**Sensitivity**

An additional variable: sensitivity, a number in range [0:1], was introduced to the “local thresholding counts” algorithm. This variable sets a lower proportion threshold Amin, where proportion A for a unique pixel is described by the formula:

Here, F and B correspond to the number of times a certain pixel has been assigned to foreground or background after all iterations. Pixels with A>Amin would be assigned to foreground, while the rest would be set to background in the final image segmentation. Setting a higher sensitivity value would reduce the number of falsely assigned foreground pixels, while a lower one would decrease false assignment of background pixels, thus increasing the confidence with which the certain pixel type is assigned. Depending on the nature of input image (predicted proportion of background and foreground), change in sensitivity can both positively and negatively influence the dice score and by changing this variable, the local thresholding algorithm can be fine tuned for specific datasets or single images. In this project, sensitivity value of 0.5 was used, as it results in picture segmentation according to the mode of foreground/background assigment of the pixel.

**Local thresholding discussion**

The main aim of local adaptive thresholding was segmentation of pictures with non-uniform illumination, therefore the efficiency of the algorithm was first and foremost tested on the NIH3T3 dataset, where such problem would often be present. Two separate algorithms were developed, as elaborated in Methods – Local thresholding, namely the “local adaptive thresholding average” and “local adaptive thresholding counts”

While developing the algorithms, there were issues which arose in both of them, due to the nature of the sliding window iterations, as well as unique upsides and downsides emerged for each of the algorithms.

The most prominant challenge in both “local adaptive thresholding average” and “local adaptive thresholding counts” were non-segmented picture edges. The sliding window iterations would always begin at the upper left corner of an image, therefore in outputs one would usually see completely black right and lower edges, which the sliding window algorithm simply could not access, if the size of an image were not a multiple of the chosen stepsize. To put it simply – the sliding window can not slide outside the image. To deal with this issue, reffered to as “the edge problem”,the algorithm was extended, by translating the pixel values onto a larger array than the original picture. In such manner a lower and a right edge was attached with height or width equal to the framesize, set by the user, carrying a NaN value in each position. The sliding window now could iterate over the bounds of the original image array, while still calculating proper threshold values in each frame, as the NaN values could simply be ignored. This method increases the runtime of the algorithm, by adding extra iterations, the number of which depends on the proportion of framesize over stepsize. By rounding the proportion down to the closest integer value, one could directly calculate exactly how many additional frames are added per every iteration row/column, thus the runtime increase can be easily approximated by the user before executing the algorithm. Although this method considers all the pixels in the input image, the intensity assignments for pixels located in the bottom and right is less consistent than for the rest of the image, as the more NaNs contained in an iteration frame, the less values are used for the calculation of the threshold and the confidence decreases for pixels further out.

Note: using matplotlib in the Otsu’s thresholding algorithm would ignore the NaN values automatically, yet due to the “plot issue” elaborated in the discussion for global Otsu’s thresholding, numpy was used and an additional if-loop had to be implemented to manually remove the NaN’s just before thresholding in each iteration.

Other solutions were considered, such as running the algorithm twice, first starting the iterations at the top left pixel and second time at the bottom right pixel, defining a “backwards sliding window”. By uniting the two algorithms the segmented picture would have cleaner edges, especially the bottom and right edge, yet here the upper right and bottom left corners would remain fully black, as those were the overlapping areas for where sliding window wouldnt reach in both algorithms. Such an algorithm also takes almost twice as much time as the previously defined solution, while still containing non-segmented areas. One could define an algorithm, where the sliding window is run 4 times, each time beginning from a different corner, yet such an algorithm would be an even larger increase in runtime and might not be worth implementing, unless the user has a small dataset and wants segmentation as perfect as possible.

A solution, that could also be further implemented, would be “patching up” the non-segmented areas after all sliding window iterations. Here one could compute additional threshold values/ assign pixel values, for additionally defined frames, which were previously not considered. Such method would only add a few up to few dozen (again, based on framesize and stepsize) seconds of runtime and would take into account the same amount of pixels for threshold calculation as for the frames in the sliding window iterations.

As the local thresholding algorithm performs Otsu’s thresholding at each iteration, the runtime of the algorithm directly correlates with the runtime of Otsu’s thresholding itself, as well as number of iterations, which in the simple case (no NaN edges, algorithm is run once, only forwards), would be approximately equal to image shape divided by stepsize, squared. Thus, for optimisation of the algorithm itself, the greatest reduction in runtime followed the optimisation and vectorization of Otsu’s thresholding (see discussion, Otsu’s thresholding), rather than optimisation of the local thresholding algorithm itself. At any case, by setting a stepsize, the user still defines the final runtime of the algorithm themselves, and one has to consider, that depending on how detailed the input image is, the segmentation can take from up to a minute (45 seconds for “mean” algorithm with NaN edges at stepsize = 50 and framesize = 200 on NIH3T3 images) to a few minutes (runtime will also differ based on outer factors, for example the processor of the computer, thus these values are only representative).

An issue that arose in all datasets was random noise in areas with no distinguishable cells, that were bigger than the iteration frame (framesize x framesize), due to random assignment of pixels to foreground or background. In the “counts” method this means, that for each iteration frame containing only background, a predominantly random array of 0’s and 1’s would be generated. For a smaller number of unique pixel foreground/background assignments these random assignments (for example at framesize = 150, stepsize = 50, only 3 frames contribute to each pixel) could easily influence the final segmentation and lead to large areas of random noise. Because the “mean” algorithm only assigns a pixel intensity once the average threshold is calculated, this allowed for a more dynamic segmentation with less or no random noise and led to higher dice score coefficients, therefore the “mean” algorithm was used as the final local thresholding algorithm.

These random noise artifacts were the main influence on the segmentation quality and led to lower dice scores, especially if the algorithms were used on the N2DH-GOWT1 dataset, where generally large areas with no distinguishable foreground (such as picture t39 of the dataset) were present. Due to no apparent non-uniformal illumination and great reduction in segmentation quality in comparison to global thresholding, local thresholding was not further analysed as a segmentation method for this dataset.

For the other two datasets (N2DL-HeLa and NIH3T3) both local thresholding counts and local thresholding means were applied. An algorithm to automatically compare the Dice scores for both segmentations for each dataset was implemented and for both datasets segmentation with and without any kind of preprocessing returned a constant higher Dice score average for the local thresholding means algorithm, which can be explained by the more dynamic nature of the algorithm, as it only performs segmentation once and thus is less prone to improper intensity assignment in the run of the algorithm, reducing, for example, the random noise explained earlier.

For the NIH3T3 dataset the local adaptive segmentation showed a clear increase in the Dice score in comparison to global Otsu thresholding, as one would predict for images with differentiating brightness, the median Dice scores being accordingly 0.817 and 0.672 without preprocessing. Local adaptive segmentation as expected proved to be the optimal segmentation method for this dataset, even though some random noise would still be present in the edges of some segmented images. To increase the segmentation accuracy, one could use a better method that deals with “the edge problem”, as well as set a smaller stepsize to retreive a more accurate average threshold. An issue that this algorithm could not deal with in the NIH3T3 dataset were reflections, which were considered as background in the ground truth images, but can by no means be considered as background in a simple local adaptive thresholding algorithm, as pixels that clearly have a higher intensity than the rest of the image will be considered as foreground in each and every frame they appear in.