**Project 2 – Code Flow and Meaning**

**1) Synopsis**

At this point we need to rein in all programs written so far, determine how/when they are to be used, what values users may adjust, how data needs to be organized, and in what order users should execute the programs. At this point, many things that feel clear to many come across as vague to others (most specifically me, Jonathan Myers), so this document hopes to serve as an initial step towards finding meaning and purpose in all the files used to generate prediction results. Please read through my understanding for the different applications, their purpose, how they are used, and make comments on these perspectives.

**2) Outline**

This document contains different sections, organized chronologically based on GitHub submissions, each discussing changes that occurred on GitHub. The sections talk about which files were modified, the purpose of those files, and a few notes on what changed. The sections also talk about my understanding of how and when the files are used and how other people may use them in the future.

**3) GitHub History**

As alluded earlier, the subsections here encompass GitHub changes chronologically ordered.

**3.1) added initial version of optical flow & variance - jiahaoxu**

This was the initial submission which included a number of programs:

1. **getData.sh** – This shell script copies all data from gs://uga-dsp/project2, moves into the data folder, then extracts all data from the tar files
2. **OpticalFlow.py** – This python code provides a number of functions (in general following the example at <https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_video/py_lucas_kanade/py_lucas_kanade.html>):
   1. **getIOU**: Calculates a prediction score using trainData zip
   2. **calculateThreshold**: Finds a threshold value in the range [5,130] that provides the best getIOU score
   3. **getHashCodeSet**: Reads all the names from train.txt or test.txt into a set called hashcodeSet
   4. **getTrainData**: The function iterates through all subdirectories of a target data directory, and for each it iterates through every pair of frames – ranging from (0,1) to (98,99). On each pair, it executes calcOpticalFlowFarneback between adjacent frame pairs. The gray results are then evaluated to conclude 2 (cilia) if the gray code >= 128, 1 if the gray code >=32, and 0 otherwise. The results for each pixel are scaled to a 256 range and the results are returned in an array of pairs with the predicted value and the supplied mask value
   5. **generateTestResult**: This function performs the same operations at getTrainData, except here there are no mask values provided for the test set and this function does not return results, rather this function generates png mask files using the values generates by the calcOpticalFlowFarneback and cartToPolar functions. This function takes in a threshold value to determine when it should mark a pixel as cilia or not (see earlier functions for derivation of threshold)
   6. **main**: This function looks to the global variables to determine which actions are appropriate. For ISTRAIN, the program generates training data and then derives the threshold values that provides the best IOU score. Otherwise, it takes the hard coded threshold value. In both cases, the program applies the threshold value to the test entries and generates cilia prediction png files
3. **PixelVariance.py** – Virtually the same as OpticalFlow.py in terms of functions: their purpose, the actions they perform, and the files read and/or generated. This program does not apply functions like calcOpticalFlowFarneback and cartToPolar, rather this function simply takes the raw pixel data from the files, scales their range from 0 to 255, and then derives a threshold value for application to test entries to generate cilia prediction png files.
4. **ToyOpticalFlow.ipynb** and **ToyVariance.ipynb** – These are Jupyter files used for visually testing the operations used in the OpticalFlow.py and PixelVariance.py files.

The PixelVariance.py efficiently performs a basic evaluation of frame values to find a simple threshold and produce predictions. Other than locations for input and output data, this program does not appears to carry any constant values that a user would want to modify. In contrast, the OpticalFlow.py reads and writes the same data types as PixelVariance.py, but OpticalFlow.py uses the calcOpticalFlowFarneback function using fixed values:

flow = cv2.calcOpticalFlowFarneback(preFrame, curFrame, None, 0.5, 3, 15, 3, 5, 1.2, 0)

Here are the function’s parameters (taken from <https://docs.opencv.org/3.4/dc/d6b/group__video__track.html#ga5d10ebbd59fe09c5f650289ec0ece5af>)

|  |  |
| --- | --- |
| **prev** | first 8-bit single-channel input image. |
| **next** | second input image of the same size and the same type as prev. |
| **flow** | computed flow image that has the same size as prev and type CV\_32FC2. |
| **pyr\_scale** | parameter, specifying the image scale (<1) to build pyramids for each image; pyr\_scale=0.5 means a classical pyramid, where each next layer is twice smaller than the previous one. |
| **levels** | number of pyramid layers including the initial image; levels=1 means that no extra layers are created and only the original images are used. |
| **winsize** | averaging window size; larger values increase the algorithm robustness to image noise and give more chances for fast motion detection, but yield more blurred motion field. |
| **iterations** | number of iterations the algorithm does at each pyramid level. |
| **poly\_n** | size of the pixel neighborhood used to find polynomial expansion in each pixel; larger values mean that the image will be approximated with smoother surfaces, yielding more robust algorithm and more blurred motion field, typically poly\_n =5 or 7. |
| **poly\_sigma** | standard deviation of the Gaussian that is used to smooth derivatives used as a basis for the polynomial expansion; for poly\_n=5, you can set poly\_sigma=1.1, for poly\_n=7, a good value would be poly\_sigma=1.5. |
| **flags** | operation flags that can be a combination of the following: |
|  | OPTFLOW\_USE\_INITIAL\_FLOW uses the input flow as an initial flow approximation. |
|  | OPTFLOW\_FARNEBACK\_GAUSSIAN uses the Gaussian winsize×winsize filter instead of a box filter of the same size for optical flow estimation; usually, this option gives z more accurate flow than with a box filter, at the cost of lower speed; normally, winsize for a Gaussian window should be set to a larger value to achieve the same level of robustness. |

These programs have not generated our optimal results, but we should supply this program to users with clear reference to the OpenCV function used along with options to override the constants we used when calling calcOpticalFlowFarneback.

**3.2) Found a pytorch tiramisu programs - jiahaoxu**

This is the PyTorch Tiramisu software loaded to GitHub by Brendan Fortuner (bfortuner) and others (<https://github.com/bfortuner/pytorch_tiramisu>). Of their many pieces, we initially got the models layers.py and tiramisu.py (with no apparent changes) along with the utility training\_utils.py (which appears to be a copy of the source file training.py with replacement of the function view\_sample\_predictions with functions get\_test\_results and get\_test\_results\_cpu). The utility file joint\_transforms.py came from another GitHub project (<https://github.com/ZijunDeng/pytorch-semantic-segmentation>) and it appears a number of ZujunDeng functions were dropped.

Here we discuss what the files submitted do and how they interact:

1. getCilia.py – This file provides a general function and a class with functions:
   1. load\_input – This function reads in the first frame (and mask when applicable) from the png files followed by formatting the image values using reshape and astype(np.uint8). One (or two) resulting lists are returned.
   2. CiliaData – This class stores data specified for Training, Validating, or Testing. The data consists of settings (input\_transform, trarget\_transform, joint\_transform, remove\_cell) along with imgs and masks lists for the values
      1. \_\_init\_\_ - Basis initialization function where the user may spefiy the following:
         1. root – Root directory holding both data and masks as subdirectories
         2. input\_transform –
         3. target\_transform –
         4. joint\_transform –
         5. remove\_cell –
         6. imgs – list of images from load\_input function
         7. masks – list of mask images from load\_input (when applicable)
      2. \_\_getitem\_\_ - returns the image (and mask when appropriate) for the index passed
      3. \_\_len\_\_ - returns the number of images stroed in the imgs list
2. layers.py – This is a copy of the code from bfortuner (see earlier). The classes and functions here appears to be an extension of the torch.nn neural network classes :
   1. DenseLayer – This is an extension of nn.Sequential and execute the forward function on nn.Sequential.
      1. \_\_init\_\_ - This takes in\_channels and growth\_rate to as some arguments for initializing certain modules (which appear needed by nn.Sequential):

self.add\_module('norm', nn.BatchNorm2d(in\_channels))

self.add\_module('relu', nn.ReLU(True))

self.add\_module('conv', nn.Conv2d(in\_channels, growth\_rate,

kernel\_size=3, stride=1, padding=1, bias=True))

self.add\_module('drop', nn.Dropout2d(0.2))

* + 1. forward – simply executes the nn.Sequential forward function
  1. DenseBlock – This is an extension of nn.Module
     1. \_\_init\_\_ - define the nn.Module properties upsample and layers using parameter variables:

self.upsample = upsample

self.layers = nn.ModuleList([DenseLayer(

in\_channels + i\*growth\_rate, growth\_rate)

for i in range(n\_layers)])

* + 1. forward – This function iterates through the layers, calculates the values, and returns what appears to be a matrix with values
  1. TransitionDown – The is an extension of nn.Sequential
     1. \_\_inti\_\_ - This initializes a number of modules used by nn.Sequential using many hardcode values:

self.add\_module('norm', nn.BatchNorm2d(num\_features=in\_channels))

self.add\_module('relu', nn.ReLU(inplace=True))

self.add\_module('conv', nn.Conv2d(in\_channels, in\_channels,

kernel\_size=1, stride=1,

padding=0, bias=True))

self.add\_module('drop', nn.Dropout2d(0.2))

self.add\_module('maxpool', nn.MaxPool2d(2))

* + 1. forward – Simply executes the nn.Sequential forward function
  1. TransitionUp - This is an extension of nn.Module
     1. \_\_init\_\_ - initializes the convTrans object with many hardcode values:

self.convTrans = nn.ConvTranspose2d(

in\_channels=in\_channels, out\_channels=out\_channels,

kernel\_size=3, stride=2, padding=0, bias=True)

* + 1. forward – This performs the convTrans function along with what appears to be cropping a number of steps, it appears to skip forward
  1. Bottleneck – This is an extension of nn.Sequential
     1. \_\_init\_\_ - Allows the user to define most of the variables passed to the super class function:

self.add\_module('bottleneck', DenseBlock(

in\_channels, growth\_rate, n\_layers, upsample=True))

* + 1. forward – Simply executes the nn.Sequential forward function
  1. center\_crop – This is the function called in TransitionUp forward function. This function appears to format the values in a layer using argument values max\_height and max\_width

1. tiramisu.py –
2. joint\_transforms.py –
3. training\_utils.py –

\*\* I had to hold off further analysis of this submit because I could not find a clear main function from which I could derive how this was used. I see there are some Jupyder files from other projects, but they don’t appear applicable for the results generated.

**3.3) Adding the files to untar all tar files and resizing all images in the dataset into same size – a-farahani**

The files in this submission are used to resize the images to match predefined dimensions, in this case that appears to be 256 by 256.