by the MultiFlow models. The EPE in the predictions depicts the value obtained for that particular optical flow field. These figures are present at the end of the chapter.

From the figures, it can be seen that the estimated optical flow fields from the MultiFlow models are very close to the ground-truth optical flow. The MultiFlow models captures pixel displacements accurately. The MultiFlow models are also able to preserve structural information from the image frames by estimating the flow fields accurately along the edges of the objects.

However, in some scenes such as *Ambush_2* and *Ambush_6*, the EPE is high. Since these scenes contain less than 20 image frame triplets, the MultiFlow models do not gather enough knowledge to learn and predict the optical flow fields. Also, an observation made by the authors of [DFI+15] states that, even though the predictions made by the original FlowNetC model looks very close to the ground-truth optical flow, the EPE is often large in some cases. This occurrence of large EPE is due to the favoritism of the end-point-error metric towards the over-smoothed regions of the ground-truth optical flow. Since the predictions made by the FlowNetC model are upscaled using bilinear interpolation, the predictions become a bit noisy, which leads to higher EPE in some cases. A similar kind of behaviour is also observed for predictions made by the MultiFlow models.

Figure 5.1: Optical Flow Fields Predicted by the MultiFlow Models - 1

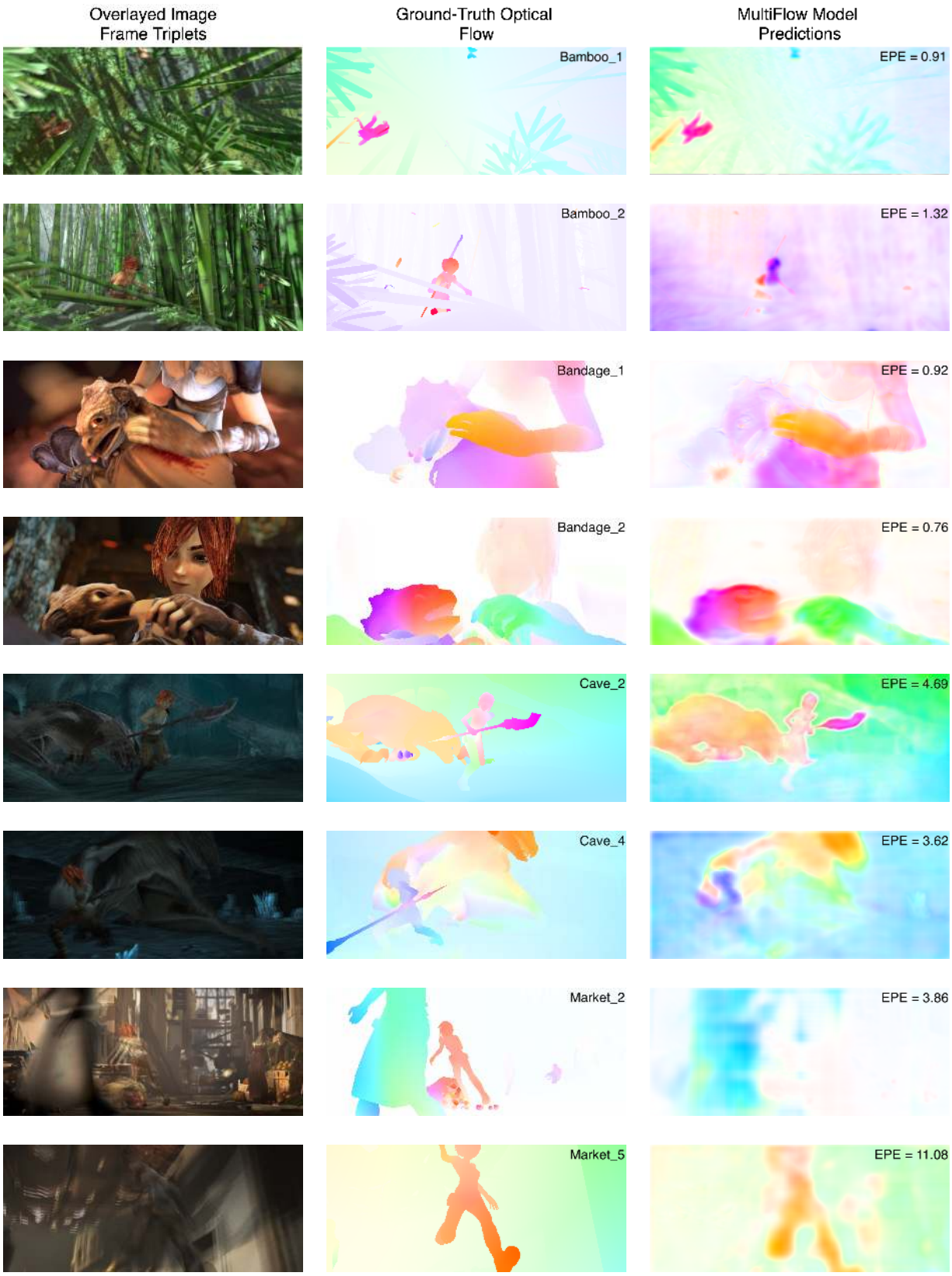


Figure 5.2: Optical Flow Fields Predicted by the MultiFlow Models - 2

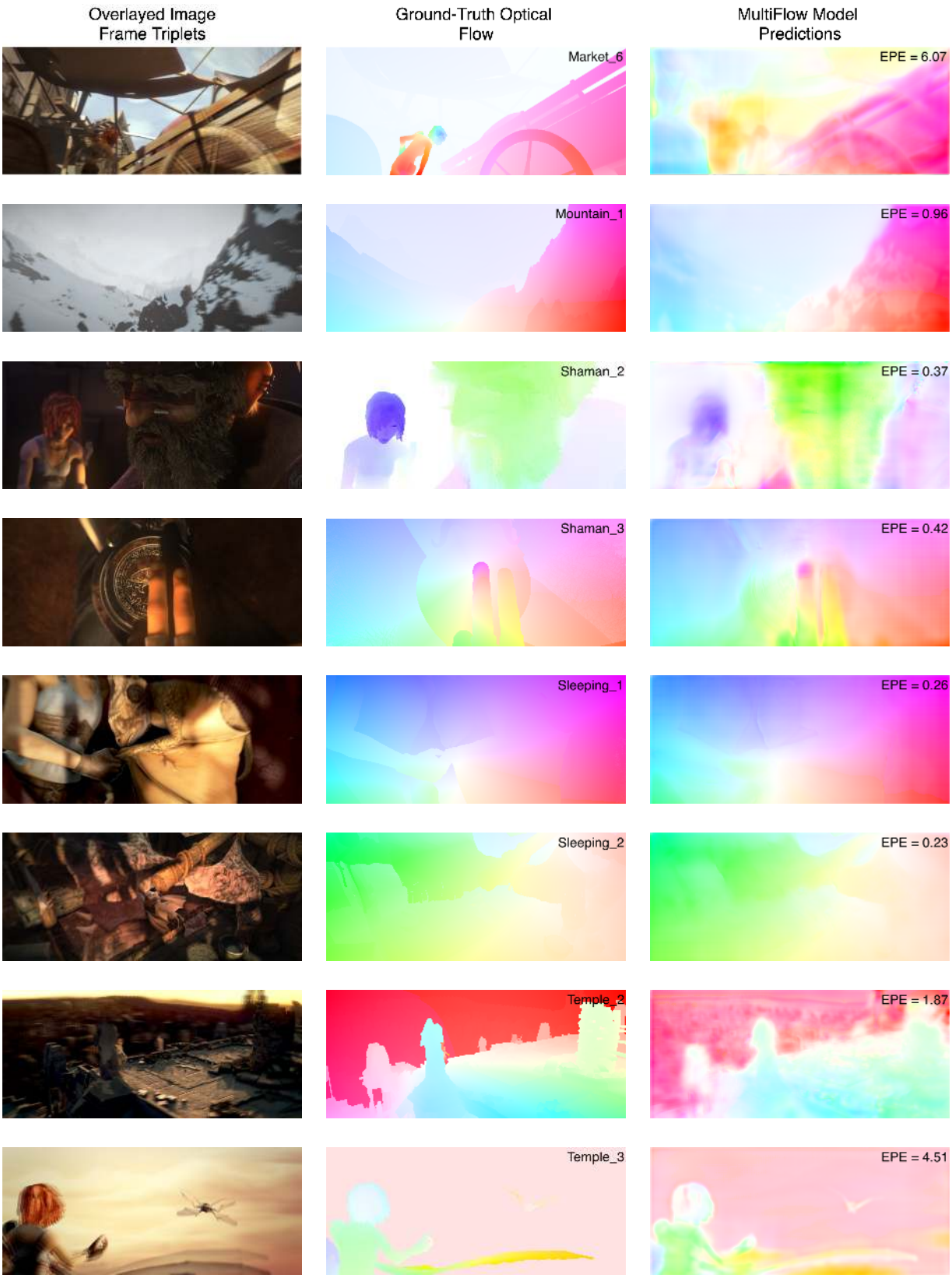


Figure 5.3: Optical Flow Fields Predicted by the MultiFlow Models - 3

6 Discussion

A neural network model has good generalization performance when it does not overfit on the test set. Overfitting is the state where test accuracy of the model is less than the train accuracy. The generalization performance of a neural network indicates how well the model performs in various scenarios. When the neural network model's generalization performance is good, it indicates that the model can achieve good performance on various datasets to solve the same task. On the other hand, if the generalization performance is poor, then the neural network needs to be trained better to achieve good performance.

In this thesis, the MultiFlow model is trained on the MPI Sintel dataset. The MultiFlow models obtained after training is not a single generalized model, but scene-specific models. The MultiFlow models are trained for each of the 23 different scenes separately and gives good results on the test data of each scene. Since the MultiFlow model is specifically trained and tested on one scene of the dataset, the model trained on one scene will not perform better on other scenes of the MPI Sintel dataset.

A generalized MultiFlow model was not obtained due to the following reasons:

1. **High Training time:** The original FlowNetC model has 39 Million trainable parameters. The MultiFlow model is bigger than the FlowNetC model and consists of 45 Million trainable parameters. Also, the MPI Sintel dataset used for training the MultiFlow model consists of image frames with resolution 436×1024 px. Due to the larger model size and image resolution, the MultiFlow model requires to be trained for a high number of epochs, which requires a high training time since the weights of each parameter needs to be fine-tuned after each epoch. The average time required to complete one epoch is approximately 14 minutes on Nvidia 1080Ti GPU.

2. **Insufficient Hardware Resources:** Deep neural networks are generally trained using clusters of GPUs. Due to the size of the MultiFlow model, multiple GPUs are needed to train the model. The currently available resources proved insufficient to train the MultiFlow model since these GPUs were not powerful enough to complete

the training process faster. Also, training deep CNN models like the MultiFlow model has high GPU memory requirement. Since training such deep neural networks requires high resources, the training hardware resources available to train the MultiFlow model proved insufficient.

3. **Unexpected GPU behavior:** Deep neural networks are executed on GPUs. The GPU is responsible for all the computations and sends the results back to the neural network to update the the weights of model. The MultiFlow model is also trained on GET Lab GPU. As discussed in Section 4.3.4, during the training process of the whole MPI Sintel dataset, a sudden spike in the loss function value in the magnitude of thousands for some number of epochs was observed. After looking into the loss records, it was found that the neural network had started relearning since the GPU had reinitialized all the weights computations, and the same predictions were repeated. To check this, the predictions made before and after the loss spike were compared. It was observed that these predictions matched with each other. Graph 6.1 depicts the loss function curve visualized in a log_{10} scale while training the MultiFlow model on the complete MPI Sintel dataset.

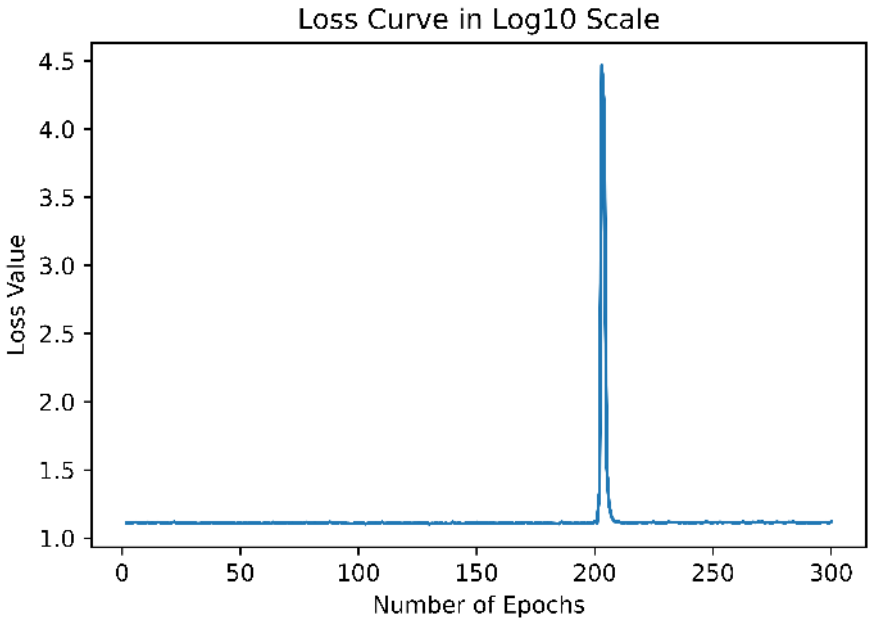


Figure 6.1: Loss Curve Spike during Neural Network Training on Complete MPI Sintel Dataset

To reconfirm the above mentioned behaviour of the GPU, the model was trained on half of the dataset on Nvidia 1080Ti GPU, and again same pattern was observed. Graph 6.2 depicts the loss function curve visualized in a log_{10} scale while training the model

on the partial MPI Sintel dataset. From both the graphs, it can be observed that the loss value spikes and then returns to the normal value in the middle of the training process. Due to this unexpected GPU behavior, training the MultiFlow model became time-consuming.

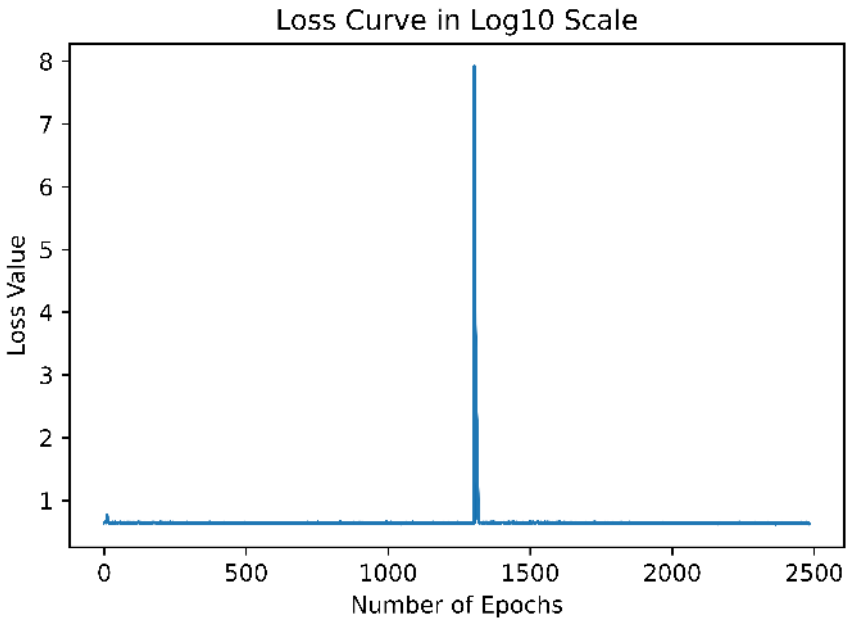


Figure 6.2: Loss Curve Spike during Neural Network Training on Half MPI Sintel Dataset

Due to the reasons mentioned above, there is no generalized MultiFlow model present in this thesis for evaluating all the image frames from the test set of the MPI Sintel dataset. Nonetheless, the scene-specific models performed good for all the scenes they were trained for and gave promising results on the test data.

7 Conclusion & Future Work

7.1 Conclusion

In this thesis, the objective of supervised optical flow estimation using more than two consecutive image frames with the help of a CNN model is successfully accomplished. This task is the extension of the standard optical flow estimation task, which only uses consecutive image pairs. For this purpose a novel CNN architecture called MultiFlow was developed which uses image frame triplets. One of the major contributions of this thesis was the development of the MultiFlow model which was based on the original FlowNetC architecture. The MultiFlow model can be referred as the advanced variant of the FlowNetC model.

In this thesis, the MPI Sintel dataset was chosen as the primary dataset for training the MultiFlow model since it met the requirement of consecutive image frame triplets. The MultiFlow models were trained and tested on the 23 scenes from the final render category of the MPI Sintel dataset. Results obtained from the MultiFlow models showed that it outperformed all the existing multi-frame approaches except the PWC-Fusion model. The MultiFlow models are able to capture pixel displacements in all possible directions and estimates the resulting optical flow field robustly. Additionally, the MultiFlow models are also able to preserve structural information along the edges of the objects thereby producing more accurate flow fields.

Various training procedures for training the MultiFlow model were implemented. A detailed study regarding their performance was carried out and a suitable training procedure was chosen to validate and verify the development of the MultiFlow model. Due to the hardware constraints, scene-specific MultiFlow models are obtained, trained for each scene separately. Hence, no generalized MultiFlow model is present in this thesis.

The inference of the MultiFlow model is performed by passing the image frame triplets to the MultiFlow model. The model is executed only once to estimate the optical flow field between the consecutive image frame triplets.

To conclude, the MultiFlow model is the second best model for optical flow estimation using multiple image frames. The model closely matches the top-performing model of PWC-Fusion. The results are promising and a step ahead in the right direction for optical flow estimation using consecutive image frame triplets.

7.2 Future Work

- The task of optical flow estimation can be further extended to more than image frame triplets such as image frame quadruplets, and the performance can be investigated.
- Given the size of the MultiFlow , a generalized MultiFlow model can be obtained with sufficient computing power. The performance of the generalized MultiFlow model can be compared with the scene-specific models from this thesis.
- Currently, the MultiFlow model does not work on occlusion handling between the image frame triplets. Further research can be done in this area to improve the performance of the MultiFlow model.

Bibliography

[BWSB12] Butler, Daniel J.; Wulff, Jonas; Stanley, Garrett B.; Black, Michael J.: A naturalistic open source movie for optical flow evaluation. In: *European conference on computer vision* Springer, 2012, pp. 611–625

[DFI+15] Dosovitskiy, Alexey; Fischer, Philipp; Ilg, Eddy; Hausser, Philip; Hazirbas, Caner; Golkov, Vladimir; Van Der Smagt, Patrick; Cremers, Daniel; Brox, Thomas: Flownet: Learning optical flow with convolutional networks. In: *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 2758–2766

[DT05] Dalal, Navneet; Triggs, Bill: Histograms of oriented gradients for human detection. In: *IEEE computer society conference on computer vision and pattern recognition (CVPR)* Bd. 1 IEEE, 2005, pp. 886–893

[DTSB15] Dosovitskiy, Alexey; Tobias Springenberg, Jost; Brox, Thomas: Learning to generate chairs with convolutional neural networks. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1538–1546

[Git19] Gituma, Mark: *Generating Optical Flow using NVIDIA flownet2-pytorch Implementation.* `https://towardsdatascience.com/generating-optical-flow-using-nvidia-flownet2-pytorch-implementation-d7b0ae6f8320`. Version: 2019. – Last accessed on 10 June 2020

[GLU12] Geiger, Andreas; Lenz, Philip; Urtasun, Raquel: Are we ready for autonomous driving? the kitti vision benchmark suite. In: *IEEE Conference on Computer Vision and Pattern Recognition* IEEE, 2012, pp. 3354–3361

[HS81] Horn, Berthold K.; Schunck, Brian G.: Determining optical flow. In: *Techniques and Applications of Image Understanding* Bd. 281 International Society for Optics and Photonics, 1981, pp. 319–331

[HZRS16] He, Kaiming; Zhang, Xiangyu; Ren, Shaoqing; Sun, Jian: Deep Residual Learning for Image Recognition. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, Las Vegas, NV, USA, June 27-30*, IEEE Computer Society, 2016, pp. 770–778

[IMS+17] Ilg, Eddy; Mayer, Nikolaus; Saikia, Tonmoy; Keuper, Margret; Dosovitskiy, Alexey; Brox, Thomas: FlowNet 2.0: Evolution of Optical Flow Estimation with Deep Networks. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR, Honolulu, HI, USA, July 21-26*, IEEE Computer Society, 2017, pp. 1647–1655

[JK19] Johnson, Justin M.; Khoshgoftaar, Taghi M.: Survey on deep learning with class imbalance. In: *J. Big Data* 6 (2019), pp. 27

[Lie] Lienen, Joris van: *Transposed Convolution.* `https://www.youtube.com/watch?v=96_oGE8WyPg&app=desktop`. – Last accessed on 5 December 2020

[LK81] Lucas, Bruce D.; Kanade, Takeo: An Iterative Image Registration Technique with an Application to Stereo Vision. In: Hayes, Patrick J. (Hrsg.): *Proceedings of the 7th International Joint Conference on Artificial Intelligence, IJCAI, Vancouver, BC, Canada, August 24-28*, William Kaufmann, 1981, pp. 674–679

[Lov02] Love, Bradley C.: Comparing supervised and unsupervised category learning. In: *Psychonomic bulletin & review* 9 (2002), Nr. 4, pp. 829–835

[Low04] Lowe, David G.: Distinctive image features from scale-invariant keypoints. In: *International journal of computer vision* 60 (2004), Nr. 2, pp. 91–110

[LSD15] Long, Jonathan; Shelhamer, Evan; Darrell, Trevor: Fully convolutional networks for semantic segmentation. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440

[MB18] Maurer, Daniel; Bruhn, Andrés: ProFlow: Learning to Predict Optical Flow. In: *British Machine Vision Conference 2018, BMVC 2018, Newcastle, UK, September 3-6, 2018* (2018), pp. 86

[MG15] Menze, Moritz; Geiger, Andreas: Object scene flow for autonomous vehicles. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3061–3070

[MHR18] Meister, Simon; Hur, Junhwa; Roth, Stefan: UnFlow: Unsupervised Learning of Optical Flow With a Bidirectional Census Loss. In: McIlraith, Sheila A. (Hrsg.); Weinberger, Kilian Q. (Hrsg.): *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, the 30th innovative Applications of Artificial Intelligence, and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7*, AAAI Press, 2018, pp. 7251–7259

[MO18] Mureşan, Horea; Oltean, Mihai: Fruit recognition from images using deep learning. In: *Acta Universitatis Sapientiae, Informatica* 10 (2018), Nr. 1, pp. 26–42

[PC15] Pinheiro, Pedro O.; Collobert, Ronan: From image-level to pixel-level labeling with convolutional networks. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1713–1721

[Pep17] Pepose, Sam: *FlowNet2 (TensorFlow)*. `https://github.com/sampepose/flownet2-tf`, 2017. – Last accessed on 14 April 2020

[Pin17] Pinard, Clement: *FlowNetPytorch*. `https://github.com/ClementPinard/FlowNetPytorch`, 2017. – Last accessed on 13 March 2020

[RFB15] Ronneberger, Olaf; Fischer, Philipp; Brox, Thomas: U-net: Convolutional networks for biomedical image segmentation. In: *International Conference on Medical image computing and computer-assisted intervention* Springer, 2015, pp. 234–241

[RGS+19] Ren, Zhile; Gallo, Orazio; Sun, Deqing; Yang, Ming-Hsuan; Sudderth, Erik; Kautz, Jan: A fusion approach for multi-frame optical flow estimation. In: *IEEE Winter Conference on Applications of Computer Vision (WACV)* IEEE, 2019, pp. 2077–2086

[Sah17] Saha, Sumit: *A Comprehensive Guide to Convolutional Neural Networks - the ELI5 way*. `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`. Version: 2017. – Last accessed on 16 May 2020

[SLJ+15] Szegedy, Christian; Liu, Wei; Jia, Yangqing; Sermanet, Pierre; Reed, Scott E.; Anguelov, Dragomir; Erhan, Dumitru; Vanhoucke, Vincent; Rabinovich, Andrew: Going deeper with convolutions. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR, Boston, MA, USA, June 7-12*, IEEE Computer Society, 2015, pp. 1–9

[SSJB16] Sevilla-Lara, Laura; Sun, Deqing; Jampani, Varun; Black, Michael J.: Optical Flow with Semantic Segmentation and Localized Layers. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR, Las Vegas, NV, USA, June 27-30*, IEEE Computer Society, 2016, pp. 3889–3898

[SYLK18] Sun, Deqing; Yang, Xiaodong; Liu, Ming-Yu; Kautz, Jan: Pwc-net: Cnns for optical flow using pyramid, warping, and cost volume. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 8934–8943

[SZ15] Simonyan, Karen; Zisserman, Andrew: Very deep convolutional networks for large-scale image recognition. In: *3rd International Conference on Learning Representations, ICLR, San Diego, CA, USA, May 7-9, Conference Track Proceedings* (2015)

[WRHS13] Weinzaepfel, Philippe; Revaud, Jerome; Harchaoui, Zaid; Schmid, Cordelia: DeepFlow: Large displacement optical flow with deep matching. In: *Proceedings of the IEEE international conference on computer vision*, 2013, pp. 1385–1392

[WSLB17] Wulff, Jonas; Sevilla-Lara, Laura; Black, Michael J.: Optical flow in mostly rigid scenes. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 4671–4680

[YCVDWM20] Yang, Fei; Cheng, Yongmei; Van De Weijer, Joost; Mozerov, Mikhail G.: Improved Discrete Optical Flow Estimation With Triple Image Matching Cost. In: *IEEE Access* 8 (2020), pp. 17093–17102

Appendix

List of Notations

- $\lfloor x \rfloor$ **:** Floor functions. Returns greatest values $\leq$ x
- *(x,y) :* Pixel at locations x, y
- $|x|$ **:** Absolute Value function. Returns absolute value of x
- $\in X$ **:** Is an element of X
- $\langle \rangle$ **:** Interval Parameter
- $\mathbb{R}$ **:** Set of Real Numbers

List of Abbreviations

- **SLAM:** Simultaneous Localization and Mapping
- **HOG:** Histogram of Gradients
- **SIFT:** Scale-Invariant Feature Transform
- **CNN:** Convolution Neural Networks
- **PWC:** Pyramid, Warping, Cost Volume
- **LSTM:** Long Short Term Memory
- **KITTI:** Karlsruhe Institute of Technology and Toyota Technological Institute
- **3D:** Three Dimensions
- **2D:** Two Dimensions
- **GANS:** Genetive Adversarial Networks

- **NLP:** Natural Language Processing
- **RGB:** Red, Green, Blue
- **AEPE:** Average End-Point-Error
- **EPE:** End-Point-Error
- **CR1:** Correlation Volume 1
- **CR2:** Correlation Volume 2
- **CR3:** Correlation Volume 3
- **ReLU:** Rectified Linear Units
- **API:** Application Program Interface
- **IDE:** Integrated Development Environment
- **GPU:** Graphics Processing Unit
- **Adam:** Adaptive Moment Estimation
- **ms:** Milliseconds
- **CPU:** Central Processing Unit

List of Architectures

- **DeepFlow:** Model for Optical Flow Estimation from [WRHS13]
- **FlowNetS:** Neural Network for Optical Flow Estimation from [DFI+15]
- **FlowNetC:** Neural Network for Optical Flow Estimation from [DFI+15]
- **FlowNet2:** Neural Network for Optical Flow Estimation from [IMS+17]
- **PWC-Net:** Neural Network for Optical Flow Estimation from [SYLK18]
- **ProFlow:** Neural Network for Optical Flow Estimation from [MB18]
- **PWC-Fusion:** Neural Network for Optical Flow Estimation from [RGS+19]
- **TIMCflow:** Neural Network for Optical Flow Estimation from [YCVDWM20]

- **MR-Flow:** Neural Network for Optical Flow Estimation from [WSLB17]
- **UnFlow:** Neural Network for Optical Flow Estimation from [MHR18]
- **U-Net:** Neural Network for Biomedical Image Segmentation from [RFB15]
- **GoogleNet:** Neural Network for Image Classification from [SLJ+15]
- **ResNet:** Neural Network for Image Classification from [HZRS16]
- **VGGNet:** Neural Network for Image Classification from [SZ15]

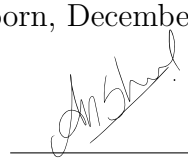
List of Tables

List of Figures

Declaration

I hereby confirm that I have prepared the submitted master thesis independently and have not used any sources or aids other than those indicated. Quotations were marked as such.

Paderborn, December 19, 2020



Anshul Suresh Bansal