Korea Advanced Institute of Science and Technology

School of Electrical Engineering

EE636 Digital Video Processing Fall 2018

Student's Name: Dinh Vu

Student's ID: 20184187

**Programming Assignment 3**

Two consecutive frames for the experiment are 10th and 11th frame, extracted from Calendar_CIF30.yuv. They are presented in the following Figure 1 and their resolution is 288×352. Because the input of FlowNet must be in RGB format, the YUV images, read by `YUV_READER.m`, have to be converted to RGB by using `ycbcr2rgb()` function in Matlab.

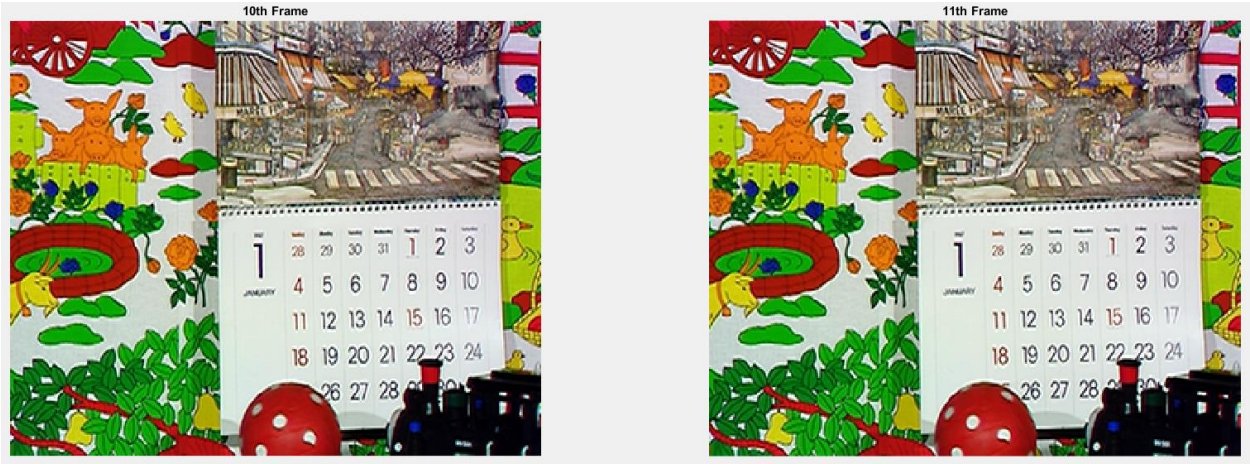


**Figure 1.** Two consecutive frames

## 1. Perform the motion estimation using the pre-trained FlowNet

All the following experiments are operated in the folder `FlowNetPyTorch_1`.

### *a. Compare the results of FlowNetS versus Lucas-Kanade and Horn-Schunck method*

The output of FlowNet Simple network is a RGB presentation of the optical flows, displayed in Figure 1.1 below.

**Figure 1.1.** The RGB map of the optical flows produced by FlowNetS

The size of this RGB map is 72×88, corresponding with the 4×4 block size because of the ratio $\frac{288}{72} = \frac{352}{88} = 4$. Therefore, in order to be compatible, the Lucas-Kanade and Horn-Schunck method must be performed with 4×4 block size.

The most importance problem is that the motion vectors, predicted by FlowNetS, must be recorded before converting to RGB format as in Figure 1.1. In order to save these original motion vectors, two following coding lines are added in `run_inference.py` (see line 98-99 of `run_inference.py` for easier understanding).

```
filemat = '../data/flow/frame_flow.mat'
savemat(filemat,{'frame_flow':np.transpose(flow_output.detach().cpu().
numpy(),(1,2,0))})
```

After getting the optical flows by `FlowNet.m`, the backward warping is applied in the motion compensation (`Motion_Compensation.m`) to reconstruct the frame. Figure 1.2 shows the 10$^{th}$ frame, the reconstruction frame and the difference between them by using Lucas-Kanade method, Horn-Schunck method and the FlowNet Simple network, respectively.


**Figure 1.2.** The results of FlowNetS versus Lucas-Kanade and Horn-Schunck method

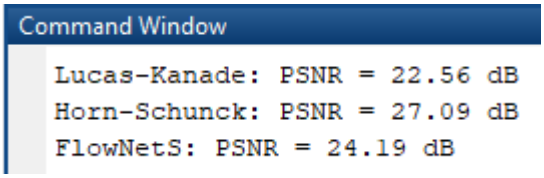


**Figure 1.3.** The PSNR of the reconstructed and the 10$^{th}$ frame

It is noticeable to see that the difference between the 10$^{th}$ frame and the reconstructed frame by FlowNetS is darker comparing to Lucas-Kanade method, however, some details are brighter than Horn-Schunck method. Moreover, Figure 1.3 presents that the frame reconstructed from motion vectors predicted by FlowNetS has the significant improvement about PSNR compare to Lucas-Kanade algorithm but still not high as Horn-Schunck

method. Overall, the FlowNet Simple network performs better than Lucas-Kanade method and not overcome to Horn-Schunck method.

***b. Compare the results of FlowNetC versus Lucas-Kanade and Horn-Schunck method***


**Figure 1.4.** The results of FlowNetC versus Lucas-Kanade and Horn-Schunck method

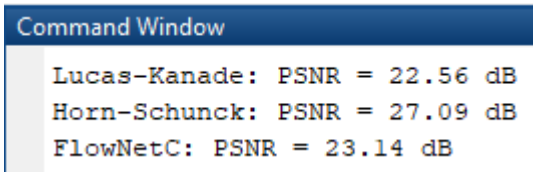


**Figure 1.5.** The PSNR of the reconstructed and the 10$^{th}$ frame

The results of FlowNetC is lightly better than Lucas-Kanade method and still not surpass Horn-Schunck method.

***c. Compare the results of FlowNetS versus FlowNetC***

It can be seen in Table 1.1 below, that FlowNetS has better performance than FlowNetC. L1 error is the mean of difference between the reconstruction and 10$^{th}$ frame.

**Table 1.1.** The comparison between 4 models

| Method | Lucas-Kanade | Horn-Schunck | FlowNetS | FlowNetC |
|---|---|---|---|---|
| PSNR (dB) | 22.56 | 27.09 | 24.19 | 23.14 |
| L1 error | 10.0553 | 6.3376 | 8.6069 | 9.4800 |

**2. With the FlowNet Simple network, perform the following task**

All the following experiments are operated in the folder `FlowNetPyTorch_2`.

### *a. Adjust the number of convolution layers in FlowNetS*

Four convolutional layers are added in the encoder of the FlowNet Simple network, which makes the network deeper. The other components of FlowNetS are unchanged. For more details, Figure 2.1 shows the encoder of FlowNetS before modifying while Figure 2.2 presents the new one.

```
18          self.conv1   = conv(self.batchNorm,   6,   64, kernel_size=7, stride=2)
19          self.conv2   = conv(self.batchNorm,  64,  128, kernel_size=5, stride=2)
20          self.conv3   = conv(self.batchNorm, 128,  256, kernel_size=5, stride=2)
21          self.conv3_1 = conv(self.batchNorm, 256,  256)
22          self.conv4   = conv(self.batchNorm, 256,  512, stride=2)
23          self.conv4_1 = conv(self.batchNorm, 512,  512)
24          self.conv5   = conv(self.batchNorm, 512,  512, stride=2)
25          self.conv5_1 = conv(self.batchNorm, 512,  512)
26          self.conv6   = conv(self.batchNorm, 512, 1024, stride=2)
27          self.conv6_1 = conv(self.batchNorm,1024, 1024)
```

**Figure 2.1.** The original encoder of FlowNetS

```
18          self.conv1   = conv(self.batchNorm,   6,   64, kernel_size=7, stride=2)
19          self.conv2   = conv(self.batchNorm,  64,  128, kernel_size=5, stride=2)
20          self.conv3   = conv(self.batchNorm, 128,  256, kernel_size=5, stride=2)
21          self.conv3_1 = conv(self.batchNorm, 256,  256)
22          self.conv3_2 = conv(self.batchNorm, 256,  256)  # Add
23          self.conv4   = conv(self.batchNorm, 256,  512, stride=2)
24          self.conv4_1 = conv(self.batchNorm, 512,  512)
25          self.conv4_2 = conv(self.batchNorm, 512,  512)  # Add
26          self.conv5   = conv(self.batchNorm, 512,  512, stride=2)
27          self.conv5_1 = conv(self.batchNorm, 512,  512)
28          self.conv5_2 = conv(self.batchNorm, 512,  512)  # Add
29          self.conv6   = conv(self.batchNorm, 512, 1024, stride=2)
30          self.conv6_1 = conv(self.batchNorm,1024, 1024)
31          self.conv6_2 = conv(self.batchNorm,1024, 1024)  # Add
```

**Figure 2.2.** More convolutional layers in the encoder of FlowNetS

### *b. The new loss function*

The original loss function for training the FlowNet Simple network is the multiscale end-point error, multiscaleEPE(), defined below (see line 243 in the `main.py` file)

```
loss_epe = multiscaleEPE(output, target,
weights=args.multiscale_weights, sparse=args.sparse)
```

In order to add the smoothness constrain to the loss function the smoothness loss is defined in Figure 2.3 below.

```
243         flow_out = output[0]
244         _, _, h, w = flow_out.shape
245         dx = flow_out[:,:,0:h-1,:] - flow_out[:,:,1:h,:]
246         dy = flow_out[:,:,:,0:w-1] - flow_out[:,:,:,1:w]
247         loss_x = torch.mean(abs(dx))
248         loss_y = torch.mean(abs(dy))
249         loss_smoothness = loss_x + loss_y
```

**Figure 2.3.** The smoothness loss

The novel loss function is the summation of the multiscale end-point error and the smoothness loss with the penalized parameter $\lambda = 1.0$, chosen after several experiments.

```
253         lambda_val = 1.0
254         loss = loss_epe + lambda_val*loss_smoothness
```

**Figure 2.4.** The novel loss function

In the main.py file, the novel loss is written from line 243 to line 254, depicted in Figure 2.3 and 2.4

### *c. Compare the results versus Lucas-Kanade and Horn-Schunck method*

The training process, which is configured with batch size equals to 8, number of epochs is 100 and the initial learning rate equals to 0.0001, consumes more than 10 hours with an Intel(R) Core(TM) i7 CPU @ 4.20 GHz and a GPU NVIDIA GTX GeForce 960Ti. The reason of choosing that small number of epochs because my GPU is not powerful, hence, it takes very long time if training with large number of epochs. Moreover, the main purpose of the experiments is that how can reuse the provided model and prove the potential of the neural network in motion estimation. Figure 2.5, 2.6 and 2.7 below present what happened during the training process.

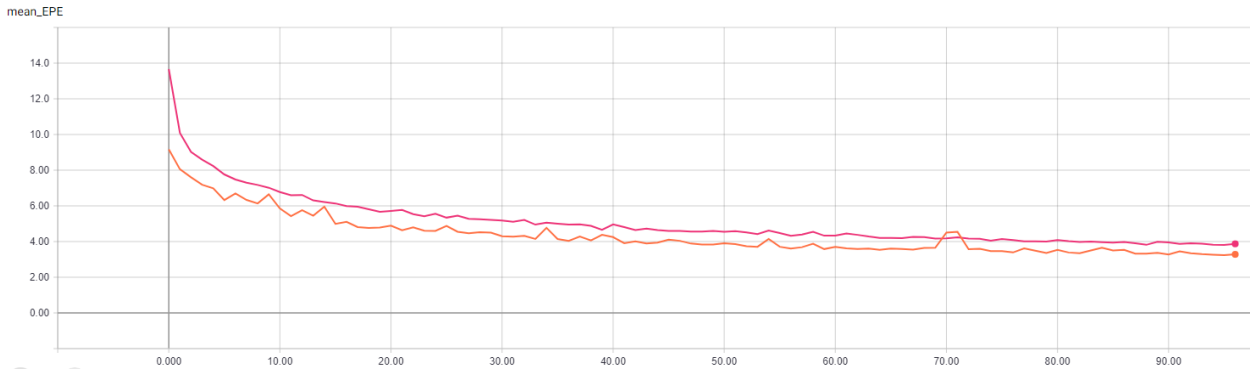


**Figure 2.5.** The end-point error at every epoch

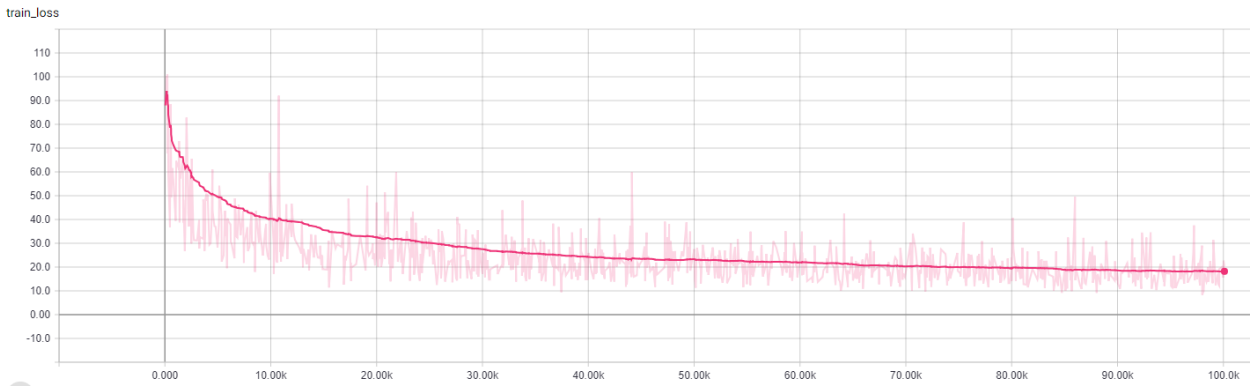


**Figure 2.6.** The training loss at every iteration

During the training process, the end-point error and training loss are decreased. However, because, the number of epochs is not high enough, the network still did not reach the optimal point. It can be seen in Figure 2.5 and 2.6, the end-point error and training loss are not converged. If the training is continuous, they will reduce slowly. On the other hand, three examples of validation set are tested during the training and the final outputs are displayed in Figure 2.7. There are still large differences between the predictions and ground truth. However, as mention before, it can be improved by increasing the number of epochs.


**Figure 2.7.** The validation examples

Figure 2.8, Figure 2.9 and Table 2.1 show the evaluation results of the modified FlowNet Simple network trained with the new loss function. The experiments imply that the new neural network performed not good as the original ones, even not similar to the

conventional algorithms. However, these outcomes are caused by the limitation of the training time. The evaluation model is not the convergence.

```
Command Window
Lucas-Kanade: PSNR = 22.56 dB
Horn-Schunck: PSNR = 27.09 dB
New FlowNetS: PSNR = 21.79 dB
```

**Figure 2.9.** The RGB map of the optical flows produced by the new FlowNetS (left) and the PSNR of the reconstructed and the 10$^{th}$ frame (right)

**Table 2.1.** The comparison between 5 models

| Method | Lucas-Kanade | Horn-Schunck | FlowNetS | FlowNetC | New FlowNetS |
|---|---|---|---|---|---|
| PSNR (dB) | 22.56 | 27.09 | 24.19 | 23.14 | 21.79 |
| L1 error | 10.0553 | 6.3376 | 8.6069 | 9.4800 | 11.2856 |

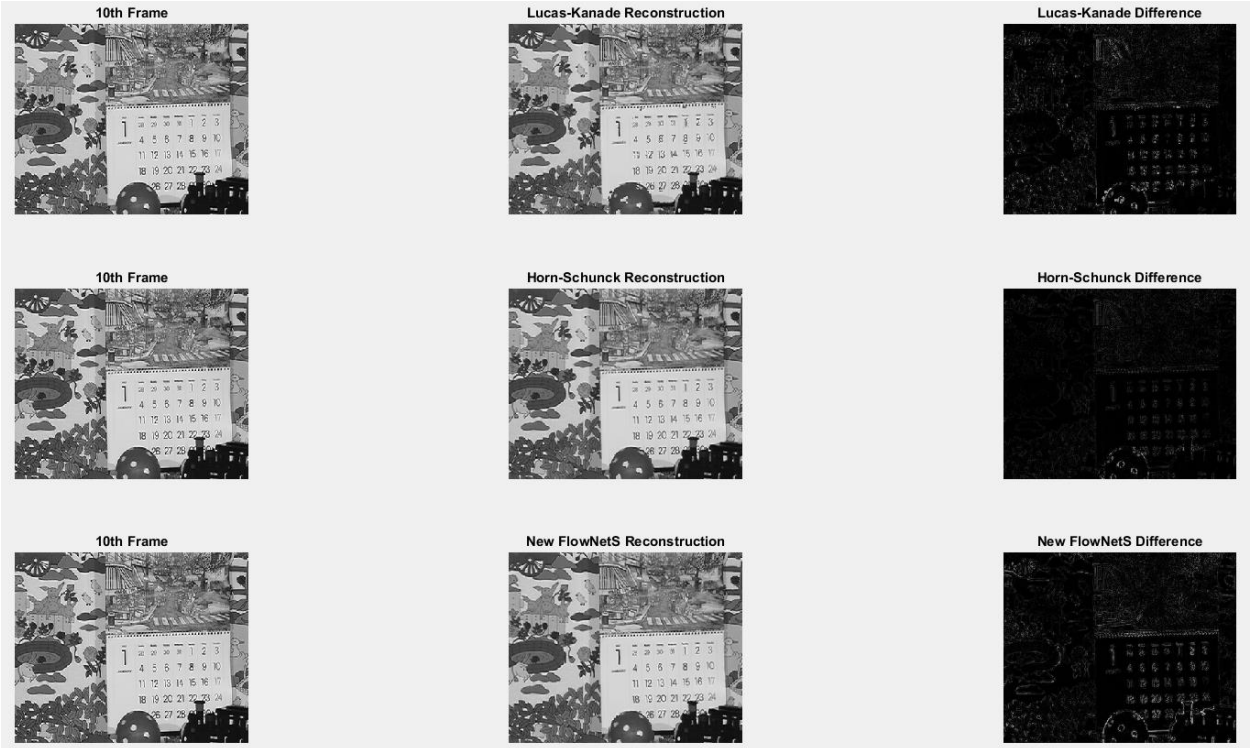


**Figure 2.8.** The results of new FlowNetS versus Lucas-Kanade, Horn-Schunck method

**Reference**

[1] Alexey Dosovitskiy, Philipp Fischer, Eddy Ilg, Philip Hausser, Caner Hazirbas, Vladimir Golkov, Patrick van der Smagt, Daniel Cremers, Thomas Brox, “FlowNet: Learning Optical Flow with Convolutional Networks”, *IEEE International Conference on Computer Vision (ICCV)*, December 2015, Chile