CENG400
SUMMER PRACTICE REPORT
Department of Computer Engineering
METU

Student's Name:
Kerem Serttaş

Instructor:
Prof. Dr. Murat Manguoğlu

START DATE: 26. 07. 2021
END DATE: 27.08. 2021

Table of Contents

1 INTRODUCTION

I have completed my summer internship at Koc University Summer Research Program. However, due to the Coronavirus pandemic, I have worked remotely rather than visiting the research facility.

During my internship, I have worked with Prof. Çigdem Gündüz Demir on a project called "Deep Learning for Medical Image Analysis". My work mainly focused on developing Convolutional Neural Networks to predict segmentation maps on medical images. I have trained and tested my networks on the “Cell Nucleus Segmentation Dataset for Fluorescence Microscopy Images” dataset and calculated the success of the networks using precision, recall, fscore metrics as well as visually testing whether the segmentation was successful on its premise. I have used the Google Colab platform for training and testing the network. Throughout the project, python programming language and its deep learning libraries such as Tensorflows Keras library is used. In addition, for preprocessing and postprocessing OpenCV library is used as well as Numpy for scientific computation and matplotlib for visualization.

2 INFORMATION ABOUT THE PROJECT

Throughout my internship at Koc University, I have primarily focused on projects to perform the segmentation task at hand. The assignment consisted of designing a Convolutional Neural Network to detect regions of the cells using the “Cell Nucleus Segmentation Dataset for Fluorescence Microscopy Images”. Two different networks can be trained and tested for the task. The first network only produced a segmentation map of the regions of the cells. The second network not only segments the regions of the cells but also separated the cells which touch each other. By doing that, the boundaries of the cells that touch each other can be better segmented.

2.1 Region and Instance Segmentation

Region segmentation task was the initial step to design and implement a Convolutional Neural Network model to perform cell nucleus segmentation task given Fluorescence Microscope images. This task involves training the model with the dataset that contains 61 RGB images of 3329 cell nuclei. At this stage, the model only learns to detect regions of the cell nuclei not regarding nuclei boundaries. As previously mentioned the second network can also detect boundaries to segment each instance of cells.

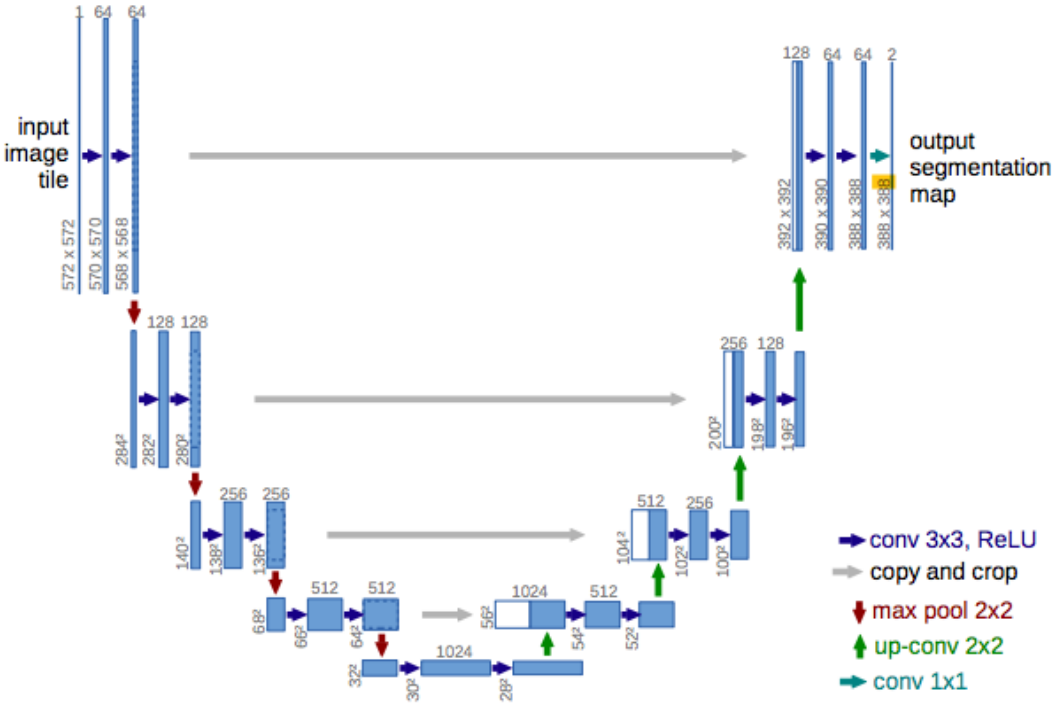
2.1.1 Analysis Phase

The main problem for this phase was to find a suitable architecture. Since the dataset is not large enough to train with a traditional Neural Network and images are from the medical field, I decided to pursue the UNet architecture. Unet Architecture is a perfect fit for our task since it relies on a strong use of data augmentation, which is necessary for our case.

UNET ARCHITECTURE

For many Computer Vision tasks, Convolutional Neural Networks are used. However, in biomedical image processing, the expected output of the network is not a single label. Expected outputs usually contain localizations in the form of a segmentation map, which indicates that each pixel is defined for a specific class.

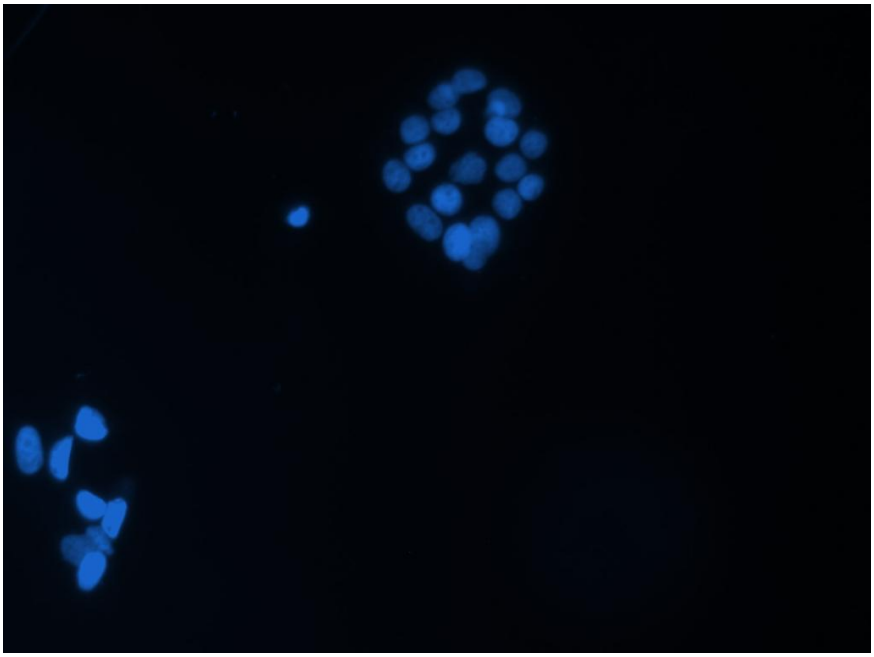
Ronnaberger (2015) proposes a so-called Fully convolutional network called Unet. It consists of an encoder and a decoder path. The encoder path follows the usual architecture of a convolutional neural network which consists of "same" convolutions followed by Rectified Linear Unit(ReLU) activation and Max pooling layers for downsampling. These actions define a step in the network. After each step convolution is saved for concatenating with the decoder path. Then another step starts with the output of the previous step. We double the number of features after each downsampling step. Once we reach the final downsampling step we perform a 1x1 convolution to pass onto the decoder path. A simple step in the decoder path consists of upsampling the previous map then using an up-convolution layer. After we up-convolve feature channels are decreased so that we can concatenate with the corresponding feature map which is saved from the same level encoder path. Finally, we reach the uppermost layer of the decoder path so that a segmentation map is produced with the same shape as the initial input image.



Original Unet Architecture from its paper

2.1.2 Design Phase

After analyzing data and deciding to use Unet architecture to build our model we design the model according to the Unet architecture. Original Unet paper has 5 levels of layers and 572 x 572 x 3 input images. However, our images have 1024 x 768 x 3 shape therefore we need to restructure the number of levels of layers. We decided to start with 4 levels of layers. However, due to the nature of this project, we decided to design the code as changeable, starting with a single level up to 9 levels deep. More details will be provided in the implementation phase.
After deciding the number of layers, we need to define the number of features for each convolution operation. In this phase, we need to inspect the sizes of the cell nuclei and decide how big we need to calculate and see for each convolution. After calculating with 1024 x 768 shaped images, we then found out that the best numbers were multiples of 45 such as 45,90,180 …



Example image from the dataset

Furthermore, we need to decide on the optimizer which calculates and updates the gradient. For that task, we decided to choose and experiment with 3 different optimizers which are RMSProp, Adadelta, and Adam. For the learning rate for each optimizer, we decided to experiment with 0.01, 0.001, and 0.0001.

We standardize each Convolution operation with (3 x 3) filter size, 'same' padding, and 'reLU' as the activation function. On each layer, first comes the convolution layer then a dropout layer and then a second convolution layer, and finally a max-pooling layer with pooling size as (2 x 2).

Another important aspect to consider while designing the network is to decide which loss function to use and to decide how big each pixel affects calculating the total loss. We decided to use categorical cross-entropy but give weight to each pixel according to its class such that background pixels have less effect than the foreground pixels. Since we use categorical cross-entropy to calculate the loss, we decide to use the 'softmax' activation function for the output layer of the model.

2.1.3 Implementation Phase

We decided to use python as the programming language of this project because of its powerful but simple data analysis tools, machine learning, and deep learning libraries.

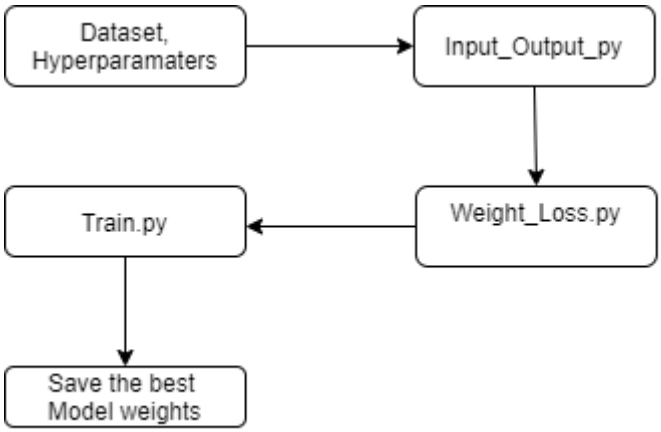
Keras library was used to implement the project as described in the design phase. Code was separated into different files to encapsulate each functionality.

One python file consisted of managing the input-output of the model by reading the dataset inputs and ground truth values accordingly and organizing them in a way that it can be used to run with different datasets easily. Note that input images consist of 3 channeled RGB images, normalized to mean of 0 and standard deviation of 1 for each channel values separately. Ground truth values consist of 1 channeled images that only contain 2 values for pixels, where a pixel value of 0 means background and a pixel value of 1 means foreground.

After managing input, ground truth pairs another file implements a different type of weights to give each pixel of the Keras.model's loss calculation. One particular method gives more weight to the foreground pixels since background pixels are a lot more dominant in the real segmentation map we use as the ground truth value.

Since we managed input-output pairs and loss weights for each input pixel, we then implement a python file for managing training and testing parameters, early stopping, maxEpoch, and batch sizes.

Lastly, a python file defines and handles the unet model itself, how many layers it will have, how many features it will have, and defines the hyperparameters of the model itself such as optimizer types and learning rate. The model implementation is made such that it can work with 1-9 layers and a different number of features for each layer as well as different number of channels to build and train the model.



Training pipeline

An example training pipeline is shown above. The dataset is initially fed to the model with the model hyperparameters. Then input_output.py manages the input-output normalization and preprocessing phase. Then, depending on the chosen weighted loss

option, a corresponding weight for each pixel weight is calculated and propagated into the Keras model. Then Train.py sets up epoch number, validation steps, and model checkpoints, set according to the chosen settings. After that step, we propagate all that into the model itself and start training. At each epoch, validation loss is calculated and the best model weights are saved. If the model stops improving after a certain number of epochs, it exits.

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input (InputLayer)	[(None, 768, 1024, 3	0	
conv2d (Conv2D)	(None, 768, 1024, 45	1260	input[0][0]
dropout (Dropout)	(None, 768, 1024, 45	0	conv2d[0][0]
conv2d_1 (Conv2D)	(None, 768, 1024, 45	18270	dropout[0][0]
max_pooling2d (MaxPooling2D)	(None, 384, 512, 45)	0	conv2d_1[0][0]
conv2d_2 (Conv2D)	(None, 384, 512, 90)	36540	max_pooling2d[0][0]
dropout_1 (Dropout)	(None, 384, 512, 90)	0	conv2d_2[0][0]
conv2d_3 (Conv2D)	(None, 384, 512, 90)	72990	dropout_1[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 192, 256, 90)	0	conv2d_3[0][0]
conv2d_4 (Conv2D)	(None, 192, 256, 180	145980	max_pooling2d_1[0][0]
dropout_2 (Dropout)	(None, 192, 256, 180	0	conv2d_4[0][0]
conv2d_5 (Conv2D)	(None, 192, 256, 180	291780	dropout_2[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 96, 128, 180)	0	conv2d_5[0][0]
conv2d_6 (Conv2D)	(None, 96, 128, 360)	583560	max_pooling2d_2[0][0]
dropout_3 (Dropout)	(None, 96, 128, 360)	0	conv2d_6[0][0]
conv2d_7 (Conv2D)	(None, 96, 128, 360)	1166760	dropout_3[0][0]
max_pooling2d_3 (MaxPooling2D)	(None, 48, 64, 360)	0	conv2d_7[0][0]
conv2d_8 (Conv2D)	(None, 48, 64, 720)	2333520	max_pooling2d_3[0][0]
dropout_4 (Dropout)	(None, 48, 64, 720)	0	conv2d_8[0][0]
conv2d_9 (Conv2D)	(None, 48, 64, 720)	4666320	dropout_4[0][0]
up_sampling2d (UpSampling2D)	(None, 96, 128, 720)	0	conv2d_9[0][0]
concatenate (Concatenate)	(None, 96, 128, 1080	0	up_sampling2d[0][0] conv2d_7[0][0]
conv2d_10 (Conv2D)	(None, 96, 128, 360)	3499560	concatenate[0][0]
dropout_5 (Dropout)	(None, 96, 128, 360)	0	conv2d_10[0][0]
conv2d_11 (Conv2D)	(None, 96, 128, 360)	1166760	dropout_5[0][0]
up_sampling2d_1 (UpSampling2D)	(None, 192, 256, 360	0	conv2d_11[0][0]
concatenate_1 (Concatenate)	(None, 192, 256, 540	0	up_sampling2d_1[0][0] conv2d_5[0][0]
conv2d_12 (Conv2D)	(None, 192, 256, 180	874980	concatenate_1[0][0]
dropout_6 (Dropout)	(None, 192, 256, 180	0	conv2d_12[0][0]
conv2d_13 (Conv2D)	(None, 192, 256, 180	291780	dropout_6[0][0]
up_sampling2d_2 (UpSampling2D)	(None, 384, 512, 180	0	conv2d_13[0][0]
concatenate_2 (Concatenate)	(None, 384, 512, 270	0	up_sampling2d_2[0][0] conv2d_3[0][0]
conv2d_14 (Conv2D)	(None, 384, 512, 90)	218790	concatenate_2[0][0]

dropout_7 (Dropout)	(None, 384, 512, 90) 0	conv2d_14[0][0]
conv2d_15 (Conv2D)	(None, 384, 512, 90) 72990	dropout_7[0][0]
up_sampling2d_3 (UpSampling2D)	(None, 768, 1024, 90 0	conv2d_15[0][0]
concatenate_3 (Concatenate)	(None, 768, 1024, 13 0	up_sampling2d_3[0][0] conv2d_1[0][0]
conv2d_16 (Conv2D)	(None, 768, 1024, 45 54720	concatenate_3[0][0]
dropout_8 (Dropout)	(None, 768, 1024, 45 0	conv2d_16[0][0]
conv2d_17 (Conv2D)	(None, 768, 1024, 45 18270	dropout_8[0][0]
input_1 (InputLayer)	[(None, 768, 1024)] 0	
out1 (Conv2D)	(None, 768, 1024, 2) 92	conv2d_17[0][0]

Total params: 15,514,922
Trainable params: 15,514,922
Non-trainable params: 0

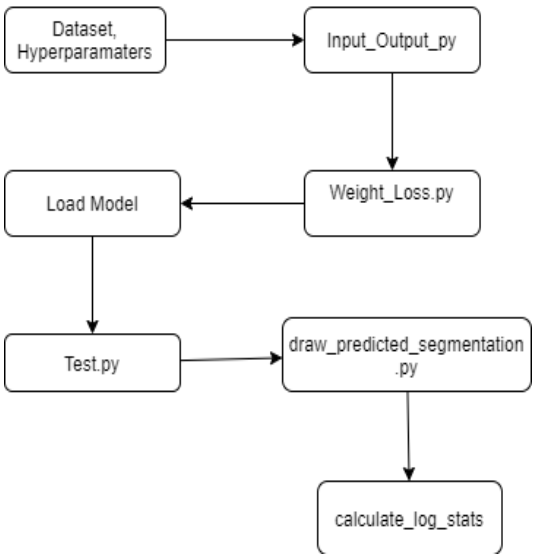
An Example Model Summary

An example model summary is shown above with 5 layers deep unet architecture as well as the corresponding array of features for each layer as the following [45,90,180,360,720].

As mentioned above, a python file manages all training and testing phases, depending on whether it's in the training mode or testing mode. We wanted to use the same input-output and weight loss calculation phases for both training and testing pipelines to prove that there is no bias, however of course different datasets are used for training validating, and testing the model.

2.1.4 Testing Phase

Testing is a vital part of this project. In the design and implementation phase, testing methods and metrics are described.

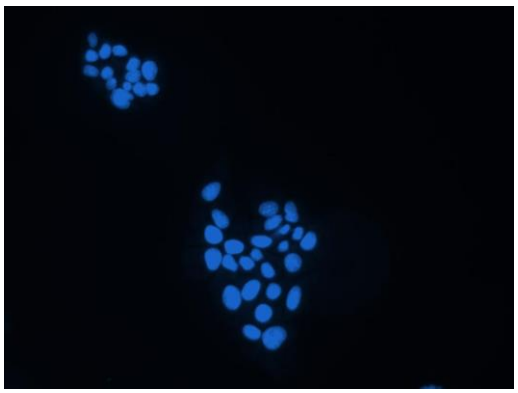


Testing Pipeline

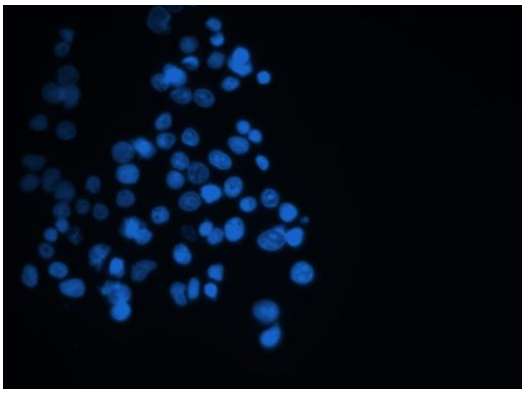
An example testing pipeline is shown above. The dataset is fed into the model first, along with the model hyperparameters. The input-output normalization and preprocessing phase are then handled by input output.py. The corresponding weight for each pixel weight is then calculated and propagated into the Keras model, depending on the weighted loss option selected. After that, we load the previously trained model and start the testing phase. After the model produces the output segmentation maps, the pipeline calls the draw_predicted_segmentation.py. This python file produces the visual PNG files of the predicted segmentation. We can visually evaluate the success of the model using these outputs. Next, pipeline calls calculate_log_stats.py python file to calculate the precision, recall, and f score of each predicted segmentation map about the expected segmentation map. After this calculation, we can examine the success of the model statistically.

For the same task, we present you 2 different models and their corresponding outputs for the two datasets; huh7, hepG2

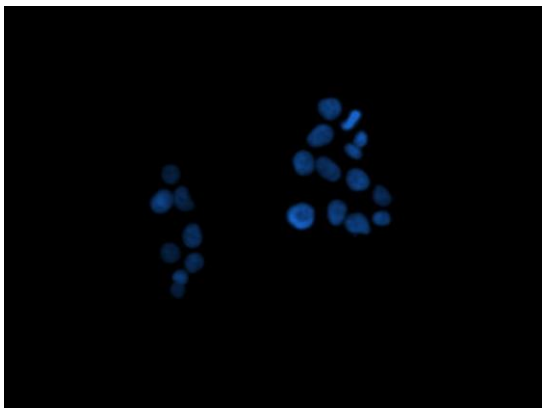
Huh7 Dataset 3 Example inputs RGB:



Huh7-ts1



Huh7-ts2

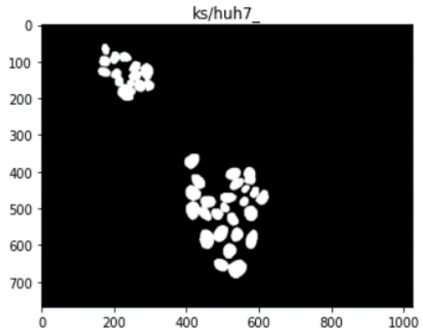


Huh7-ts5

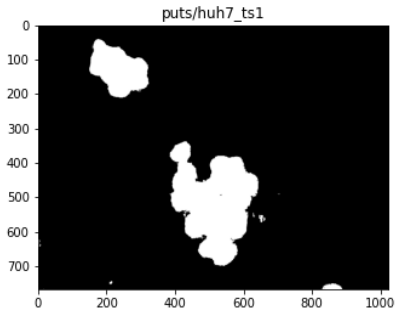
Now we present first the originally expected segmentation map, then 2 different models corresponding output. The first image in each row is the expected map, and the second

image in each row is the corresponding out that the model generates. Only 3 example pictures are given for the means of simplicity.

Model 1 results with huh7 sample ts1:

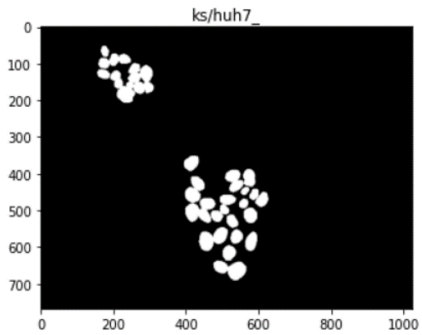


Expected Map

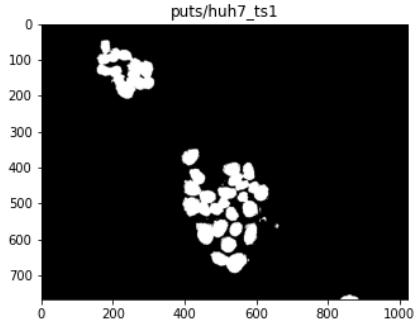


Predicted Map

Model 2 with huh7 sample ts 1:

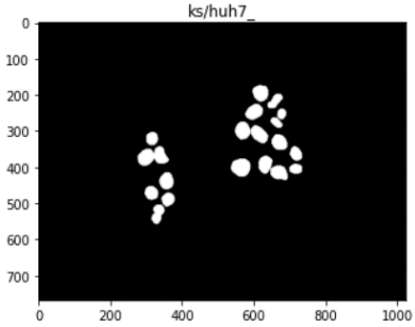


Expected Map

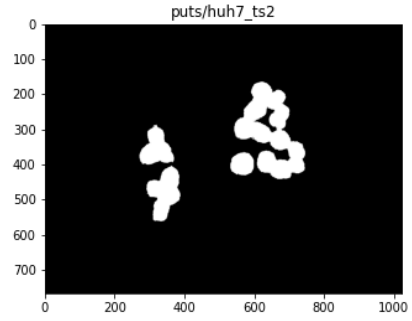


Predicted Map

Model 1 with huh7 sample ts 2:

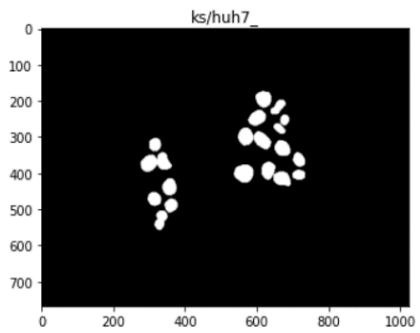


Expected Map

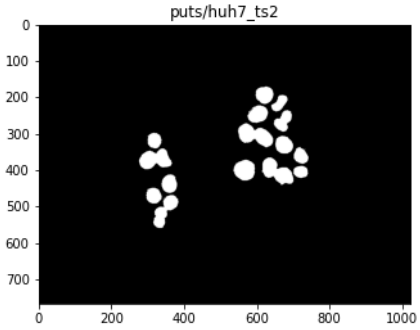


Predicted Map

Model 2 with huh7 sample ts 2:

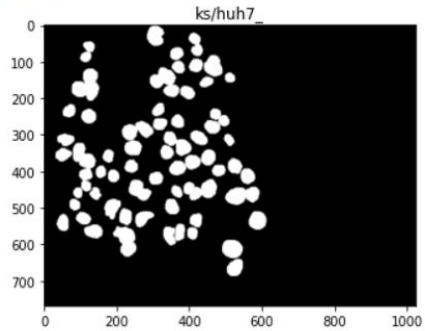


Expected Map

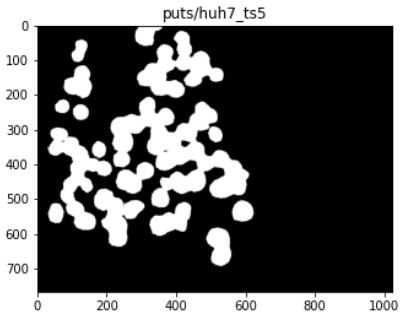


Predicted Map

Model 1 with huh7 sample ts 5:

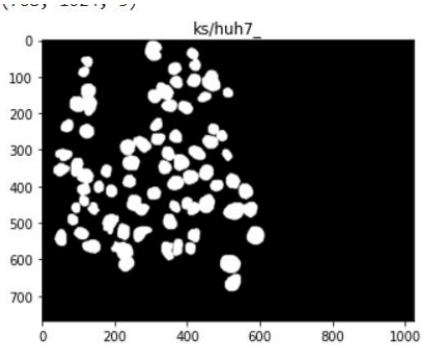


Expected Map

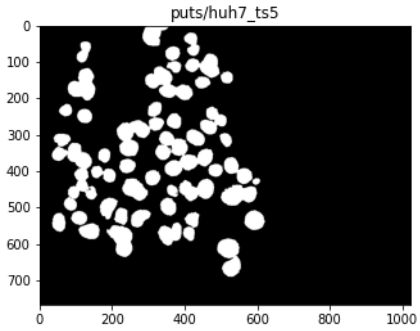


Predicted Map

Model 2 with huh7 sample ts 5:

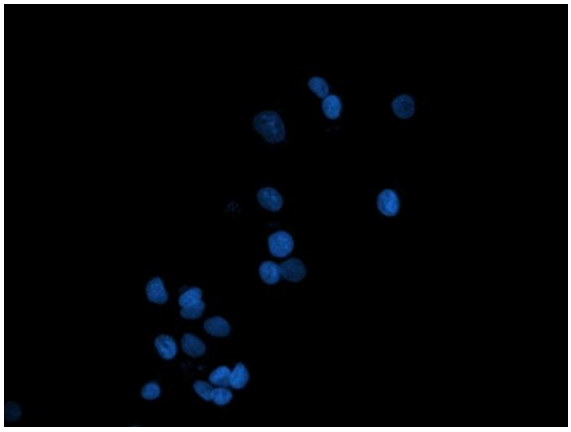


Expected Map

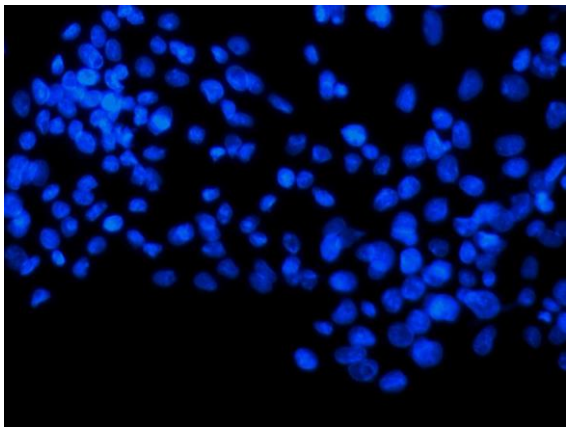


Predicted Map

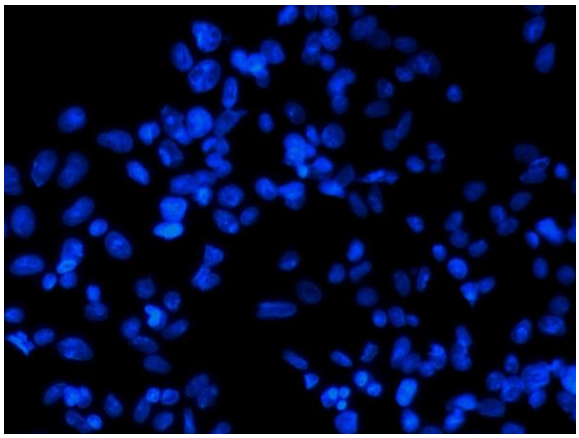
HepG2 Dataset 3 Example inputs RGB:



HepG2 ts-4



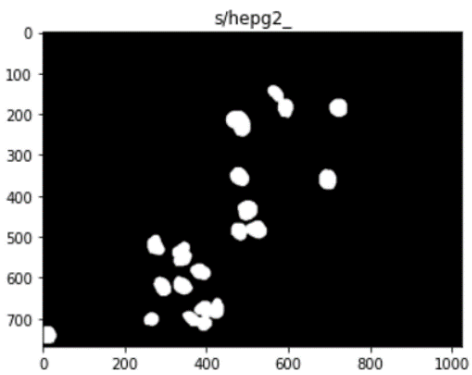
HepG2 ts-14



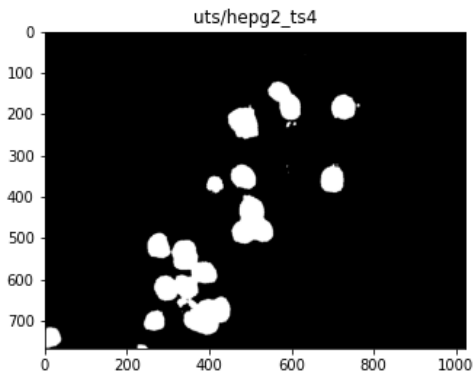
HepG2 ts-15

Now we present first the originally expected segmentation map, then 2 different models corresponding output. The first image in each row is the expected map, and the second image in each row is the corresponding out that the model generates.

Model 1 results with hepG2 sample ts4:

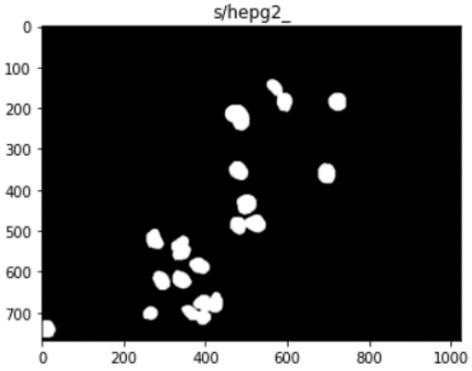


Expected Map

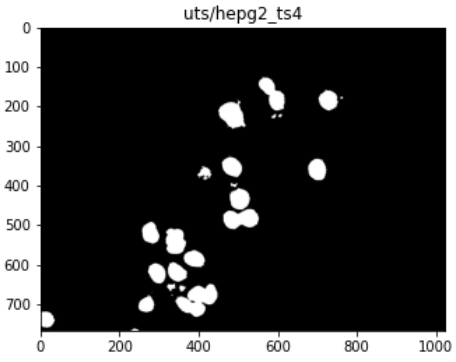


Predicted Map

Model 2 results with hepG2 sample ts4:

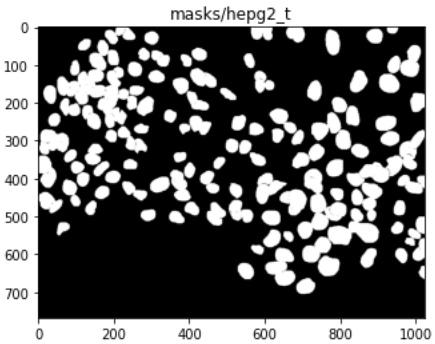


Expected Map

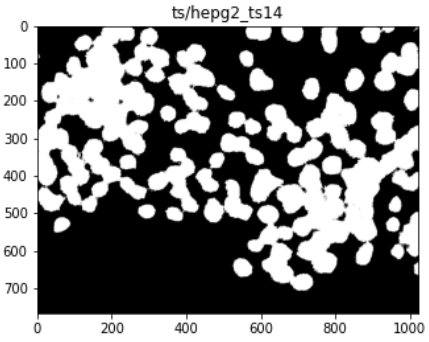


Predicted Map

Model 1 results with hepG2 sample ts14:

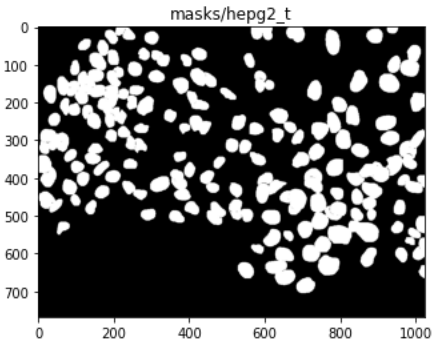


Expected Map

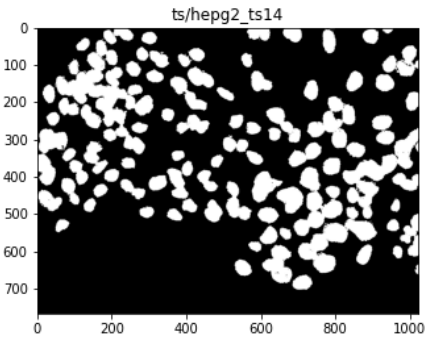


Predicted Map

Model 2 results with hepG2 sample ts14:

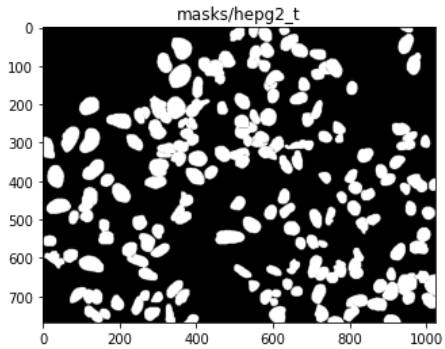


Expected Map

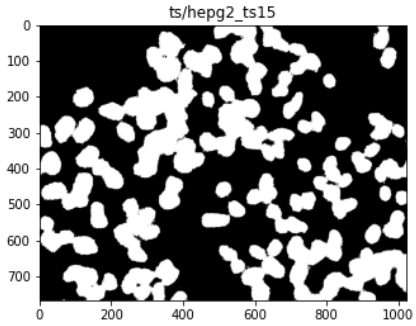


Predicted Map

Model 1 results with hepG2 sample ts15:

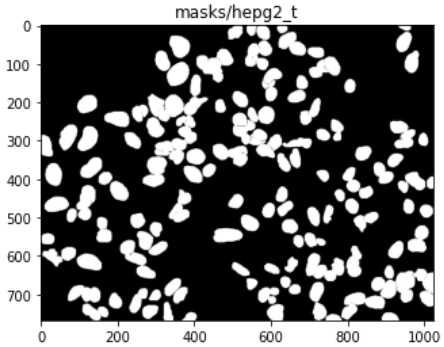


Expected Map

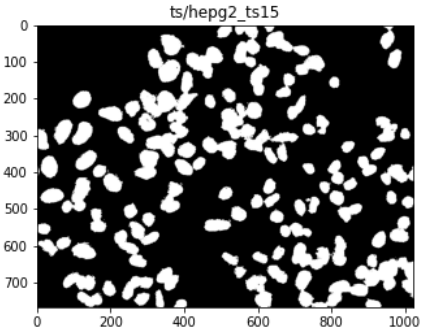


Predicted Map

Model 2 results with hepG2 sample ts15:



Expected Map



Predicted Map

Huh7 Dataset Statistics:

Models	Huh7 Precision Mean %	Huh7 Recall Mean %	Huh7 F Score Mean %	Huh7 Precision Standard Deviation %	Huh7 Recall Standard Deviation %	Huh7 F Score Standard Deviation %
Model1	61.24	99.71	75.17	5.74	0.65	4.44
Model2	84.20	97.37	90.19	4.77	2.37	2.06

HepG2 Dataset Statistics:

Models	HepG2 Precision Mean %	HepG2 Recall Mean %	HepG2 F Score Mean %	HepG2 Precision Standard Deviation %	HepG2 Recall Standard Deviation %	HepG2 F Score Standard Deviation %
Model1	52.28	99.92	67.15	16.17	0.14	17.52
Model2	75.28	97.52	82.89	19.97	2.13	18.09

3 METHODOLOGIES AND STRATEGIES USED IN THE INSTITUION

Koc University VPRD(Office of Vice President for Research and Development) Research Internship is designed for motivated undergraduates from various universities who desire to strengthen their research skills in preparation for graduate school. The program allows undergraduates to get research experience that will help them determine whether they want to continue graduate school or a career in professional research.

Throughout my internship, I worked closely with prof. Cigdem Gunduz Demir. Initially, she taught us the basics of neural networks and medical image segmentation. We were 4 students from different universities. We work to learn and improve our current understanding of the project and the subject.

4 CONCLUSION

At the beginning of my internship, I had no experience in the image segmentation field. I have never maintained a machine learning pipeline. Before working on an actual project, I developed simple projects on the image segmentation field to adapt. Throughout my internship, I realized that developing on a machine learning environment needs many experimenting. It is important to start with an educated guess and intuition, however, each dataset and each problem needs specific care.
Finally, being part of this amazing environment was a great pleasure. I would like to thank Prof. Çiğdem Gündüz Demir and Koc University VPRD.

5 REFERENCES

1 - Ronneberger, O. (2015, October 5). *U-Net: Convolutional Networks for Biomedical Image Segmentation*. SpringerLink. https://link.springer.com/chapter/10.1007/978-3-319-24574-4_28?error=cookies_not_supported&code=ca902b16-12b5-4ba1-ad36-2531fc667f2b