بسم الله الرحمن الرحيم

وزارة التعليم العالي

الجامعة الافتراضية السوريّة

ماجستير التأهيل والتخصص في المعلوماتية الحيوية

الجامعة الافتراضيّة السوريّة

SYRIAN VIRTUAL UNIVERSITY

عدُّ الخلايا الدمويّة المؤتمت باستعمال شبكة عصبونيّة مبنيّة على شبكة UNet مستحدثة وباستعمال حجرات عدّادة من نوع نيوباور

تحت إشراف الدكتور حيّان حسن مشكوراً

تقدمة الطالب : حيدر عمر عبد الدائم

Automated Blood Cell Count using a Neural Network based on a Novel U-Net Architecture and Neubauer Haemocytometer

Author: Haider Abd Aldaim, Syrian Virtual University, Damascus, Syria Email: haider_176932@mail.svuonline.org

Abstract: Large and expensive analytical machines pose an insurmountable hurdle to providing healthcare in remote suburban and rural areas. Such machines have proven to be cost prohibitive, immobile and locked down to specific vendors or countries. With advancement in machine learning algorithms and improved compute units a new chimeric option between manual slow and inaccurate analysis and large cost prohibitive immobile analysis machines. This option can prove to be a solution with minimal disadvantages. The objective of this paper is to develop a machine learning algorithm/neural network capable of carrying out blood cells count using a single image from a blood sample on a Neubauer counting chamber/slide. Developing such algorithms and models could pave the path for a universal and readily available on-the-fly blood tests in rural and suburban areas. Methods: a methodolgy based on heavy mathematical preprocessing of the images followed by UNet have been developed as a special architecture for the segmentation of medical images offering less computational load shifting the bulk of the processing onto mathematical models reducing the overall computational cost of the process. Results: the trained model has shown remarkable accuracy when it comes to segmentation (97.59%). However instance segmentations performance of stacked cells is yet insufficient for medical use.

Introduction:

Manual complete blood count have proven to be a tiresome error-prone task in the modern world. While automated flow cytometers exist, they tend to be large, complicated to operate and expensive. With the development of AI in the past couple of years, the idea of a chimeric blood count system combining cheap readily available counting chambers and microscopes with machine learning and vision to create a cheap and reliable system have never been more approachable. In this paper we explore the development of a machine vision model to tackle such a task.

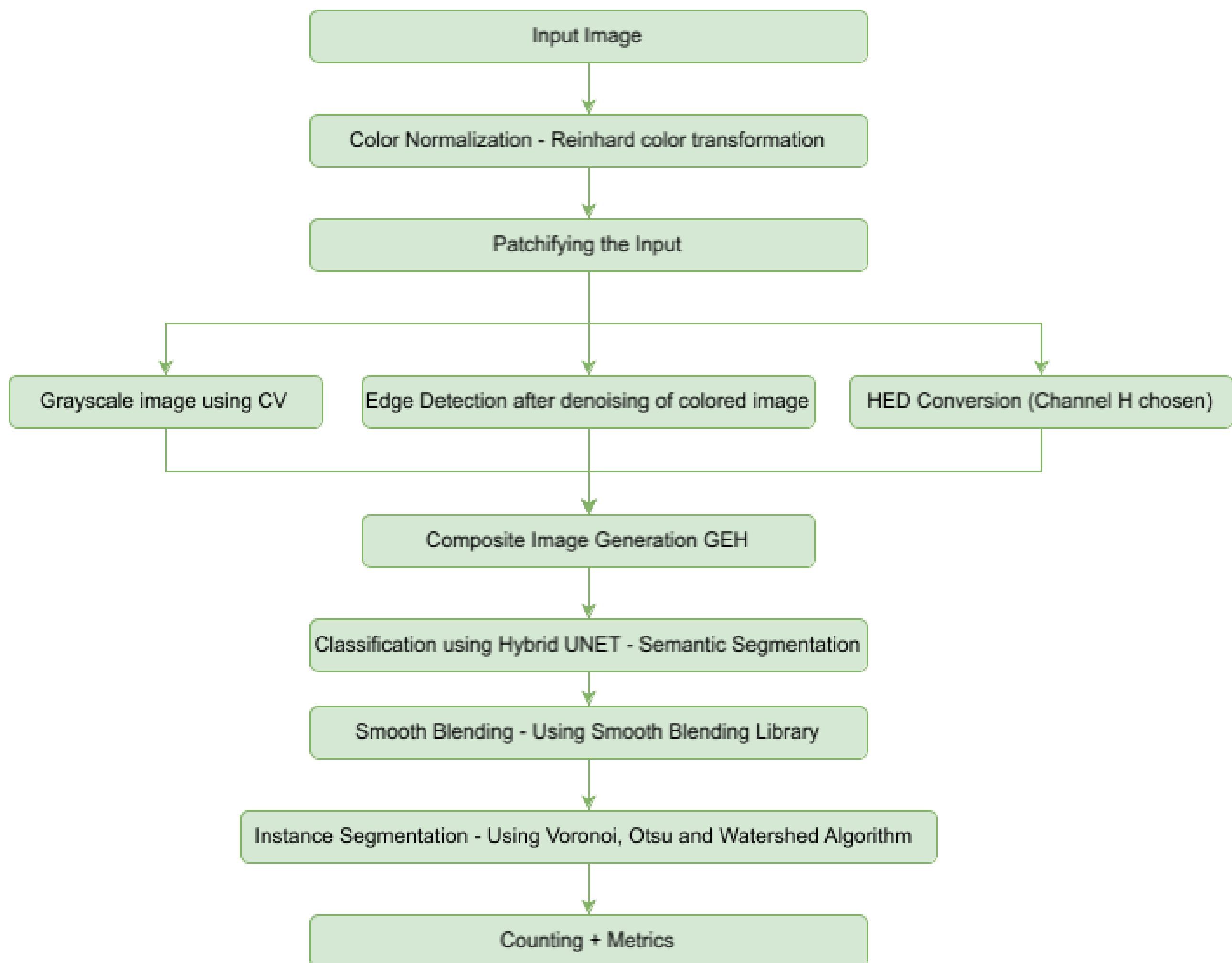

Figure 1: Project's Workflow

Materials and Methods:

After sifting through the literature for suitable architectures three models have been identified Meta's SAM 2, YOLOv11 and UNet. Meta's SAM 2 i.e Segment Anything Model is a pretrained open source (Apache 2.0) model that requires only tweaking to better fit a customized task, However, as we plan to use our model in rural areas, providing a suffiecnt processing power to use this model have proven to be cumbersome and thus we chose not to use the aforementioned model. YOLOv11 i.e You Only Look Once model is quick, lightweight and open source model with the ability to run on limited resources, however, further analysis of the model has shown that the model is mainly designed to identify real life day to day objects and is illsuited for medical images. UNet architecture is an open source architecture/model devised specially for the spacial segmentation of medical images. It is also

lightweight and relatively simple to implement. Multiple versions of the model exists and for our purposes we will use a tweaked version that we create.
Noise reduction/thresholding was intentionally ignored with hope of reducing overfitting as real life data has innate noise. The images go through the following steps, firstly, the colorspace of the input image will be transfered into a common colorspace using Reinhard transformation, this transformation helps minimize errors in the segmentation due to color discrepancies. Secondly, the images are used to generate three pseudodimensions (G = Grayscale,E = Edges, H= Haematoxylin) which are stacked together, the images will then be "patchified" into easily consumable chunks for the neural network to process, these chunks will be 256x256 pixels. The training dataset consisting of 1328 images was deemed sufficient and no data augmentation methodoligies were used. Image segmentation using UNet follows generating a mask of all BCs. The segmented patches will then be smooth blended back into a complete image and a combination of Voronoi, Otsu and Watershed algorithm will be used to segment the mask into instances. Finally, a count is run on the segmented image and the results of the analysis will be displayed in human readable format.
We will discuss the methodologies used in depth in the following subchapters.

Reinhard transformation:

Reinhard transformation is a popular method for color normalization in medical imaging, particularly for standardizing images across different scanners or devices. However, there are several other techniques used for similar purposes were considered such as : Histogram Matching, Linear Transformations, Z-Score Normalization, Contrast-Limited Adaptive Histogram Equalization (CLAHE). At the end Reinhard transformation was chosen due to its suitibility for microscopic images and relatively low computational cost.
How it works :
The Reinhard transformation is a global mapping technique that adjusts the pixel values of an input image so that its color distribution and intensity resemble a reference image (usually a "target" or a standard reference image). The transformation involves two key steps: intensity normalization and chromaticity scaling. Step 1: Convert to a Logarithmic Color Space, specifically by taking the logarithm of the intensity values of each pixel. This helps to reduce the dynamic range and handle very bright or dark regions in a controlled way. Step 2: Compute Statistics of the Image. After converting the image to a logarithmic space, statistics (such as the mean and standard deviation) of both the input image and the reference image are computed. This allows for a direct comparison of their intensity

Model: "functional"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 256, 256, 3)	0	-
conv2d (Conv2D)	(None, 256, 256, 64)	1,792	input_layer[0][0]
batch_normalization (BatchNormalizatio…	(None, 256, 256, 64)	256	conv2d[0][0]
re_lu (ReLU)	(None, 256, 256, 64)	0	batch_normalizat…
conv2d_1 (Conv2D)	(None, 256, 256, 64)	36,928	re_lu[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 256, 256, 64)	256	conv2d_1[0][0]
re_lu_1 (ReLU)	(None, 256, 256, 64)	0	batch_normalizat…
max_pooling2d (MaxPooling2D)	(None, 128, 128, 64)	0	re_lu_1[0][0]
conv2d_2 (Conv2D)	(None, 128, 128, 128)	73,856	max_pooling2d[0]…
batch_normalizatio… (BatchNormalizatio…	(None, 128, 128, 128)	512	conv2d_2[0][0]
re_lu_2 (ReLU)	(None, 128, 128, 128)	0	batch_normalizat…
conv2d_3 (Conv2D)	(None, 128, 128, 128)	147,584	re_lu_2[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 128, 128, 128)	512	conv2d_3[0][0]
re_lu_3 (ReLU)	(None, 128, 128, 128)	0	batch_normalizat…
max_pooling2d_1 (MaxPooling2D)	(None, 64, 64, 128)	0	re_lu_3[0][0]
conv2d_4 (Conv2D)	(None, 64, 64, 256)	295,168	max_pooling2d_1[…
batch_normalizatio… (BatchNormalizatio…	(None, 64, 64, 256)	1,024	conv2d_4[0][0]
re_lu_4 (ReLU)	(None, 64, 64, 256)	0	batch_normalizat…
conv2d_5 (Conv2D)	(None, 64, 64, 256)	590,080	re_lu_4[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 64, 64, 256)	1,024	conv2d_5[0][0]
re_lu_5 (ReLU)	(None, 64, 64, 256)	0	batch_normalizat…
max_pooling2d_2 (MaxPooling2D)	(None, 32, 32, 256)	0	re_lu_5[0][0]
conv2d_6 (Conv2D)	(None, 32, 32, 512)	1,180,160	max_pooling2d_2[…
batch_normalizatio… (BatchNormalizatio…	(None, 32, 32, 512)	2,048	conv2d_6[0][0]
re_lu_6 (ReLU)	(None, 32, 32, 512)	0	batch_normalizat…

Figure 2: Model's Summary part 1

conv2d_7 (Conv2D)	(None, 32, 32, 512)	2,359,808	re_lu_6[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 32, 32, 512)	2,048	conv2d_7[0][0]
re_lu_7 (ReLU)	(None, 32, 32, 512)	0	batch_normalizat…
max_pooling2d_3 (MaxPooling2D)	(None, 16, 16, 512)	0	re_lu_7[0][0]
conv2d_8 (Conv2D)	(None, 16, 16, 1024)	4,719,616	max_pooling2d_3[…
batch_normalizatio… (BatchNormalizatio…	(None, 16, 16, 1024)	4,096	conv2d_8[0][0]
re_lu_8 (ReLU)	(None, 16, 16, 1024)	0	batch_normalizat…
conv2d_9 (Conv2D)	(None, 16, 16, 1024)	9,438,208	re_lu_8[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 16, 16, 1024)	4,096	conv2d_9[0][0]
re_lu_9 (ReLU)	(None, 16, 16, 1024)	0	batch_normalizat…
conv2d_transpose (Conv2DTranspose)	(None, 32, 32, 512)	2,097,664	re_lu_9[0][0]
concatenate (Concatenate)	(None, 32, 32, 1024)	0	conv2d_transpose… re_lu_7[0][0]
conv2d_10 (Conv2D)	(None, 32, 32, 512)	4,719,104	concatenate[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 32, 32, 512)	2,048	conv2d_10[0][0]
re_lu_10 (ReLU)	(None, 32, 32, 512)	0	batch_normalizat…
conv2d_11 (Conv2D)	(None, 32, 32, 512)	2,359,808	re_lu_10[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 32, 32, 512)	2,048	conv2d_11[0][0]
re_lu_11 (ReLU)	(None, 32, 32, 512)	0	batch_normalizat…
conv2d_transpose_1 (Conv2DTranspose)	(None, 64, 64, 256)	524,544	re_lu_11[0][0]
concatenate_1 (Concatenate)	(None, 64, 64, 512)	0	conv2d_transpose… re_lu_5[0][0]
conv2d_12 (Conv2D)	(None, 64, 64, 256)	1,179,904	concatenate_1[0]…
batch_normalizatio… (BatchNormalizatio…	(None, 64, 64, 256)	1,024	conv2d_12[0][0]
re_lu_12 (ReLU)	(None, 64, 64, 256)	0	batch_normalizat…
conv2d_13 (Conv2D)	(None, 64, 64, 256)	590,080	re_lu_12[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 64, 64, 256)	1,024	conv2d_13[0][0]

Figure 3: Model's Summary part 2

re_lu_13 (ReLU)	(None, 64, 64, 256)	0	batch_normalizat…
conv2d_transpose_2 (Conv2DTranspose)	(None, 128, 128, 128)	131,200	re_lu_13[0][0]
concatenate_2 (Concatenate)	(None, 128, 128, 256)	0	conv2d_transpose… re_lu_3[0][0]
conv2d_14 (Conv2D)	(None, 128, 128, 128)	295,040	concatenate_2[0]…
batch_normalizatio… (BatchNormalizatio…	(None, 128, 128, 128)	512	conv2d_14[0][0]
re_lu_14 (ReLU)	(None, 128, 128, 128)	0	batch_normalizat…
conv2d_15 (Conv2D)	(None, 128, 128, 128)	147,584	re_lu_14[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 128, 128, 128)	512	conv2d_15[0][0]
re_lu_15 (ReLU)	(None, 128, 128, 128)	0	batch_normalizat…
conv2d_transpose_3 (Conv2DTranspose)	(None, 256, 256, 64)	32,832	re_lu_15[0][0]
concatenate_3 (Concatenate)	(None, 256, 256, 128)	0	conv2d_transpose… re_lu_1[0][0]
conv2d_16 (Conv2D)	(None, 256, 256, 64)	73,792	concatenate_3[0]…
batch_normalizatio… (BatchNormalizatio…	(None, 256, 256, 64)	256	conv2d_16[0][0]
re_lu_16 (ReLU)	(None, 256, 256, 64)	0	batch_normalizat…
conv2d_17 (Conv2D)	(None, 256, 256, 64)	36,928	re_lu_16[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 256, 256, 64)	256	conv2d_17[0][0]
re_lu_17 (ReLU)	(None, 256, 256, 64)	0	batch_normalizat…
conv2d_18 (Conv2D)	(None, 256, 256, 2)	130	re_lu_17[0][0]

Total params: 62,098,950 (236.89 MB)

Trainable params: 31,043,586 (118.42 MB)

Non-trainable params: 11,776 (46.00 KB)

Optimizer params: 31,043,588 (118.42 MB)

Figure 4: Model's Summary part 3

distributions. Moreover, each color channel (RGB, typically), the mean and standard deviation of pixel intensities are computed for both the input image and the reference image. Step 3: Match the Intensity Distribution, the next step involves adjusting the pixel values of the input image so that the mean and standard deviation of its intensity distribution match those of the reference image. This is done by a scaling and shifting operation, which is a linear transformation based on the computed statistics. Step 4: Chromatic Adjustment (Color Correction), after intensity normalization, the next step is to handle the chromaticity (color balance) of the image. The Reinhard transformation attempts to preserve the color relationships between pixels while matching the overall intensity distribution. This is done by scaling the chromatic channels (e.g., the RGB color channels) to match the color distribution in the reference image. The goal is to make the color balance in the transformed image similar to that of the reference image, avoiding any shifts in hue or saturation. This is typically achieved by scaling the color channels independently while maintaining their relative proportions, ensuring that the image doesn't look unnaturally desaturated or color-shifted after the transformation. Step 5: Apply Inverse Logarithmic Transform (if necessary). After the normalization process, the image may be transformed back into the linear space (inverse logarithmic transformation), depending on the application. This step can help bring the image back to a more realistic brightness level, especially if the image was originally captured in a high dynamic range (HDR) format.

Due to the dataset having over a thousand images no one image was chosen as a reference image but rather the standard deviation and mean of all images was calculated and averaged and used a reference.

Means (BGR) = (195.11752, 183.79773, 183.8279)

Standard deviation (BGR) = (40.50984, 58.086155, 55.982433)

Furthermore, there was no need for inversing the logarithmic transform in this case.

Channel's generation: Grayscale pseudochannel

The first pseudochannel to be generated is the grayscale layer. This layer uses a simple transformation from the BGR colorspace to HSV colorspace. The V channel was simply isolated as the grayscale layer.

Channel's generation: Edges pseudochannel

The second pseudochannel was more complicated to generate. We attempted to use built-in edge detection algorithms (Canny) in OpenCV however to no avail. The use of a simple Sobel filter with the dimensions of 3x3 on both axes after applying a

fastNlMeansDenoising filter followed by GaussianBlur filter proved to be sufficient to generate a fairly accurate mask of the images edges.
Channel's generation: Haematoxylin pseudochannel
Scikit library was used for this job. Converting the image from OpenCV's BGR to RGB followed by changing the scale from 0 - 255 to 0 - 1. And then the function rgb2hed is called to convert RGB to HED colorspace. Finally, the H channel was chosen followed by the conversion back to 0 - 255.

"Patchification":

A special function was written with purpose of turning incoming images into patches of 256x256 size which the UNet network can easily process. At first the image dimensions are read, followed by a loop sliding a frame across the images generating the wanted patches. A stride of 240 was chosen creating an overlap of 16 in both axes.

Data augmentation:

As forementioned, data augmentation was not used as the dataset was of sufficient size.
The UNet Network:
A Unet Network with 4 encoder blocks and 4 decoder blocks was employed resulting in a neural network with the shape (see pages 3-5) and parameters (see page 5). The nerual network was trained on 100 images due to insufficient ram generating a model of the size 237MB after 10 epochs with an accuracy of 97.59%.

Smooth Blending:

To simplify the process of stitching the patches back together smooth_tiled_predictions.py library was used [12]. This library is developed to sticht GIS images together, however it has shown remarkable performance avoiding the "edge effect" (i.e the tile edges being misclassified due to stitching aberrations). Nevertheless, the library required some modification to function as it used deprecated functions.

Instance Segmentation: Voronoi:

Voronoi image segmentation is a technique that partitions an image into regions based on distance to a set of seed points. It involves creating a Voronoi diagram, which divides the image into polygons, each associated with a seed point. Pixels within a polygon are closer to their associated seed point than to any other. The seed points can be manually selected or determined automatically using various methods

(Brightness in our case). Voronoi segmentation is useful for separating objects with distinct boundaries and can be applied to various image types, including medical, satellite, and microscopy images. It can be combined with other segmentation techniques, such as thresholding or edge detection, to improve accuracy and robustness. Voronoi segmentation has applications in various fields, including medical image analysis, object tracking, and pattern recognition.

Instance Segmentation: Otsu:

Otsu image segmentation is a popular technique for thresholding images. It automatically selects a threshold value to separate pixels into two classes: foreground and background. The method analyzes the image's histogram, which represents the distribution of pixel intensities. It aims to find the threshold that maximizes the variance between the two classes while minimizing the variance within each class. Otsu's method is particularly effective for images with bimodal histograms, where the pixel intensities are clustered around two distinct values. It is widely used in various applications, including medical image analysis, object detection, and document analysis.

Instance Segmentation: Watershed:

The watershed algorithm is a popular technique used for image segmentation, particularly when dealing with complex images where simple thresholding and contour detection may not yield accurate results. It treats the image as a topographic surface, where pixel intensities represent elevation. The algorithm identifies catchment basins, which are regions that drain into a common local minimum, and divides the image along the boundaries of these basins. This process is analogous to how water flows downhill and collects in valleys, with the boundaries between watersheds representing the dividing lines. The watershed algorithm is effective in segmenting objects with irregular shapes and can be particularly useful when dealing with images containing touching or overlapping objects. However, it is important to note that the watershed algorithm can be sensitive to noise and may oversegment the image, leading to the creation of many small regions. To mitigate this issue, various techniques, such as marker-based watershed segmentation, can be employed to guide the algorithm and improve the quality of the segmentation results.

Instance Segmentation: pyclEsperanto Library:

pyclEsperanto Library offers a convenient method to implement all the forementioned methodologies by calling abstracted functions. "voronoi_otsu_labeling" function was used which uses all the abovementioned

algorithms to yield an instance segmented image with the highest "class" number denoting the number of instances found in the image, i.e BCs count.

Results:

The UNet model has shown remarkable accuracy during training (see figure 5). However, the model was trained for only 10 epochs due to processing power limitations and thus, even better results can be expected when more processing power is available. When it comes to instance segmentation performance the model has shown subpar performance in images with many overlapping cells but well enough with images that are not so "crowded". (see figures 7-9 for results).

```
Fit model on training data
Epoch 1/10
225/225 ━━━━━━━━━━━━━━━━━━━━ 2242s 10s/step - accuracy: 0.9313 - loss: 0.1908 - val_accuracy: 0.9424 - val_loss: 0.1697
Epoch 2/10
225/225 ━━━━━━━━━━━━━━━━━━━━ 2351s 10s/step - accuracy: 0.9612 - loss: 0.1099 - val_accuracy: 0.9644 - val_loss: 0.1100
Epoch 3/10
225/225 ━━━━━━━━━━━━━━━━━━━━ 1962s 9s/step - accuracy: 0.9656 - loss: 0.0976 - val_accuracy: 0.9574 - val_loss: 0.1224
Epoch 4/10
225/225 ━━━━━━━━━━━━━━━━━━━━ 1955s 9s/step - accuracy: 0.9665 - loss: 0.0941 - val_accuracy: 0.9639 - val_loss: 0.1208
Epoch 5/10
225/225 ━━━━━━━━━━━━━━━━━━━━ 1955s 9s/step - accuracy: 0.9719 - loss: 0.0765 - val_accuracy: 0.9581 - val_loss: 0.1152
Epoch 6/10
225/225 ━━━━━━━━━━━━━━━━━━━━ 1958s 9s/step - accuracy: 0.9729 - loss: 0.0745 - val_accuracy: 0.9672 - val_loss: 0.1084
Epoch 7/10
225/225 ━━━━━━━━━━━━━━━━━━━━ 1957s 9s/step - accuracy: 0.9740 - loss: 0.0693 - val_accuracy: 0.9673 - val_loss: 0.0938
Epoch 8/10
225/225 ━━━━━━━━━━━━━━━━━━━━ 1959s 9s/step - accuracy: 0.9737 - loss: 0.0700 - val_accuracy: 0.9709 - val_loss: 0.0895
Epoch 9/10
225/225 ━━━━━━━━━━━━━━━━━━━━ 1957s 9s/step - accuracy: 0.9736 - loss: 0.0687 - val_accuracy: 0.9636 - val_loss: 0.1072
Epoch 10/10
225/225 ━━━━━━━━━━━━━━━━━━━━ 1958s 9s/step - accuracy: 0.9759 - loss: 0.0632 - val_accuracy: 0.9654 - val_loss: 0.1030
```

Figure 5: Training's output

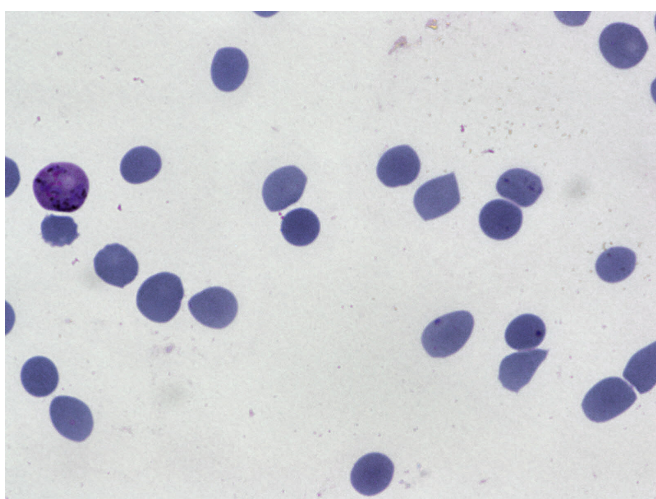

Figure 6: Input image

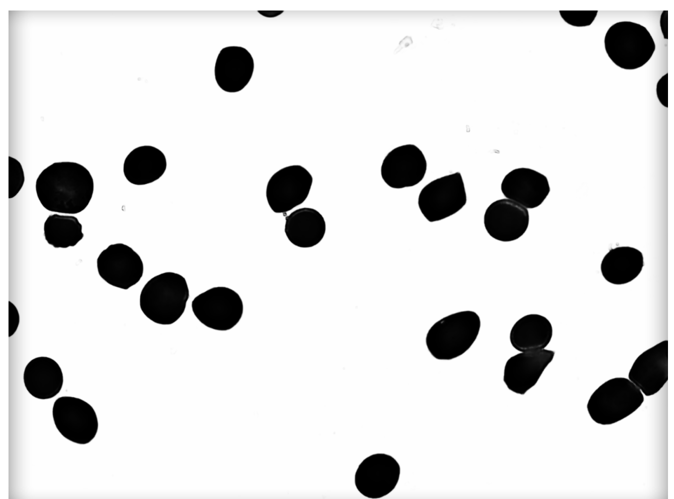

Figure 8: UNet model's output

Figure 9: Instance segmented output

```
The number of RBCs in the image is : 29
```

Discussion:

In this section we shall discuss each step of this project with its limitations and possible improvements:

Reinhard Transformation:

Before processing the incoming images finding a common colorspace was deemed necessary as most microscopic images were taken in different lighting conditions with various lightsources. Plus, the cameras used to take such images are usually not configured in the same way when it comes to exposure, color balance, etc. Reinhard transformation was chosen for being simple, easy to implement, and not resource intensive. However, it has many limitations including the fact it might not perform as well in situations where significant local variations in lighting or scanner settings exist (e.g., complex multi-modal images), as well as, as a global transformation, it may not preserve local details or handle noise well in areas with highly variable intensities, moreover, the method assumes that the reference image represents the ideal or "target" distribution, which may not always be the case in real-world medical datasets. Thus, other implementations of color equalisation such as CLAHE may prove to be more profitable for the prediction process with its in-built noise reduction at a relatively low cost increase. Future improvements may include: using a dataset that is homogenically stained, with pictures taken using the same camera. As well as, choosing a color equalisation method that accounts well for the local variations of intensity in an image.

Generating the composite pseudolayers G,E,H:

The idea to generate such composite came to be through careful observation of the human's way of regarding an image. Human vision tends to ignore most of the color input in the early stages of understanding an image, evident by the nearly instance identification of subjects in an image regardless of whether the image is colored or not. Color data helps only adding "final touchs" of information to the image being observed. A model that processes the input image as grayscale with later use of image color info as last layer was devised but never came to fruition due to the complexity of the implementation. Thus, we ended up using a composite input consisting of the grayscale image, the edges detected using Sobel in an attempt to reduce the overall processing the needed to be done with the neural network, and finaly the H layer from the HED color space showing cells that are stained with haemotoxylin. This method has shown extremely promising results with the composite clearly highlighting the cells to the human eye. Improving the edge detectors and the color separation in future implementations maybe the way forward.

Image Patchification:

There isn't much to discuss here as the method used did what is supposed to do. Patches of 256x256 were generated which seems to be a nice middleground between the much smaller model with 128x128 and the exponentially bigger model with 512x512 inputs.

The UNet model:

The UNet model was one of the largest hurdles developing this project due to its complexity and resource intesivity. Other more complex UNet models were considered at beginning such as UNet+, UNet++, and UNet3+. However, due to the complexity of the implementation of such models they deprecated in favor of the simple to implement UNet model(for example the UNet3+ model uses a fully connected implementation between each and every layer of the encoders and decoders!) . The model is ofcourse old and newer models should in theory provide better accuracy. Nevertheless, the accuracy of the model was impressive and such future experimentation with such models may prove to be interesting however unnecessary.

Smooth blending:

A library from github was used to implement the blending. This library was developed to merge satelite images for GIS analysis. Although the library performed well the edges had some aberrations which should not be there and thus a future

improvement could be either a rewrite of the whole library or from ground up implementation of a smooth blending library that is specialised in microscopic images.

Instance Segmentation:

Instance segmentation took place using a library called pyclEsperanto. This library uses a combination of Voronoi algorithm with Otsu followed by Watershed method. While initial tests were promising the library's performance was insufficient to isolate cells in highly crowded images. Possible improvments in this regard include either finding new algorithms to isolate the cells or developing an instance segmentation neural network that can be plugged in after th UNet to perform the instance segmentation. Another possibilty is to tweak the UNet network to provide masks with isolated cells which can be easily count.

Metrics and cell counting:

Lastly, the topic of metrics and cell counting to be discussed. After applying instance segmentation the issue of counting cells is relatively trivial. We used the numpy.max function to denote the cells count as it provides the last "class" number which is the number of classes. However, in later iterations it is possible to do calculations such as RBCs mean diameter, volume and much more. Moreover, WBCs can be classified not only into their major groups such as neutrophils and monocytes but also divided into groups according to their maturity. But, due to time and resource limitations and the insufficient accuracy of the results we did not implement such functions.

Acknowledgement:

To my family, friends and valued teachers in my university and online, without which this project would not be possible.

Citations:

Dataset:
[1] https://github.com/Deponker/Blood-cell-segmentation-dataset
Reinhard transformation:
[2] https://www.kaggle.com/code/charansai612/color-transformation-reinhard
[3] https://eurasip.org/Proceedings/Eusipco/Eusipco2021/pdfs/0001231.pdf
Grayscale layer:

[4]
https://docs.opencv.org/4.7.0/d8/d01/group__imgproc__color__conversions.html#gga4e0972be5de079fed4e3a10e24ef5ef0aa4f6bc658bc546e1660fcab6bf7858f4
H Layer:
[5]
https://scikit-image.org/docs/stable/api/skimage.color.html#skimage.color.rgb2hed
Edge Layer:
[6] https://docs.opencv.org/4.x/d2/d2c/tutorial_sobel_derivatives.html
[7] https://imagejdocu.list.lu/gui/process/find_edges
UNet:
[8] https://www.digitalocean.com/community/tutorials/unet-architecture-image-segmentation
[9] https://keras.io/api/layers/normalization_layers/batch_normalization/
[10] https://www.tensorflow.org/api_docs/python/tf/keras/Model
[11] https://medium.com/@mlquest0/unet-3-fully-explained-next-generation-unet-2a8e204e4cf9
Smooth blending:
[12]
https://github.com/bnsreenu/python_for_microscopists/tree/master/229_smooth_predictions_by_blending_patches
Vonoroi, Otsu & Watershed:
[13] https://en.wikipedia.org/wiki/Otsu%27s_method
[14] https://github.com/clEsperanto/pyclesperanto_prototype/blob/master/demo/segmentation/voronoi_otsu_labeling.ipynb
Additional resources:
[15] Digital Sreeni youtube channel with hunderds of videos about machine learning and image analysis : https://www.youtube.com/@DigitalSreeni

Appendix 1 – Code Blocks

```
# Imports
import cv2
import numpy as np
import os
from skimage.color import rgb2hed
import math
# Neural Network Imports
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Activation, ReLU
from tensorflow.keras.layers import BatchNormalization, Conv2DTranspose, Concatenate
from tensorflow.keras.models import Model, Sequential, load_model
#from tensorflow.keras.utils import plot_model
# Smooth Blending Library Import
from smooth_tiled_predictions import predict_img_with_smooth_windowing
# Segmentation Library
import pyclesperanto_prototype as cle
```

Codeblock 1: Imports

```
#Load Images Function

def load_dataset(path1, path2):
    # Read the images folder like a list
    image_dataset = os.listdir(path1)
    mask_dataset = os.listdir(path2)

    # Make a list for images and masks filenames
    orig_img = []
    mask_img = []
    #Temporary limit
    x=0
    for file in image_dataset:
        orig_img.append(cv2.imread(path1+file))
        for mask in mask_dataset:
            if mask.split(".")[0] == file.split(".")[0]:
                mask_img.append(cv2.imread(path2+mask))
                break
        #Temp
        x=x+1
        if x == 100:
            break

    return orig_img, mask_img
```

Codeblock 2: Loading dataset into the RAM

```
# Reinfeld Tranfsormation Function
def reinhard_transformation(input_image, reference_image = None, reference_mean = None, reference_std = None):
    # Convert images to float32 for accurate scaling
    input_image = input_image.astype(np.float32)
    if reference_image != None:
        reference_image = reference_image.astype(np.float32)

    # Compute the mean and standard deviation for each channel in input and reference images
    input_mean = np.mean(input_image, axis=(0, 1), keepdims=True)
    input_std = np.std(input_image, axis=(0, 1), keepdims=True)

    if reference_mean == None and reference_image != None:
        reference_mean = np.mean(reference_image, axis=(0, 1), keepdims=True)
    if reference_std == None and reference_image != None:
        reference_std = np.std(reference_image, axis=(0, 1), keepdims=True)

    # Do calculation if there is either a reference image or standard deviation and mean values
    if (reference_image != None) or (reference_mean != None and reference_std != None):
        # Normalize the input image by adjusting the mean and standard deviation
        normalized_input = (input_image - input_mean) / input_std
        transformed_image = normalized_input * reference_std + reference_mean

        # Clip the values to the valid range [0, 255]
        transformed_image = np.clip(transformed_image, 0, 255)

        # Convert back to uint8
        transformed_image = transformed_image.astype(np.uint8)

        return transformed_image
    # Return Mean and Standard deviation of input image if no input image or mean and standard deviation values are found
    if (reference_image == None) and (reference_mean == None and reference_std == None):
        return input_mean,input_std
```

Codeblock 3: Reinhard Transformation

```
def patchify(image, patch_size, stride=1):
    # Get the dimensions of the input image
    image_height, image_width = image.shape[:2]

    # Get patch dimensions
    patch_height, patch_width = patch_size

    # Calculate number of patches that fit the image size
    patches_height = (image_height - patch_height) // stride + 1
    patches_width = (image_width - patch_width) // stride + 1

    # Initialize an array to store the patches
    patches = []

    # Iterate over the image to extract patches
    for i in range(0, patches_height * stride, stride):
        for j in range(0, patches_width * stride, stride):
            patch = image[i:i+patch_height, j:j+patch_width]
            patches.append(patch)

    # Convert the list of patches to a NumPy array
    patches = np.array(patches)

    # Reshape to (num_patches, patch_height, patch_width, channels) if the image is color
    if len(image.shape) == 3:
        patches = patches.reshape(-1, patch_height, patch_width, image.shape[2])
    else:
        patches = patches.reshape(-1, patch_height, patch_width)

    return patches
```

Codeblock 4: Patchification function

```
#OpenCV Grayscale Conversion Wrapper
def generateGrayscale(img):
    grey_img= cv2.cvtColor(img, cv2.COLOR_BGR2HSV_FULL)[...,2]
    return grey_img
```

Codeblock 5: A function to return the grayscale layer

```
#HED Conversion returning channel 0
def generateHEDImage(img):
    rgbimg = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)/255
    hedimg = rgb2hed(rgbimg)
    return (hedimg[...,0]*255.999).astype(np.uint8)
```

Codeblock 6: A function to return the haemotoxylin layer

```
#Edge Detection using Sobel
def generateEdgeImageSobel(img):
    scale = 1
    delta = 0
    ddepth = cv2.CV_16S
    #Denoising
    denoised_img = cv2.fastNlMeansDenoising(img,None,7,7,21)
    src = denoised_img
    # Apply gaussian blur
    src = cv2.GaussianBlur(src, (3, 3), 0)
    #Convert to grayscale
    gray = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)
    #Apply sobel filter
    grad_x = cv2.Sobel(gray, ddepth, 1, 0, ksize=3, scale=scale, delta=delta, borderType=cv2.BORDER_DEFAULT)
    #Apply sobel filter Gradient-Y
    grad_y = cv2.Sobel(gray, ddepth, 0, 1, ksize=3, scale=scale, delta=delta, borderType=cv2.BORDER_DEFAULT)

    abs_grad_x = cv2.convertScaleAbs(grad_x)
    abs_grad_y = cv2.convertScaleAbs(grad_y)

    grad = cv2.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0)
    return grad
```

Codeblock 7: A function to generate edge layer

```
# Stack the 3 Generated Layers into PseudoRGB Image
def generateGEHImageStack(img):
    img_G = generateGrayscale(img)
    img_E = generateEdgeImageSobel(img)
    img_H = generateHEDImage(img)
    return np.dstack((img_G,img_E,img_H))
```

Codeblock 8: A function to generate a composite image using all of the previous layers

```python
### U-Net Definition
# Convolutional block
def convolution_operation(entered_input, filters=64):
    # Taking first input and implementing the first conv block
    conv1 = Conv2D(filters, kernel_size = (3,3), padding = "same")(entered_input)
    batch_norm1 = BatchNormalization()(conv1)
    act1 = ReLU()(batch_norm1)

    # Taking first input and implementing the second conv block
    conv2 = Conv2D(filters, kernel_size = (3,3), padding = "same")(act1)
    batch_norm2 = BatchNormalization()(conv2)
    act2 = ReLU()(batch_norm2)

    return act2

def encoder(entered_input, filters=64):
    # Collect the start and end of each sub-block for normal pass and skip connections
    enc1 = convolution_operation(entered_input, filters)
    MaxPool1 = MaxPooling2D(strides = (2,2))(enc1)
    return enc1, MaxPool1

def decoder(entered_input, skip, filters=64):
    # Upsampling and concatenating the essential features
    Upsample = Conv2DTranspose(filters, (2, 2), strides=2, padding="same")(entered_input)
    Connect_Skip = Concatenate()([Upsample, skip])
    out = convolution_operation(Connect_Skip, filters)
    return out

def U_Net(Image_Size):
    # Take the image size and shape
    input1 = Input(Image_Size)

    # Construct the encoder blocks
    skip1, encoder_1 = encoder(input1, 64)
    skip2, encoder_2 = encoder(encoder_1, 64*2)
    skip3, encoder_3 = encoder(encoder_2, 64*4)
    skip4, encoder_4 = encoder(encoder_3, 64*8)

    # Preparing the next block
    conv_block = convolution_operation(encoder_4, 64*16)

    # Construct the decoder blocks
    decoder_1 = decoder(conv_block, skip4, 64*8)
    decoder_2 = decoder(decoder_1, skip3, 64*4)
    decoder_3 = decoder(decoder_2, skip2, 64*2)
    decoder_4 = decoder(decoder_3, skip1, 64)

    out = Conv2D(2, 1, padding="same", activation="softmax")(decoder_4)

    model = Model(input1, out)
```

```python
def trainModel():
    #Data Import & Parameters
    orig, msk = load_dataset("dataset/Original/","dataset/Mask/")
    mean_reinhard = (195.11752, 183.79773, 183.8279)
    std_reinhard = (40.50984, 58.086155, 55.982433)
    patch_size = (256,256)
    patch_stride = 240
    #Generate Patchified Preprocessed Images for the Neural Network
    patchified_img_dataset = []
    patchified_msk_dataset = []

    for i, img in enumerate(orig):
        img_rein = reinhard_transformation(img,reference_mean = mean_reinhard, reference_std = std_reinhard)
        img_stacked = generateGEHImageStack(img_rein)
        patchified_img = patchify(img_stacked,patch_size,patch_stride)
        patchified_msk = patchify(msk[i],patch_size,patch_stride)
        for patimg in patchified_img: patchified_img_dataset.append(patimg)
        for patmsk in patchified_msk: patchified_msk_dataset.append(np.expand_dims((patmsk[...,0]/255).astype(np.uint8),2))
        del orig[i]
        del msk[i]
    ##################################################################
    # Convert Patchified Arrays into Numpy Arrays
    patchified_img_dataset = np.array(patchified_img_dataset)
    patchified_msk_dataset = np.array(patchified_msk_dataset)
    ##################################################################
    # Define Input Shape & Model
    input_shape = (256, 256, 3)
    model = U_Net(input_shape)
    img_size = (256, 256)
    num_classes = 1
    batch_size = 5
    epochs = 10
    ###################################################################
    # Compile Model & Define Callbacks
    model.compile(optimizer="rmsprop", loss="sparse_categorical_crossentropy", metrics=['accuracy'])

    callbacks = [
        keras.callbacks.ModelCheckpoint("segModel.keras", monitor='accuracy',mode='max',save_best_only=True)
    ]
    # Train the model
    print("Training")
    hist = model.fit(x=patchified_img_dataset,y=patchified_msk_dataset,batch_size=batch_size,epochs=epochs,verbose='auto',callbacks=callbacks,
```

Codeblock 10: UNet model training (fitting) function

```
def predictModel(imgdir,modelName = "segModel.keras"):
    img = cv2.imread(imgdir)
    img_rein = reinhard_transformation(img,reference_mean = (195.11752, 183.79773, 183.8279), reference_std = (40.50984, 58.086155, 55.982433)
    img_stacked = generateGEHImageStack(img_rein)
    model = load_model(modelName)
    patch_size = 256
    n_classes = 2
    predictions_smooth = predict_img_with_smooth_windowing(img_stacked, window_size=patch_size, subdivisions=2,nb_classes=n_classes,pred_func=
    predictions255 = (predictions_smooth[...,0]*255).astype(np.uint8)
    return img,predictions255
```

Codeblock 11: UNet model prediction function

```
def countCells(img, showimgs = False):
    cle.select_device("cpu-haswell-AMD")
    sigma_spot_detection = 21
    sigma_outline = 3
    segmented = cle.voronoi_otsu_labeling(np.invert(prediction), spot_sigma=sigma_spot_detection, outline_sigma=sigma_outline)
    if showimgs:
        cv2.imshow("Input Mask",prediction)
        cv2.imshow("Segmented",np.array(segmented).astype(np.uint8))
        cv2.waitKey(0)
        cv2.destroyAllWindows()
    return segmented,np.max(np.array(segmented))
```

Codeblock 12: Cell count function

Table of Contents

Introduction: 2
Materials and Methods: 3
    Reinhard transformation: 4
    Channel's generation: Grayscale pseudochannel 8
    Channel's generation: Edges pseudochannel 8
    "Patchification": 9
    Data augmentation: 9
    Smooth Blending: 9
    Instance Segmentation: Voronoi: 9
    Instance Segmentation: Otsu: 10
    Instance Segmentation: Watershed: 10
    Instance Segmentation: pyclEsperanto Library: 10
Results: 11
Discussion: 12
    Reinhard Transformation: 12
    Generating the composite pseudolayers G,E,H: 13
    Image Patchification: 13
    The UNet model: 13
    Smooth blending: 13
    Instance Segmentation: 14
    Metrics and cell counting: 14
Acknowledgement: 14
Citations: 14
Appendix 1 – Code Blocks 16